SHARP: Shared State Reduction for Efficient Matching of Sequential Patterns

Cong Yu*
Aalto University
cong.yu@aalto.fi

Tuo Shi*
Aalto University
tuo.shi@aalto.fi

Matthias Weidlich
Humboldt-Universität zu Berlin
weidlima@hu-berlin

Bo Zhao
Aalto University
bo.zhao@aalto.fi

ABSTRACT

The detection of sequential patterns in data is a basic functionality of modern data processing systems for complex event processing (CEP), OLAP, and retrieval-augmented generation (RAG). In practice, the respective engines typically evaluate multiple shared patterns simultaneously, in order to improve the result quality for downstream applications. The evaluation of a large number of patterns under tight latency bounds is challenging, though, since matching needs to maintain *state*, i.e., intermediate results, that grow exponentially in the input size. Hence, systems turn to best-effort processing, striving for maximal recall under a latency bound. Existing techniques, however, consider patterns in isolation, neglecting the optimization potential induced by state sharing and corresponding interactions and interference across shared patterns.

We describe SHARP, a state management library that employs *state reduction* for efficient best-effort pattern matching in shared patterns. To this end, SHARP incorporates state sharing between patterns through a new abstraction, coined pattern-sharing degree (PSD). At runtime, PSD facilitates the categorization and indexing of partial pattern matches. Once a latency bound is exceeded, SHARP realizes best-effort processing by using a cost model to select a subset of partial matches for further processing in constant time. In experiments with real-world data, SHARP achieves a recall of 95%, 93% and 72% for pattern matching in CEP, OLAP, and RAG applications, under a bound of 50% of the average processing latency.

PVLDB Reference Format:
Cong Yu, Tuo Shi, Matthias Weidlich, and Bo Zhao. SHARP: Shared State Reduction for Efficient Matching of Sequential Patterns. PVLDB, 19(5): 987 - 1000, 2026. doi:10.14778/3796195.3796210

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/benyucong/SHARP.

1 INTRODUCTION

The detection of sequential patterns with low latency is a data management functionality with a wide range of applications: Complex event processing (CEP) engines detect user-defined patterns over high-velocity event streams [6, 14, 59]; online analytical processing (OLAP) systems evaluate queries featuring the `MATCH_RECOGNIZE` operator that takes a set of tuples as input and returns all matches of the given pattern [24, 27, 82]; and graph databases evaluate patterns of regular path queries over knowledge graphs to facilitate retrieval-augmented generation (RAG) [3, 8, 47]. In all these applications, patterns define the order of data elements along with the correlation and aggregation predicates over their attribute values.

Pattern matching is challenging, though: applications enforce strict latency bounds [27, 71, 81] as part of the service level objectives (SLO) [31]. At the same time, the evaluation is computationally hard, since it requires maintaining *state*, i.e., partially matched patterns, which grow exponentially in the size of the input [23, 30, 47, 77]. Due to these challenges, exhaustive pattern evaluation becomes infeasible, especially in short peak times of increased computational load [22, 69]. Systems therefore resort to best-effort processing: they strive to maximize the number of detected pattern matches, while satisfying a latency bound [12, 61, 63, 64, 81].

In practice, the above mentioned challenges are amplified by the fact that many applications require the simultaneous evaluation of multiple *shared patterns*. The reason being that the result quality of many downstream applications [20, 25, 33, 59] can directly be improved by evaluating a set of similar, yet different patterns. We illustrate how shared patterns improve the quality of results and the underlying performance challenges with an example of graph retrieval-augmented generation (GraphRAG), as follows.

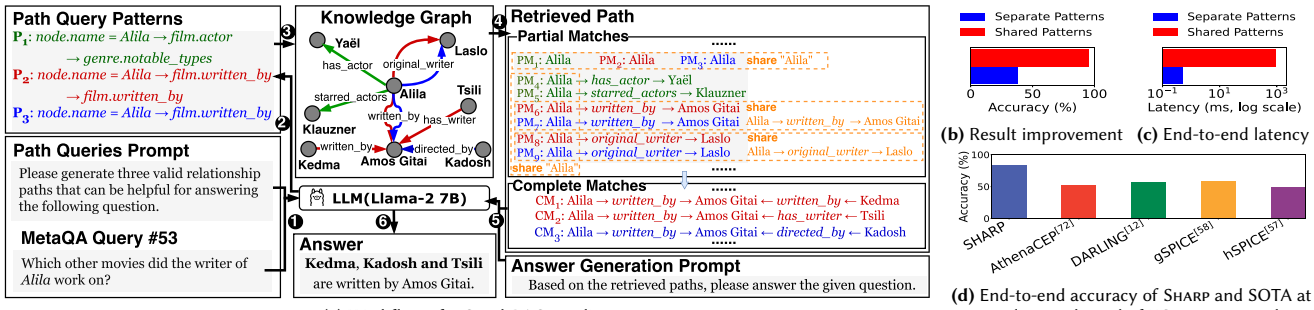
Consider an application in which the inference of a large language model (LLM) is augmented using an external knowledge graph (KG), as shown in Fig. 1a: ❶ The application submits (i) a question for the LLM (from the MetaQA benchmark [79]) and (ii) a query prompt to generate patterns of path queries for the KG. ❷ The patterns and prompts are sent to the LLM (fine-tuned on `WebQSP` [75] and `CWQ` [79]). The LLM then generates a set of path query patterns and selects the top-k (P_1, P_2 and P_3), e.g., using cosine similarity and beam-search [68]. ❸ The path queries are evaluated on the KG and ❹ the resulting patterns are ❺ integrated in an answer generation prompt for ❻ the final response. Here, P_1, P_2 and P_3 share sub-patterns and one snapshot of shared state is illustrated as *partial matches* $\{PM_1, PM_2, \ldots, PM_9, \ldots\}$ in ❹.

Compared to the use of single, separate patterns, shared patterns improve the result quality by 2.80× in accuracy and 2.51× in recall[1] (over 14,872 questions in MetaQA), see Fig. 1b. For one question in ❶, query #53, the LLM only generates incorrect answers (i.e., LLM hallucination) if not using the shared patterns. Yet, the improvement incurs a cost: Pattern matching latency in the KG ❹ increases by *four orders of magnitudes* (see Fig. 1c)[2] compared to the use of a single pattern, as 3,962× more partial matches are generated. Even when adopting optimizations for state sharing [4, 5],

*Both authors have contributed equally.

Proceedings of the VLDB Endowment, Vol. 19, No. 5 ISSN 2150-8097.
doi:10.14778/3796195.3796210

[1] Recall is the ratio of correct LLM responses to the number of ground-truth answers.
[2] Here, the end-to-end processing latency comprises LLM generation (325 ms) and KG pattern matching (1,079 ms); the latter being the performance bottleneck.



(a) Workflow of a GraphRAG pipeline

(b) Result improvement (c) End-to-end latency

(d) End-to-end accuracy of SHARP and SOTA at 300 ms latency bound of KG pattern matching

Fig. 1: An example of how shared patterns enhance the end-to-end responses of GraphRAG and the underlying performance challenge

the number of generated partial matches still increases by 2,001×. In large-scale industrial applications, such as those reported for GraphRAG by Microsoft [39, 56], Amazon [9], and Siemens [60], such computational challenges are further amplified.

Existing approaches for best-effort pattern matching, however, are limited in their effectiveness as they treat each pattern in isolation. Specifically, load shedding techniques [12, 61, 63, 64, 81] that discard input data (DARLING [12], hSPICE [63] and gSPICE [64]), partial matches (pSPICE [61] and [80]), or a combination of both (AthenaCEP [81]) take shedding decisions for each pattern separately. While systems such as pSPICE [61], hSPICE [63] and gSPICE [64] incorporate operators that are part of multiple patterns, their utility assessment is done for each single pattern in isolation. By ignoring the interaction among multiple patterns via their shared state, these techniques neglect a significant optimization potential.

In this paper, we study how to realize best-effort processing of pattern workloads, when incorporating state sharing in their evaluation. This is difficult as the state (i.e., partial matches) of pattern matching affects the results and performance as follows:
(1) Partial matches differ in how they contribute to the result of a single pattern and in their computational resources (i.e., latency).
(2) Partial matches differ in their importance for shared multiple patterns, which may be captured further by a cost model.
(3) The relation between partial matches and patterns is subject to changes, e.g., due to concept drift in data distributions, and therefore, requires efficient indexing mechanisms to track this.

We describe **SHARP**a state management library for efficient best-effort pattern matching with shared state. It addresses the above challenges and overcomes the limitations of state-of-the-art approaches. Fig. 1d highlights that, under a latency bound of 300 ms (in KG pattern matching), the state of the art achieves only an accuracy below 60% for the aforementioned pattern matching scenario. In contrast, SHARP incorporates optimizations for the shared state across multiple patterns, boosting accuracy to around 85%. These improvements are facilitated by the following contributions:

(1) Efficient pattern-sharing assessment (§4.1). SHARP captures state sharing per partial match using a new abstraction called *pattern-sharing degree* (PSD). PSD keeps track of how different patterns share a partial match in terms of overlapping sub-patterns and enables efficient lookup of this information for an exponentially growing set of partial matches. The structure of the PSD is derived from the pattern execution plan. At runtime, SHARP relies on this structure to cluster the generated partial matches and, through a bitmap-based index, enables their retrieval in constant time.

(2) Efficient cost model for state selection (§4.2). For each partial match, SHARP examines its contribution to all patterns and its computational overhead, which determines the processing latency. To achieve this, SHARP maintains a cost model to estimate the number of complete matches that each partial match may generate, as well as the runtime and memory footprint caused by it. SHARP updates the cost model incrementally and facilitates a lookup of the contribution and the overhead per partial match in constant time.

(3) State reduction problem formulation (§3) and its optimization (§4.3). We formulate the problem of satisfying latency bounds in pattern evaluation as a *state reduction* problem: Upon exhausting a latency bound, SHARP selects a set of partial matches for further processing based on a multi-objective optimization problem. SHARP limits the overhead of solving this problem by a *hierarchical selection* lifting the pattern-sharing degree and the cost model to a coarse granularity by clustering, and by employing a greedy approximation strategy for the multi-objective optimization space.

We implemented **SHARP** in C++ and Python (3.5K LoC) and evaluated it (§5) on three pattern-matching applications: complex event processing (CEP) over event streams, OLAP with MATCH_RECOGNIZE queries, and GraphRAG. Across real-world datasets, SHARP attains shared-pattern matching recall of 95%, 93%, and 72% for CEP, OLAP, and GraphRAG under a 50% bound on average processing latency. Compared to existing state management strategies, SHARP improves recall by 11.25× (CEP), 2.4× (OLAP), and 2.1× (GraphRAG).

2 FOUNDATIONS OF PATTERN MATCHING

2.1 Data and Execution Models

Input data of pattern matching is a sequence of data elements $S = \langle d_1, d_2, \dots \rangle$. Each $d_i = \langle a_1, \dots, a_m \rangle$ is an instance of a schema defined by a finite sequence of attributes. We further define a prefix of the data sequence S up to index k, $S(\dots k) = \langle d_1, \dots, d_k \rangle$ and the suffix starting k as $S(k \dots) = \langle d_k, d_{k+1}, \dots \rangle$.

Pattern P defines a sequence of data that satisfy a set of predicates C (a time window, the sequential order, and value constraints over attributes). For instance, P_1 in Fig. 1a ❸ defines the path sequence of three nodes in KG and the predicates, e.g., node.name=Alila. Pattern matching evaluates a set of patterns $\mathbf{P} = \{P_1, \dots, P_n\}$ over S, ensuring P_i is detected within its latency bound L_i (based on SLO).

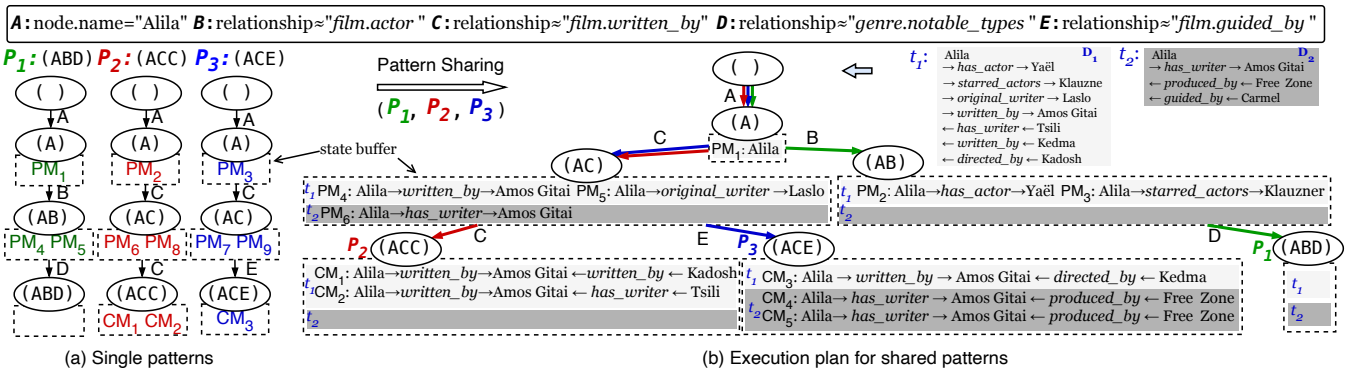


Fig. 2: The execution plan DAG^b of (a) separate single patterns and (b) multiple shared patterns

Complete matches are created by evaluating P over S=$\langle d_1, d_2, \ldots \rangle$, each being a subsequence $\langle d'_1, \ldots, d'_m \rangle$ of S that preserves the sequential order, i.e., for $d'_i = d_k$ and $d'_j = d_l$, it holds that $i < j$ implies $k < l$, and satisfies all predicates in C. We denote complete matches of pattern P_i over $S(\ldots k)$ as $\texttt{CM}_{P_i}(k)$, and the complete matches for all patterns as $\texttt{CM}(k) = \bigcup_i \texttt{CM}_{P_i}(k)$. Fig. 1a ❺ shows a snapshot of complete matches for P_2 (CM_1 and CM_2) and P_3 (CM_3).

Partial matches (PMs) are sub-sequences of complete matches. Their sequences of data elements strictly satisfy the time window and sequential order, but only satisfy a subset of predicates in C. We write $\texttt{PM}(k)$=$\{\langle d_1, \ldots, d_j \rangle, \ldots, \langle d'_1, \ldots, d'_l \rangle\}$ for the set of PMs after evaluating patterns $\mathbf{P}$ over $S(\ldots k)$. Fig. 1a ❹ shows PMs for all patterns PM_{1-9} in corresponding colors.

Execution plan of a pattern P_i is represented as a directed acyclic graph with buffers (DAG^b), G_i, where the path from the *starting vertex* to an *ending vertex* constructs complete matches. Fig. 2a shows bespoke execution plans for P_1, P_2 and P_3. Each node of G_i represents a sub-pattern SP of P_i and maintains a buffer to store the intermediate results, partial matches, in SP. The edge (SP, SP') represents a transition between sub-patterns and is guarded by P_i's predicates C_i. When processing an input d_{k+1}, the pattern matching engine checks if d_{k+1} and all PMs stored in SP, satisfy predicates C_i. Then, it activates the corresponding state transitions and generates new PMs and complete matches.

The DAG^b model is general enough to capture bespoke automata- or tree-based execution models [36, 74] for different applications (see §2.2). Formally, pattern execution is a *function* that takes an element d_{k+1}, the current PMs $\texttt{PM}(k)$, and the execution plan G as input, and outputs new PMs, $\texttt{PM}(k+1)$ and complete matches $\texttt{CM}(k+1)$, ensuring the latency below predefined bounds for all patterns:

$$f(d_{k+1}, \texttt{PM}(k), G) \mapsto \{\texttt{PM}(k+1), \texttt{CM}(k+1)\} \texttt{s.t.} \forall P_i \in \mathbf{P}, \texttt{Latency}(P_i) \leqslant L_i$$

Pattern sharing merges pattern execution plans to reuse overlapping sub-patterns. As shown in Fig. 2b, P_1, P_2 and P_3 share sub-pattern (A), while P_2 and P_3 also share sub-pattern (AC). The evaluation is similar to single patterns, by checking state transition predicates between maintained PMs and the input data. (AC) maintains two PMs, i.e., PM_4 and PM_5, at time t_1. When processing the two input data elements in D_1, i.e., "← written_by ← Kedma" and "← has_writer ← Tsili", PM_4 derives two complete matches (CM_1, CM_2) by satisfying the predicates in edges written_by and has_writer. The state (AC) then transitions to the state (ACC).

2.2 Need for Shared-State in Pattern Matching

2.2.1 Pattern Matching Applications. We target the fundamental problem of pattern matching, which is the backbone of several categories of data processing systems: those for (i) complex event processing (CEP), (ii) OLAP with the MATCH_RECOGNIZE operator, and (iii) GraphRAG. While each category induces different domain-specific applications, the data and execution models defined in §2.1 are general enough for these applications, as explained below.

(i) CEP detects predefined patterns in unbounded streams of events with low latency [6, 14, 59]. Each event is a tuple of attribute values. Patterns are defined as a combination of event types, operators, predicates, and a time window. These operators include conjunction, sequencing (SEQ) that enforces a temporal order of events, Kleene closure (KL) that accepts one or more events of a type, and negation (NEG) that requires the absence of specific-typed events.

Two execution models have been proposed for CEP. (1) In the automata-based model [74], PMs denote partial runs of an automaton that encodes the required event occurrences. The state transitions are guarded by predicates to check if partial runs advance in the automaton. (2) In the tree-based model [36], events are inserted into a hierarchy of buffers that are guarded by predicates. The evaluation then proceeds from the leaf buffers of the tree to the root, filling operator buffers with derived PMs.

Both automata- and tree-based models are covered by our DAG^b model (§2.1). Automata and trees are specific forms of a DAG^b. Fundamentally, the DAG^b captures the essence of pattern matching: *the state and state transitions*. CEP's *selection and consumption policies* determine how many data elements can be skipped to select and if they can be reused [11, 13, 74, 83]. In this paper, we target the most challenging configuration of skip-till-any selection + reuse consumption (see details in our technical report [76])

(ii) OLAP queries with MATCH_RECOGNIZE clause [24] perform pattern matching on the rows (i.e., tuples) of a table or view. Many platforms have supported this, including Oracle, Apache Flink, Azure Streaming Analytics, Snowflake, and Trino [7, 19, 28, 48, 65]. MATCH_RECOGNIZE specifies types of rows based on their attribute values and defines a pattern as a regular expression over these types. The pattern is then evaluated for a certain partition of the input tuples. OLAP systems also specify whether matches may overlap and how the result is constructed per match. Common execution models for MATCH_RECOGNIZE pattern matching are based

on automata, which are covered by the DAG^b model in §2.1. Depending on the structure of the regular expression, a deterministic or non-deterministic automaton is constructed, which is then used to process tuples while scanning the table or view used as input.

(iii) GraphRAG enhances the inference capabilities of RAG systems by integrating external knowledge graphs (KG), particularly for rich relationships between entities [39, 56]. As shown in Fig. 1a, this is achieved by evaluating the patterns of *regular path queries* on the KG [33, 45, 67]. Here, the input of patterns is the tuples of nodes representing entities and the edges that reflect semantic relationships between them. A pattern specifies a sequence of edges i.e., a regular expression over edge labels, to retrieve structures of complex relations and contextual dependencies encoded in the KG. Again, the common execution model is automata-based, searching the edges of the KG till accepting states in the automata. This is also captured by our DAG^b model defined in §2.1.

2.2.2 Quality Enhancement via Shared Patterns. The use of multiple shared patterns promises to significantly improve the query quality in CEP [35, 51, 54], OLAP [41, 82], and GraphRAG [4, 33, 45, 67]. This is because sharing state allows several patterns to *collaboratively* generate more insights in higher-level semantics.

Consider the GraphRAG example in Fig. 1, the use of shared patterns enables the LLM to generate correct responses for all 14,872 questions in the MetaQA benchmark [79], improving the accuracy by 2.80×. In contrast, only using separate patterns in isolation failed to provide correct responses for 11,897 questions.

In particular, question #53 requires multi-hop paths in the KG to first query the writer of `Alila`, i.e., Amos Gitai, and then retrieve other movies that are directed or written by him. Since the label of edges in the KG may be different than that specified in single patterns, for instance, `written_by` and `has_writer` are different labels but semantically equivalent. Shared patterns can leverage semantic similarity to evaluate a set of overlapping patterns (that only differ in the edge label) simultaneously. As a result, when using separate patterns, LLM reports "`I apologize, but I don't think we discussed a movie called Alila or its scriptwriter`".

Although such enhancement can also be achieved by evaluating bespoke patterns first, buffering the matches, and aggregating them, the processing latency becomes unacceptable. In our experiments, it takes 23 seconds (the shared pattern is finished in 1,079 ms).

Similar quality enhancements are reported in industrial-scale use cases. Amazon employs GraphRAG to improve response accuracy by 35% in financial reports and vaccine documents [9]. Neo4j leverages GraphRAG to monitor and optimize supply chain management for a leading global automotive manufacturer [10, 43].

2.3 Challenges and the Design Space for Best-Effort Processing on Shared Patterns

2.3.1 Challenges. The evaluation of common pattern matching presents the challenge of exponential computational complexity. Zhang et al. [77] have proved exponential complexity in CEP patterns with Kleene closure operators. In the same vein, Huang et al. [23] showed that `MATCH_RECOGNIZE` presents exponential runtime complexity with Kleene closure. Previous work [30, 38, 47] has proven that the evaluation of a regular path query is NP-hard. Therefore, pattern-matching systems turn to best-effort processing to maximize the result quality, i.e., the number of complete matches, while satisfying a latency bound.

Existing best-effort approaches fall short in exploiting the optimization space of shared states across a set of patterns. In particular, they discard selected input data [12, 63, 64, 81] or partial matches [61, 80, 81] to reduce processing latency; however, they fail to capture the interactions and inferences of shared partial matches between shared patterns. As a result, existing best-effort mechanisms cannot keep high quality within tight latency bounds.

Figure 1d highlights such limitations. At the 300 ms latency bound, input-based approaches achieve the accuracy of 56% (`DARLING` [12]). While the hybrid approach achieves 53% (`AthenaCEP` [81]). Their low accuracy performance is due to the focus on optimizing a single pattern. `pSPICE` [61], `hSPICE` [63] and `gSPICE` [64] indeed consider a CEP operator in multiple pattern queries in which each query is assigned a weight in advance. Yet, they do not consider the interaction and interference of shared patterns. Instead, they calculate the utility of events to discard with respect to separate patterns in isolation, by multiplying the pattern's predefined weight. As a result, `hSPICE` and `gSPICE` only achieve 53% and 60% accuracy.

2.3.2 Design Space. We explore the design space of best effort pattern matching with shared state. The fundamental problem is that the state of pattern matching, i.e., partial matches, has very different impact on the result quality and performance in *three dimensions*:

(1) *For bespoke patterns*, partial matches contribute differently in constructing complete matches while consuming various computational resources (i.e., increase the processing latency). For instance, some partial matches generate more partial matches (longer subpatterns) and consume lots of CPU cycles and memory footprint, but will not lead to complete matches due to the selection of query predicates. Fig. 2b illustrates this dimension. For pattern P_2, partial match PM_4 (in state `(AC)`'s buffer) contributes two complete matches, CM_1 and CM_2 (in state `(ACC)`'s buffer). In contrast, PM_5 has already consumed computational effort i.e., state transitions till `(AC)`'s buffer, but does not generate any complete matches.

(2) *An individual partial match* may have varying impacts on multiple shared patterns: it may generate more complete matches for certain patterns than others in the same set of shared patterns. As demonstrated in Fig. 2b, PM_4 generates two complete matches, CM_1 and CM_2 for P_2, but only one complete match, CM_3 for P_3. Similarly, PM_6 generates two complete matches, CM_4 and CM_5 for P_3, but none for P_2. In addition, patterns may have various weights/utilities/priorities defined by user applications, which further complicates the impact of a single partial match on the final results.

(3) The impact between partial matches and shared patterns changes dynamically due to concept drift in payload value distribution, especially in data streams. This means that pattern matching engines and optimization techniques shall adapt to such dynamics. For instance, in Fig. 2b, at time point t_1, both state `(AC)` and `(AB)` have two PMs and `(ACC)` has two complete matches while `(ACE)` has one. That is equal resource consumption for PMs in `(AC)` and `(AB)`, but PMs contribute more to P_2 than P_3. However, when processing new data elements in D_2, at t_2, one more PM is generated at `(AC)` while two more complete matches are generated at `(ACE)`. Now,

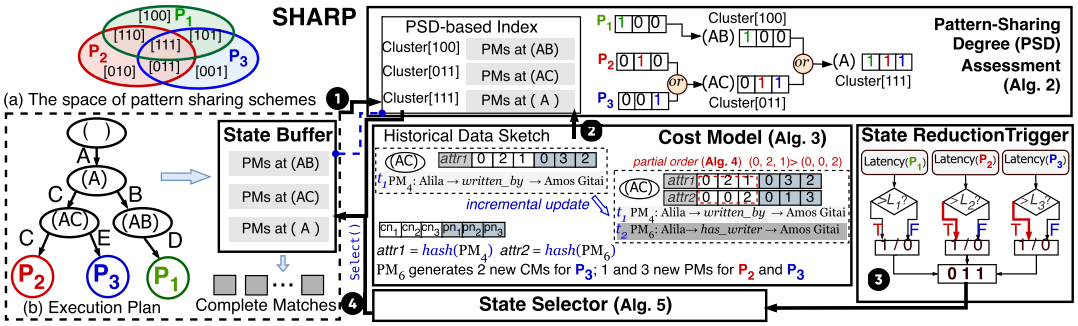


Fig. 3: System architecture of SHARP (the exmaple execution plan is identical to Fig. 2)

Algorithm 1: SHARP workflow

Input: Input data d_{k+1} and sequence prefix $S(\ldots k)$, execution plan G, patterns $\mathbf{P}$, complete matches $\texttt{CM}(k)$ and partial matches $\texttt{PM}(k)$.

Output: A subset of partial matches $\texttt{PM'(k)}$ to be processed.

1 $\mathbf{C} = \texttt{PSD}(G, \mathbf{P})$; // Assess Pattern-Sharing Degree (Alg.2)

2 $\mathbf{Q} = \texttt{Cost}(\texttt{CM}(k), \mathbf{P}, \texttt{PM}(k))$; // Calculate cost model (Alg.3)

3 $\mathbf{C}$.Partial_Order($\mathbf{Q}$) ; // Compute state partial order (Alg.4)

// Scan input data and perform state reduction

4 **while** d_{k+1} *arrives* **do**

5 **if** $\forall P_i \in \mathbf{P},\ \exists l_{i,k+1} \geqslant L_i$; // Trigger state reduction

6 **then** $\texttt{PM'(k)}$=Select($\mathbf{C}, \{P_i\}_{l_{i,k+1} \geqslant L_i}$) ; // Select a subset of state for pattern matching (Alg.5)

// Incrementally update PSD indexing and the cost model

7 **while** *a new match* ρ' *arrives* **do** $\mathbf{C}$.insert(ρ'), $\mathbf{Q}$.update(ρ');

8 **return** $\texttt{PM'(k)}$;

PMs in (AC) consume more computational resources than those in (AB) while PMs contribute more to P_3.

3 RESEARCH PROBLEM FORMULATION

In this paper, we design a *state reduction* mechanism that selects a subset of partial matches to process for shared patterns. The goal is to maximize results quality while satisfying bespoke latency bounds for all patterns. We denote $\texttt{CM}_{P_i}(j)$ and $\texttt{CM}'_{P_i}(j)$ as the complete matches without and with *state reduction*. Then, $\delta_{P_i}(k) = \sum_{1 \leqslant j \leqslant k} |\texttt{CM}_{P_i}(j) \backslash \texttt{CM}'_{P_i}(j)|$ is the *recall loss* for pattern P_i. We formulate the *state reduction* as a multi-objective optimization problem:

Problem 1. Given a prefix sequence $S(\ldots k)$, an execution plan G, partial matches $\texttt{PM}(k)$, and the latency bounds L_i for each pattern $P_i \in \mathbf{P}$, the state reduction problem for the best-effort pattern matching is to select a subset $\texttt{PM'}(k) \subset \texttt{PM}(k)$, such that the recall loss $\delta_{P_i}(k)$ is minimized for all $P_i \in \mathbf{P}$, respecting the latency bounds for all patterns, $\texttt{Latency}(P_i) \leqslant L_i$.

4 SHARP DESIGN

SHARP's goal is to realize the *state reduction* for best-effort processing in shared patterns. Its design is based on the analysis of the corresponding challenges and the design space to optimize the interactions and inferences of the shared state (§2.3). To this end, SHARP designs a new abstraction to capture the interactions and inferences of the shared state, and maintains a cost model to select partial matches for further processing to satisfy the latency bounds.

Fig. 3 illustrates the architecture of SHARP, while Alg.1 explains its workflow. First, SHARP uses a new abstraction of ❶ *Pattern-Sharing Degree* (PSD) to capture state sharing schemes across several patterns (line 1 in Alg.1). To efficiently manage the dynamically changing state, SHARP builds a bitmap-based indexing mechanism for PSD to cluster partial matches for fast lookup and updates.

It then employs a ❷ *cost model* to efficiently and effectively access partial matches' contribution to the final complete matches and their computational overhead (line 2 in Alg.1). The cost model ensures that SHARP always selects the most promising subset of partial matches to process (line 3 in Alg.1), i.e., state reduction.

SHARP monitors the processing latency for patterns, upon exceeding a latency bound (i.e., we call this overload), ❸ the *state reduction trigger* generates an *overload label* encoded in a bitmap (line 5 in Alg.1). After this, SHARP's ❹ *state selector* uses this overload label and the PSD-based indexing to instantly locate state buffers that are related to violated latency bounds, and select a subset of partial matches to proceed with pattern matching (line 6 in Alg.1). Lastly, SHARP incrementally updates the PSD and the cost model (line 7 in Alg.1). SHARP repeats the above steps until the processing latency is below the latency bound.

Next, we explain the details of the pattern-sharing degree (§ 4.1), the cost model (§ 4.2), and optimizations of the state selector (§ 4.3).

4.1 Pattern-Sharing Degree

The *pattern-sharing degree* (PSD) is SHARP's abstraction to capture how different patterns share the state through overlapping sub-patterns. We design PSD to be expressive enough to cover the full space of pattern-sharing schemes. For n patterns the number of all possible sharing combinations is 2^n and SHARP's PSD uses an n-bit vector to systematically encode the entire space of sharing schemes.

Example 1. Fig. 3a shows the space of sharing schemes for patterns, P_1, P_2 and P_3: eight possible combinations in different colors. SHARP uses three bits to encode this space: [111] denotes the state shared by all patterns (i.e., sharing degree of three), while [011] denotes those only shared by P_2 and P_3 (i.e., sharing degree of two).

Note that SHARP takes an input execution plan for shared patterns. For a specific execution plan, the number of sharing schemes is already determined. The upper bound is linear—the sum of all pattern lengths (i.e., no sharing at all). Fig. 3b shows the execution

Algorithm 2: Pattern-sharing degree (PSD) assessment

Input: Evaluation Plan G, patterns $\mathbf{P}$ and partial matches $\texttt{PM}(k)$.
Output: PSD-indexed partial match clusters $\mathbf{C}$.

1 **for** $P_i \in \mathbf{P}$ **do** $\texttt{PSD}(P_i) \leftarrow 2^{n-i-1}$; // $\forall P_i$, set the i^{th} bit to 1
// PSD Assessment through Depth-First Traversal of G
2 **for** $P_i \in \mathbf{P}$ ***and*** *there is a reachable state* SP *in* G **do**
3 $\texttt{PSD}(SP) \leftarrow \texttt{PSD}(SP) \vee \texttt{PSD}(P_i)$; // Update the PSD of SP
// PSD-based Indexing
4 $\mathbf{C} \leftarrow$ *Clustering sub-patterns with the same PSD*;
5 $\forall C(b') \in \mathbf{C}, C(b') = \{\rho \in \texttt{PM}(k) | \texttt{PSD}(\rho) = b'\}$;
6 **return** $\mathbf{C}$;

plan with six states. Two of them contain shared states, i.e., P_1, P_2 and P_3 share (A), while P_2 and P_3 share (AC).

To address the design space (see §2.3) of state reduction for shared patterns, we design Sharp's PSD to (i) incorporate how each partial match is shared across different patterns, (ii) efficiently retrieve all partial matches for bespoke shared patterns—based on corresponding sharing schemes, and (iii) efficiently adapt to dynamic changes between partial matches and shared patterns.

To this end, we build bitmap structures to represent the sharing degree of each sub-pattern. This structure enables CPU bitwise instructions to efficiently manage PSD—reducing the system overhead of PSD. For a pattern P_i, the bitmap for each sub-pattern SP (and each PM ρ in SP) is an n-bit array $\texttt{PSD}(SP)$ (and $\texttt{PSD}(\rho)$), where the i-th bit is set to 1 if SP is a sub-pattern of P_i, and 0 otherwise.

Algorithm 2 shows the process to construct PSD as follows: First, Sharp assigns each pattern P_i an initial bitmap $\texttt{PSD}(P_i) = 2^{n-i-1}$, where only the i-th bit is 1 (line 1). Sharp then traverses the execution plan G in a depth-first manner to search all sub-patterns of each P_i (line 2). If there exists reachable state SP in G, SP is shared by P_i. Sharp computes the bitmap for each sub-pattern SP, $\texttt{PSD}(SP) = \bigvee_{SP' \in SP.\text{succ}} \texttt{PSD}(SP')$. That is applying a bitwise OR operator across the bitmaps of its successor sub-patterns.

After constructing the PSD for the execution plan G, Sharp organizes partial matches with the same bitmap b' into a cluster $C(b') = \{\rho \in \texttt{PM}(k) | \texttt{PSD}(\rho) = b'\}$ (lines 4-5). Whenever a new partial match is generated, Sharp indexes it to the corresponding cluster. Due to the efficiency of bitwise operating instructions, PSD indexing locates the partial matches via a bitmap in $O(1)$ time complexity. Here, the number of clusters is bounded by the number of states (i.e., nodes) in the execution plan G. We illustrated the above process of PSD assessment (Alg. 2) with the following example.

Example 2. For the execution plan in Fig. 3b Sharp (line 1) assigns each pattern an initial bitmap: $\texttt{PSD}(P_1)$=[100], $\texttt{PSD}(P_2)$=[010], and $\texttt{PSD}(P_3)$=[001]. Sharp then (line 2) computes the bitmap for each sub-pattern by performing OR across the bitmaps of its successor sub-patterns: $\texttt{PSD(AB)}=\texttt{PSD}(P_1)$, $\texttt{PSD(AC)}=\texttt{PSD}(P_2) \vee \texttt{PSD}(P_3)$=[010] $\vee$[001]=[011], $\texttt{PSD(A)}=\texttt{PSD(AC)} \vee \texttt{PSD(AB)}$=[100] $\vee$[011]=[111]. Finally (lines 4-5), Sharp groups partial matches into clusters C([100]), C([011]), and C([111]) based on their bitmaps.

4.2 Cost Model

The goal of the cost model is to assess (i) how "promising" a partial match is for the shared patterns—the number of complete matches it can contribute and (ii) how "expensive" a partial match is for its entire lifespan—the computational overhead (resource consumption) it incurred. In addition, (iii) the cost model calculation must be lightweight. Because the latency bounds have already been violated, the cost model shall not introduce extra overhead in the first place.

4.2.1 Definition of the Cost Model. For a partial match $\rho \in PM(k)$, the cost model accesses its ***contribution*** to the final results of shared patterns and the computational ***overhead*** for final and intermediate results during pattern matching.

Contribution. A partial match ρ may result in complete matches for multiple shared patterns. Sharp captures ρ's *contribution* to pattern P_i as the number of complete matches that are generated by ρ. We define the contribution of ρ up to a time point k' as

$$\Delta^{+}_{P_i}(\rho) = |\{\rho' \in \texttt{CM}_{P_i}(k') | \rho \text{ generates } \rho'\}|, \quad (1)$$

where $\texttt{CM}_{P_i}(k')$ is the complete matches of pattern P_i over $S(\ldots k')$. We define the contribution of ρ to all patterns in $\mathbf{P}$ as a *vector*

$$\vec{\Delta}^{+}_{\mathbf{P}}(\rho) = [\Delta^{+}_{P_1}(\rho), \ldots, \Delta^{+}_{P_n}(\rho)].$$

Overhead. Sharp considers a partial match ρ's computational overhead as the resource consumption caused by ρ itself and all its derived partial matches and complete matches in the future. In addition, partial matches in different states consume different computational resources (i.e., CPU cycles and memory footprint) due to predicates' complexity and the size of the partial match itself (i.e., the length of a sub-pattern). We capture this through a function $\Theta(\rho) \mapsto r$, $r \in \mathbb{N}^{+}$ that maps ρ's computational overhead to a real number. Users can materialize Θ based on their specific applications. The computational overhead of ρ to pattern P_i is

$$\Delta^{-}_{P_i}(\rho) = \sum_{\rho' \in \bigcup \texttt{PM}_{P_i}(k') \wedge \rho \text{ generates } \rho'} \Theta(\rho'), \quad (2)$$

$\texttt{PM}_{P_i}(k')$ is the set of partial matches of P_i over $S(\ldots k')$. We define the computational overhead of ρ to all patterns in $\mathbf{P}$ as a *vector*

$$\vec{\Delta}^{-}_{\mathbf{P}}(\rho) = [\Delta^{-}_{P_1}(\rho), \ldots, \Delta^{-}_{P_n}(\rho)].$$

Example 3. For the partial match PM_4 in Fig. 3 ❷, its contribution and overhead to P_1, P_2, P_3 at time t_1 are: $\vec{\Delta}^{+}_{\mathbf{P}}(PM_4) = [0, 2, 1]$ and $\vec{\Delta}^{-}_{\mathbf{P}}(PM_4) = [0, 3, 2]$, as it generates two complete matches (CM_1, CM_2) for P_2 and one (CM_3) for P_3, while generating three (PM_4, CM_1, CM_2) and two (PM_4, CM_3) partial/complete matches for P_2 and P_3, respectively. (Here, for simplicity, we assume $\Theta(\rho) = 1$.)

4.2.2 Efficient Estimation of the Cost Model. The materialization of the cost model, $\vec{\Delta}^{+}_{\mathbf{P}}$ and $\vec{\Delta}^{-}_{\mathbf{P}}$, requires *future* statistics (e.g., the number of generated partial matches in runtime), which can only be computed in retrospect. To address this issue, Sharp designs an efficient estimation for the cost model based on historical statistics of pattern matching. That is, the number of partial and complete matches generated by each partial match in the previous time window or time slice [81]. The rationale is that the fresh history of runtime statistics may predict the counterparts in the near future [29, 81].

Sharp maintains a *historical data sketch* for each state buffer in the execution plan to efficiently monitor and estimate the number of future complete and partial matches generated per partial match. The sketch contains multiple entries, each summarizing the number of partial matches with the same attribute values. Sharp represents each entry as a vector $[attr, cn_1, \ldots, cn_n, pn_1, \ldots, pn_n]$ where $attr$ is the *attribute key* value. Sharp uses a hash function to derive

Algorithm 3: Cost model estimation

Input: Patterns $\mathbf{P}$ and partial matches $\texttt{PM}(k)$.
Output: The contribution $\Delta^+_{P_i}(\rho)$ and the computational overhead $\Delta^-_{P_i}(\rho)$ of each $\rho \in \texttt{PM}(k)$ to all patterns $P_i \in \mathbf{P}$.

// Incremental Update of the Historical Data Sketch
1 **if** *a newly generated match* ρ' *arrives* **then**
  // Update the complete/partial match counting
2 **for** ρ *that generates* ρ' **do**
3 $attr = \texttt{hash}(\rho)$; // Get the attribute key
4 **if** ρ' is a complete match **then** $\texttt{Sketch}[attr].cn_i$++;
5 **if** ρ' is a partial match **then** $\texttt{Sketch}[attr].pn_i$++;
// Look-up contribution and computational overhead
6 **while** *evaluating* $\rho \in \texttt{PM}(k)$ **do**
7 $attr = \texttt{hash}(\rho)$; // Get the attribute key
8 $\forall P_i \in \mathbf{P}, \Delta^+_{P_i}(\rho) = \texttt{Sketch}[attr].cn_i$; // Contribution
9 $\forall P_i \in \mathbf{P}, \Delta^-_{P_i}(\rho) = \texttt{Sketch}[attr].pn_i \times \Theta(\rho)$; // Overhead
10 **return** $[\Delta^+_{P_i}(\rho)]_{P_i \in \mathbf{P},\ \rho \in \texttt{PM}(k)}, [\Delta^-_{P_i}(\rho))]_{P_i \in \mathbf{P},\ \rho \in \texttt{PM}(k)}$;

the attribute key of ρ, i.e., $attr = \texttt{hash}(\rho)$. pn_i and cn_i denote the number of partial and complete matches of pattern P_i generated from all partial matches associated with $attr$. PM_4 in Example 3 is hashed to $attr_1$: [$attr_1$, cn_1=0, cn_2=2, cn_3=1, pn_1=0, pn_2=3, pn_2=2].

For efficiency, SHARP *incrementally* updates the historical data sketch, as illustrated in Alg.3, lines 1-5. When a new match ρ' of pattern P_i is generated, SHARP updates the corresponding entries based on the hashed attribute key. Specifically, (i) if ρ' is a complete match, SHARP updates the counter cn_i=cn_i + 1, and (ii) if ρ' is a partial match, it increases the partial match counter pn_i = pn_i + 1. For instance, in Fig. 3 ❷, when PM_6 is created at t_2, a new entry $attr_2$ is constructed: [$attr_2$, cn_1=0, cn_2=0, cn_3=2, pn_1= 0, pn_2= 1, pn_2=3]. cn_i and pn_i are updated accordingly.

SHARP incrementally updates the historical data sketches: new matches refresh statistics while stale estimations fade—either via a sliding time window (e.g., CEP) or updated statistics by newer observations (e.g., GraphRAG). This design enables lightweight maintenance of data sketches and adaptiveness to concept drifts.

SHARP estimates the cost model using the updated historical data sketch (Alg. 3, lines 6–9). For a partial match ρ, SHARP first hashes the *attribute key*, i.e., $attr = \texttt{hash}(\rho)$, and then looks up the historical data sketch to locate the entry $[attr, cn_1, \ldots, cn_n, pn_1, \ldots, pn_n]$. The contribution and computational overhead of ρ to pattern P_i are estimated as $\Delta^+_{P_i} = cn_i$ and $\Delta^-_{P_i} = pn_i \times \Theta(\rho)$.

The design of the *historical data sketch* enables the estimation and lookup in $O(1)$ time complexity. In practice, users may also customize the hash function (e.g., based on application-specific value distribution) to avoid hash collisions for the efficacy of the cost model estimation.

4.2.3 Partial Order of Partial Matches. To facilitate efficient state reduction in §4.3, SHARP organizes partial matches in a partial order based on the cost model. This allows SHARP to efficiently and approximately select partial matches for state reduction (see §4.3.2)

The partial order must consider that a partial match may contribute differently to multiple shared patterns. In Fig. 3 ❷, PM_4 contributes to P_2 and P_3 with contributions [cn_1=0, cn_2=2, cn_3=1]. In contrast, PM_6 only contributes to P_3 with [cn_1=0, cn_2=0, cn_3=2].

Algorithm 4: Cost model-based partial order of state

Input: Two partial matches ρ and ρ', and their contributions, $\Delta^+_{P_i}(\rho)$ and $\Delta^+_{P_i}(\rho')$, to all patterns $P_i \in \mathbf{P}$.
Output: The partial order between ρ and ρ'.
1 **if** $\forall P_i \in \mathbf{P},\ \Delta^+_{P_i}(\rho) > \Delta^+_{P_i}(\rho')$ **then return** $\rho > \rho'$;
2 **else if** $\sum_i \Delta^+_{P_i}(\rho) > \sum_i \Delta^+_{P_i}(\rho')$ **then return** $\rho > \rho'$;
3 **else return** $\rho \leqslant \rho'$;

Algorithm 4 outlines how SHARP computes such a partial order. First (line 1), SHARP compares the contribution of two partial matches, ρ and ρ', on each pattern P_i. If $\forall P_i \in \mathbf{P},\ \Delta^+_{P_i}(\rho) > \Delta^+_{P_i}(\rho')$, ρ is better than ρ'. If that does not hold (line 2), SHARP compares the total contribution of ρ and ρ'. If $\sum_i \Delta^+_{P_i}(\rho) > \sum_i \Delta^+_{P_i}(\rho')$, ρ is better than ρ'. Otherwise (line 3), ρ is worse than ρ'.

EXAMPLE 4. For PM_4 and PM_6 in Fig. 3 ❷, since $PM_4.cn_2 > PM_6.cn_2$ and $PM_4.cn_3 < PM_6.cn_3$, SHARP cannot determine their order in line 1 of Alg. 4. SHARP then compares their total contributions (line 2). The total contribution of PM_4 is $PM_4.cn_2$ + $PM_4.cn_3$ = 3, which is greater than that of PM_6, i.e., $PM_6.cn_3$ = 2. Thus, $PM_4 > PM_6$.

For efficient implementation of the partial order, SHARP maintains partial matches in a max-heap structure [73] for state buffers. It is constructed during the initialization of PSD. Updating the max-heap is sublinear time complexity [73], which is negligible compared to the overhead of the evaluation of pattern matching itself.

4.3 State Selector

SHARP's *state selector* carefully chooses a subset of partial matches to process, when the *state reduction trigger* is activated by overloading, i.e., processing latency exceeds the pre-defined bound (e.g., SLO).

4.3.1 Problem Formulation. We formulate the partial match selection as a *multi-objective optimization problem* to decide *what* and *how many* partial matches to select, based on SHARP's *cost model*.

The goal is to select a set of partial matches $\rho \in \texttt{PM'(k)}$ with the highest total contribution. That is to maximize $\sum_{\rho \in \texttt{PM'(k)}} \vec{\Delta}^+_{\mathbf{P}}(\rho) = [\Delta^+_{P_1}, \ldots, \Delta^+_{P_n}]$. At the same time, for each pattern P_i, the processing latency incurred by $\texttt{PM'(k)}$, $\texttt{Latency}(P_i)$, must be below the latency bound $\texttt{LatencyBound}_i$.

Note that the processing latency $\texttt{Latency}(P_i)$ is caused by the computational overhead of partial matches, $\Delta^-_{P_i}$. Reducing the overhead results in lower latency. This means that the latency bound indicates the upper-bound capacity of computational overhead that is allowed in pattern matching. Based thereon, $\texttt{PM'(k)}$'s total computational overhead must be below $\frac{\texttt{LatencyBound}_i}{\texttt{Latency}(P_i)} \Delta^-_{P_i}$.

Therefore, we formulate the state selection in shared patterns as the following multi-objective optimization problem.

$$\max \quad \sum_{\rho \in \texttt{PM'(k)}} \vec{\Delta}^+_{\mathbf{P}}(\rho) = [\Delta^+_{P_1}, \ldots, \Delta^+_{P_n}] \tag{3}$$
$$\texttt{s.t.} \ \sum_{\rho \in \texttt{PM'(k)}} \texttt{PSD}(\rho)[i] \cdot \Delta^-_{P_i}(\rho) \leqslant \frac{\texttt{LatencyBound}_i}{\texttt{Latency}(P_i)} \Delta^-_{P_i}, \forall P_i \in \mathbf{P}$$

$\texttt{PSD}(\rho)[i]$ represents the pattern sharing degree: if ρ is shared by P_i, $\texttt{PSD}(\rho)[i] \geqslant 1$, indicting that ρ's computational overhead should

Algorithm 5: State selection

Input: Overload label b_{OL}, PSD-indexed partial match clusters **C**.
Output: A set of partial matches $\mathsf{PM}'(k)$ to be processed.

```
// Select partial matches related to non-overloaded patterns
1 PM'(k) ← ⋃_{PSD': PSD' (bitmap index) AND b_OL=0} C(PSD');
// Select partial matches related to overloaded patterns
2 while the count constraint in Eq. (3) is holding do
    // Select the highest quality PM based on the PSD
3   PSD = max{PSD: PSD (bitmap index) AND b_OL ≠ 0};
4   ρ = C(PSD).pop(), PM'(k) ← PM'(k) ∪ {ρ};
5 return PM'(k);
```

be counted under P_i. The objective function, $\sum_{\rho \in \mathsf{PM}'(k)} \vec{\Delta}^{+}_{\mathbf{P}}(\rho)$, is a vector of contributions for patterns in **P**, with various or even conflicting utilities and therefore, different optimization goals. i.e., multi-objective optimization.

The above problem is NP-hard via reduction from the multi-dimensional multi-objective knapsack problem [15, 34], where each element corresponds to a partial match and the knapsack capacity corresponds to the computational overhead. It is impractical to solve it online in the already overloaded pattern-matching engine with violated latency bounds. Therefore, we design an efficient implementation for SHARP's state selector to approximate the solution (§4.3.2).

4.3.2 Efficient Implementation in SHARP. The design of the state selector is based on the idea of *hierarchical selection*. Specifically, SHARP must (i) first select all partial matches from state buffers that are not associated with patterns with violated latency bounds using PSD (i.e., not causing overload), (ii) efficiently locate the affected state buffers and retrieve the partial matches using PSD, and (iii) efficiently select the partial matches from affected buffers based on both PSD and the partial order of the cost model.

To achieve (i) and (ii), SHARP employs the bitmap-based *overload label* in the state reduction trigger (§4, Fig. 3 ❸) to connect PSD's bitmap index to instantly locate the unaffected state buffers (i.e., not related to latency violation) while retrieving partial matches from the affected buffers. Specifically, we define the overload label b_{OL} as an n-bit array (i.e., bitmap). Whenever a pattern $P_i \in \mathbf{P}$ violates its latency bound, SHARP's state reduction trigger sets b_{OL}'s i-th bit to 1.

The overload label and PSD bitmap index are both n-bitmaps. SHARP performs the bitwise AND between the overload label and the PSD index to locate unaffected state buffers (i.e., the result is a zero bitmap). This is because the i-th bit in PSD index means if the state is shared by the i-th pattern P_i. The state selector then selects all partial matches in unaffected state buffers without computing Eq. (3) (see Alg. 5, line 1). SHARP then locates the affected state buffers in constant time, based on non-zero results from the AND operation.

To realize (iii), select partial matches in affected state buffers, SHARP uses a *greedy* approach (Alg. 5, line 3) based on the partial order of the cost model and the PSD of partial matches. In particular, SHARP selects partial matches in *the order of PSD values*. A larger PSD value means the state is shared by more patterns and conveys a larger contribution value.

Within a state buffer (Alg. 5, line 4), SHARP greedily selects partial matches based on the partial order of the cost model. That is, SHARP always prioritizes the selection of partial matches with the highest contribution value, until reaching the upper bounds of the overhead capacity of partial matches (defined in Eq.3), resulting in the linear complexity (compared to the NP-hard in Eq.3). In addition, such selection is performed at the attribute cluster level, instead of the PM instance level, which significantly further reduces the search space.

EXAMPLE 5. When P_2 and P_3 violate their latency bounds, SHARP sets the overload label b_{OL} =[011] (see Fig. 3 ❸) and uses PSD indexing to identify affected states via AND. (AB) is unaffected since it is not shared by P_2 and P_3: PSD(AB) AND b_{OL} =[100] AND [011] =[000]. In contrast, (AC) and (A) are affected as both patterns share them. Thus, SHARP first selects partial matches in (AB), then in (AC) and (A). Between (AC) and (A), it prioritizes (A) due to higher PSD ([111] vs [011]). For efficiency, SHARP approximately selects partial matches per buffer using the partial order (e.g., selects PM_4 over PM_6 in Example 4) until reaching the overhead bound in Eq. (3).

The state selector is lightweight. In (i) and (ii), the PSD-based index enables $O(1)$ time to locate state buffers and retrieve partial matches. In (iii), the partial order enables $O(n)$ search time in the number of clusters within a single state buffer–significantly small n in practice. Furthermore, this searching overhead can be hidden, since scanning partial matches is inherently the core processing of pattern matching—overlapping the searching at d_k and pattern match evaluation at d_{k+1}. As a result, the state selector only introduces negligible incremental system overhead.

5 EVALUATION

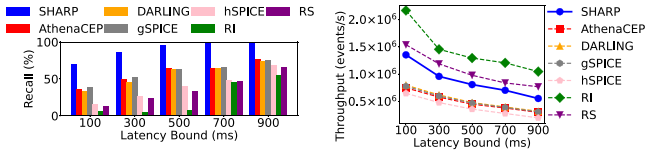
We have evaluated the effectiveness and efficiency of SHARP in various scenarios. After outlining the experimental setup in §5.1, our experimental evaluation answers the following questions:
(1) What are the overall effectiveness and efficiency of SHARP? (§5.2)
(2) How sensitive is SHARP to pattern properties, including pattern selectivity, pattern length, and the time window? (§5.3)
(3) How does SHARP adapt to concept drifts of input data? (§5.4)
(4) How do resource constraints impact SHARP? (§5.5)
(5) How does SHARP adapt to complex pattern interactions? (§5.6)
(6) What is the scalability performance of SHARP? (§5.7)
(7) SHARP' performance compared to SOTA GraphRAG systems.(§5.8)
(8) SHARP's state selection compared to the optimal solution. (§5.9)
We also evaluated SHARP's performance and robustness under different pattern sharing schemes and selection/consumption policies. Due to limited space, we report details in a technical report [76].

5.1 Experimental Setup

Our experiments have the following setup:

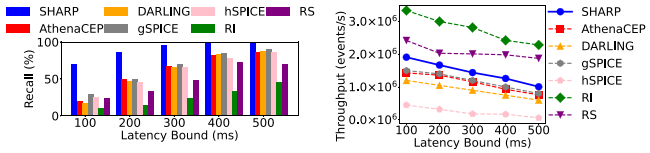
Testbeds. We conduct experiments on two clusters. **(1)** LUMI supercomputer [32]–each node features two AMD EPYC 7763 CPUs (128 cores) and 512 GB RAM, running SUSE Linux Enterprise Server 15 SP5. **(2)** A GPU cluster [2]–each node being equipped with four NVIDIA H100 80GB GPUs, two Intel Xeon Platinum 8468 CPUs (96 cores), and 1.5 TB RAM, running Red Hat Enterprise 9.5.

Baselines. We compare SHARP to **six** baselines: **(1)** Random input (RI) selects input data randomly. **(2)** Random state (RS) selects partial matches randomly. **(3)** DARLING [12] selects input data based on utility and the queue buffer size. **(4)** AthenaCEP [81] selects the combination of input data and partial matches to process based on its



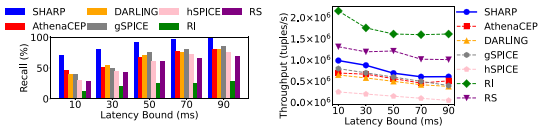
(a) Recall **(b)** Throughput

Fig. 4: The overall performance of shared CEP patterns P_3-P_4 over DS1 at different latency bounds



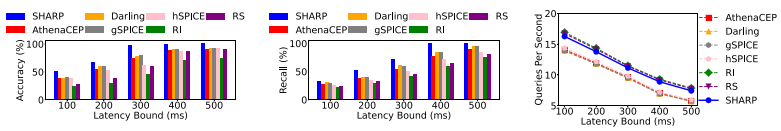
(a) Recall **(b)** Throughput

Fig. 5: The overall performance of shared CEP patterns P_3-P_4 over Citi_Bike [1] at different latency bounds



(a) Recall **(b)** Throughput

Fig. 6: The overall performance of shared MATCH_RECOGNIZE patterns P_5-P_6 over Crimes [46] at various latency bounds



(a) Accuracy **(b)** Recall **(c)** Throughput Improvement

Fig. 7: The overall performance of GraphRAG on the Meta-QA benchmark [79] (P_{39}-$P_{14,910}$) at different latency bounds

Tab. 1: 14,910 patterns evaluated in the experiments

P_1	SEQ(A a, B+ b[], C c, D d) WHERE SAME [ID] AND SUM(b[i].x) < c.x
P_2	SEQ(A a, B+ b[], E e, F f) WHERE SAME [ID] AND a.x + SUM(b[i].x) < e.x + f.x
P_3	SEQ(A a, B b, C c, D d, E e, F f, G g) WHERE SAME [ID] AND a.v < b.v AND b.v + c.v < d.v AND $2r \cdot \arcsin\left(\sin^2\left(\frac{e.x - d.x}{2}\right) + \cos(d.x)\cos(e.x)\sin^2\left(\frac{e.y - d.y}{2}\right)\right)^{\frac{1}{2}} \leqslant f.v$
P_4	SEQ(A a, B b, C c, D d, H h, I i, J j) WHERE SAME [ID] AND a.v < b.v AND b.v + c.v < d.v AND $r \cdot \arccos(\sin(d.x)\sin(h.x) + \cos(d.x)\cos(h.x)\cos(h.y - d.y)) \leqslant i.v$
P_5	SEQ(A a, B b, NEG(C) c, D d) WHERE SAME [ID] AND a.x < b.x
P_6	SEQ(A a, B b, NEG(C) c, E e) WHERE SAME [ID]
$P_7 - P_{38}$	32 shared patterns with common predicate operators of sequence, Kleene closure and negation (due to limited space, please find details in the technical report [76])
$P_{39} - P_{14,910}$	*14,872* queries in the Meta-QA benchmark [79]

cost model. **(5)** gSPICE [64] selects input data using a decision-tree-based black-box model trained on data properties. **(6)** hSPICE [63] selects input data using a probabilistic model based on event type, position, and partial match state.

Datasets. We have evaluated SHARP and baseline approaches using two synthetic datasets and three real-world datasets:

(1) Synthetic Datasets (DS1 and DS2). **(i)** DS1 contains tuples consisting of five uniformly-distributed attributes: a categorical type ($\mathcal{U}(\{A, B, C, D, E, F, G, H, I, J\})$), a numeric *ID* ($\mathcal{U}(1, 10)$), and numeric attributes X ($\mathcal{U}(-90, 90)$), Y ($\mathcal{U}(-180, 180)$), and V ($\mathcal{U}(1, 3 \times 10^6)$). **(ii)** DS2 has similar settings, i.e., a categorical type ($\mathcal{U}(\{A, B, C, D, E, F\})$), a numeric *ID* ($\mathcal{U}(1, 25)$), and one numeric attribute X ($\mathcal{U}(1, 100)$).

(2) Citi_Bike [1] is a publicly available dataset of bike trips in New York City. We use it for CEP patterns.

(3) Crimes [46] is a public crime record dataset from the City of Chicago's Data Portal. We use it for MATCH_RECOGNIZE patterns.

(4) KG-Meta-QA [79] is a knowledge graph that captures structured information of movies. We use it for path query patterns in GraphRAG.

Patterns. We have evaluated 14,910 patterns, as shown in Tab. 1. P_1-P_6 are pattern *templates* evaluated over synthetic and real-world data by materializing the schema (e.g., materializing A in P_3 with bike_trip). They cover three representative pattern-sharing schemes. P_1 and P_2 share a Kleene closure sub-pattern SEQ(A,B$^+$). P_3 and P_4 share the sub-pattern SEQ(A,B,C,D) with computationally expensive predicates. P_5 and P_6 share a negation pattern SEQ(A,B,$\neg$C). P_7-P_{38} access SHARP's scalability by ranging the number of shared patterns from 2 to 32. Due to limited space, we present their details in a technical report [76]. P_{39}-$P_{14,910}$ evaluate the patterns in GraphRAG, taken from the 14,872 patterns in Meta-QA benchmark [79].

Metrics. We measure end-to-end result quality and throughput for pattern matching under strict latency bounds (wall-clock time). We set bounds (§5.2) using application SLOs for real-world datasets and baseline latency (without state reduction) for synthetic datasets. Result quality is measured by *recall*: the ratio of complete matches with state reduction to those without it. We omit accuracy for CEP and MATCH_RECOGNIZE, since complete matches are always exact (100% accuracy). For GraphRAG, we evaluate the full pipeline in Fig. 3 ❶-❻, beyond KG matching. Its recall is the fraction of LLM responses matching ground truth, and accuracy is the fraction of correct answers among all generated responses. For *throughput*, we report *events/tuples per second* for CEP and MATCH_RECOGNIZE, and *LLM queries per second* (QPS) for GraphRAG.

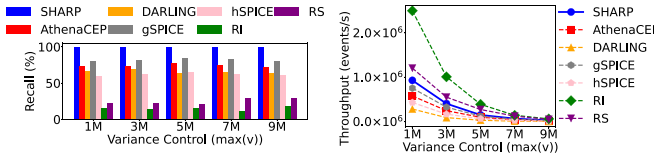
5.2 Overall Effectiveness and Efficiency

We first investigate the overall performance of SHARP in CEP, MATCH_RECOGNIZE and GraphRAG using synthetic and real-world datasets.

We execute the shared CEP patterns, P_3 and P_4, over DS1. Fig. 4 demonstrates the results. The latency without state reduction is 1,035 ms. We set the latency bound ranging from 100 ms to 900 ms. At all latency bounds, SHARP achieves the highest recall value across all baselines (Fig. 4a), achieving over 95% at 500–900 ms. The margin becomes larger at tighter latency bounds. At 100 ms, SHARP achieves 70% recall, 1.96×, 4.10×, 1.81×, 4.3×, 5.30×, and 11.25× higher than AthenaCEP, DARLING, gSPICE, hSPICE, RS and RI.

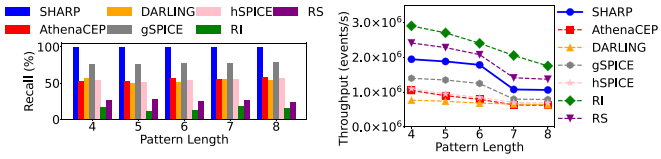
A similar trend is observed in the real-world Citi_Bike dataset, where latency bounds (100–500 ms) align with SLOs in bike-sharing applications. [21, 55]. As shown in Fig. 5a, SHARP outperforms all baselines, improves the recall by 3.5× (Athena CEP), 4.0×(DARLING), 2.4×(gSPICE), 2.8×(hSPICE), 2.8×(RS) and 7×(RI).

We attribute SHARP's high recall to the combination of *pattern-sharing degree* (§4.1) and the cost model (§4.2) which capture both pattern-sharing schemes and the cost of state, i.e., partial matches. AthenaCEP and DARLING do not consider shared patterns, resulting



(a) CEP Recall **(b)** CEP Throughput

Fig. 8: Impact of selectivity on shared patterns P_3-P_6 over DS1



(a) CEP Recall **(b)** CEP Throughput

Fig. 9: Impact of pattern length on shared patterns P_1-P_2 over DS2

in lower recall. Although hSPICE and gSPICE consider multiple patterns, they ignore the interaction and interference among patterns in shared states, which again leads to lower recall.

We then examine the throughput performance (Fig. 4b and Fig. 5b). SHARP's throughput is higher than AthenaCEP (1.8×), DARLING (1.69×), gSPICE (1.73×), and hSPICE (2.1×), but lower than the random approaches. However, the high throughput of RS and RI comes at the expense of poor recall (below 20%). Compared to all baselines, SHARP strikes a better trade-off of recall and throughput.

SHARP's superior performance stems from the PSD design (§4.1), the cost model (§4.2) and efficient hierarchical state selection (§4.3). PSD enables efficient separation of non-latency-violated state and the priority of pattern share degree in the shared patterns, which significantly reduces the search space compared to AthenaCEP, DARLING, gSPICE, and hSPICE. While the greedy selection in latency-violated state buffers further reduces and overlaps SHARP's overhead.

Similar trends are observed in MATCH_RECOGNIZE patterns and the GraphRAG application. Fig. 6 shows the results of executing shared MATCH_RECOGNIZE patterns, P_5 and P_6, over Crimes [46], ranging the latency bounds from 10 ms to 90 ms (crime detection requires a sub-100 ms latency [40, 57]). Here, SHARP outperforms all baselines in recall (Fig. 6a). At the 10 ms bound, SHARP achieves the recall of 74% that is 1.2×, 2.56×, 1.85×, 1.84×, 2.0×, and 3.7× higher than AthenaCEP, DARLING, gSPICE, hSPICE, RS and RI. SHARP's throughput again falls between that of the existing techniques (AthenaCEP, DARLING, gSPICE, and hSPICE) and the random approaches (Fig. 6b).

For GraphRAG, SHARP executes the full pipeline in Fig. 1a on Meta-QA [79]. Across 14,872 LLM queries, the average end-to-end latency is 1,032 ms, with 635 ms spent on KG pattern matching. Fig. 7 reports end-to-end results while varying the KG matching latency bound from 100 ms to 500 ms, following IR SLOs [66]. SHARP achieves the best accuracy and recall: it maintains 100% accuracy at 400 ms/500 ms, and reaches 70% at 200 ms, i.e., 1.27×–2.1× higher than AthenaCEP, DARLING, gSPICE, hSPICE, RS, and RI. For recall, the margin shrinks at tight bounds; at 100 ms, SHARP is comparable to baselines (Fig. 7b), since few LLM responses cannot cover most ground truth. Still, SHARP's selector ensures >80% generated responses align with ground truth. SHARP matches the throughput of RS/RI and is 1.41× higher than other baselines (Fig. 7c), showing low overhead in complex pipelines.

5.3 Sensitivity Analysis of Pattern Properties

Next, we examine SHARP's sensitivity and robustness to various pattern properties, considering selectivity, pattern length, and the time window size, because these properties affect the size of the state that changes in runtime. Due to the limited space, we only report the experimental results of CEP patterns. We provide detailed results of MATCH_RECOGNIZE patterns in the technical report [76].

Selectivity. Pattern predicates select data and partial matches. We control the selectivity by changing the value distribution of V in DS1. In particular, we change the distribution of V from $\mathcal{U}(0, 1\times10^6)$ to $\mathcal{U}(0, 1\times10^9)$, which increases the selectivity for P_3 and P_4.

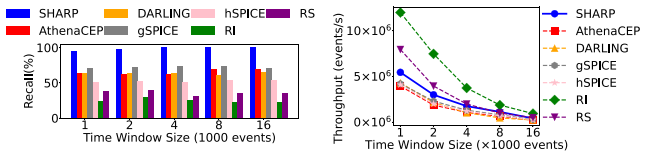
Fig. 8a shows SHARP's robustness. It keeps a stable recall of 100% across all selectivity configurations, outperforming baselines. In contrast, the recall of baselines changes with the selectivity. Specifically, the recall of AthenaCEP and DARLING drops by 10% and 5%. The increased selectivity results in more partial matches, which lowers the throughput for all methods (Fig. 8b). However, SHARP's throughput is consistently higher than all baselines – 14.1% higher than the second-best gSPICE. We attribute SHARP's robustness to its cost model, which efficiently adapts to changes in selectivity.

Pattern length. We control the pattern length of P_1 and P_2 over DS1, ranging from four to eight by changing the length of the Kleene closure, B+. Fig. 9 shows that the recall of SHARP is not affected by pattern length (stable in 100% recall), consistently outperforming baselines. In contrast, the recall fluctuates by 5.4%, 4.7%, 3.2%, 7.8%, 10.1%, and 9.8% for AthenaCEP, DARLING, gSPICE, hSPICE, RI and RS, respectively. The increased pattern length leads to lower throughput due to more generated partial matches. However, SHARP still outperforms all non-random baselines in throughput, i.e., 25.9%, 28.5%, 23.2%, and 25.2% higher than AthenaCEP, DARLING, gSPICE, and hSPICE. These results indicate that SHARP is robust to changes in pattern length, and complex patterns can benefit more from SHARP.

Time window size. We change the size of the sliding time window of P_3 and P_4 over data streams in DS2, ranging from 1k to 16k events. The slide is one event. Fig. 10a shows that SHARP consistently yields the highest recall compared to baselines. SHARP's recall increases with increasing time window, from 95% to 100%. This is because a larger time window provides more historical statistics for the cost model to learn, which allows SHARP to select more promising states. Regarding throughput, a larger time window increases the size of the maintained state, resulting in lower throughput for SHARP and the baselines. However, on average, SHARP's throughput remains 13.2% higher than the best of the non-random baselines.

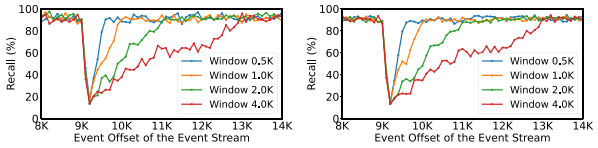
5.4 Adaptivity to Concept Drifts

We investigate SHARP's adaptivity to concept drifts, using P_3 and P_4 over data streams derived from DS1. To control the concept drift, we change the value distribution of $D.V$ from $\mathcal{U}$ $(1\times10^6, 3.5\times10^6)$ to $\mathcal{U}$ $(1, 2\times10^6)$ at the offset of 9k in the event stream. Fig. 11 shows an abrupt drop of recall (to 18%) immediately after the concept drift. This is because SHARP's cost model is no longer accurate due to the flipped value distribution. However, SHARP can swiftly detect the drift and quickly update its cost model to improve the recall. After one time window, SHARP improves the recall back to normal level



(a) Recall **(b)** Throughput

Fig. 10: Impact of time window size on patterns P_3–P_4 over DS1



(a) Recall of P_3 **(b)** Recall of P_4

Fig. 11: Adaptivity to concept drifts in shared patterns P_3–P_4 over DS1

and stabilizes between 95% to 100%. The convergence is slower for larger time windows due to longer lifespan of stale partial matches.

5.5 Impact on Resource Constraints

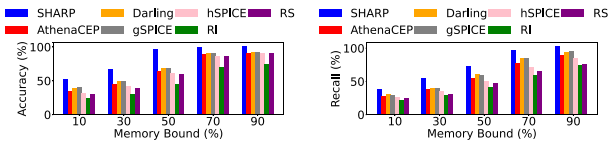
We examine how SHARP handles limited memory capacity using GraphRAG with memory bounds from 90% to 10% of its original memory footprint. We measure the accuracy and recall. As shown in Fig. 12, SHARP maintains the highest accuracy across all memory bounds, achieving 95% accuracy with only 50% memory, that is, 1.52×, 1.31×, 1.28×, 1.60×, 1.33×, 1.83×, and 2.05× higher than AthenaCEP, DARLING, gSPICE, hSPICE, RS, and RI, respectively. Regarding recall, SHARP outperforms baselines in all memory bounds by 1.5× (AthenaCEP), 1.4× (DARLING), 1.3× (gSPICE), 1.6× (hSPICE), 1.8× (RS) and 2.1× (RI). But the recall drops to 40% at 10% memory bound, because it depends on the statistical efficiency of LLM-generated responses–a small set cannot cover the majority of ground truth.

5.6 Adaptivity to Complex Pattern Interactions

We investigate SHARP's adaptivity to complex pattern interactions. We (i) examine how changing one pattern's latency bound affects other shared patterns, and (ii) the impact of varying pattern latency bounds at opposite directions: increase one and decrease the other.

For (i), we fix P_3 's latency bound at 500 ms while ranging P_4's from 100 ms to 500 ms, using Citi_Bike datasets. Fig. 13 illustrates the recall of P_3 and P_4. Here, relaxing P_4's latency bound reduces P_3's recall (Fig. 13a). The reason is two-fold. First, relaxed bounds allow P_4 to consume more computational resources, leaving less for P_3. Second, relaxing bounds on P_4 also increases the number of partial matches for P_3 (shared by P_3 and P_4). SHARP's PSD and cost model captures this, resulting in only 14% drops in recall, compared to AthenaCEP's 19%, DARLING's 20%, hSPICE's 22% and gSPICE's 26%.

For (ii), we decrease P_3's latency bound from 500 ms to 100 ms, while increasing P_4's from 100 ms to 500 ms. Fig. 14 shows decreasing recall in P_3 and increasing recall in P_4. Because P_3's tighter latency bounds consume less computational resources, making room for P_4's increased consumption at relaxed bounds. Again, SHARP's PSD and cost model results in superior performance than AthenaCEP (4.2×), DARLING (6.1×), hSPICE (12.5×), and gSPICE (10.2×).



(a) Accuracy **(b)** Recall

Fig. 12: Impact of memory constraint on GraphRAG over Meta-QA

5.7 Scalability Analysis

We examine SHARP's scalability for shared patterns, ranging the number of shared patterns from 2 to 32. We use P_7 –P_{38} for CEP and MATCH_RECOGNIZE on DS1 and 32 patterns from Meta-QA [79] for GraphRAG. Fig. 15 shows the recall and throughput. SHARP consistently achieves the highest recall and throughput with an increasing number of shared patterns. The throughput is only slightly below the random baselines. However, poor recall of RS and RI makes them unusable. Note that increasing shared patterns significantly increases the number of partial matches by 2 to 16 orders of magnitude, resulting in deteriorated throughput for all approaches. However, SHARP' throughput drops slower than other baselines, presenting stronger scalability, 1.26× better than gSPICE and 1.25×, 1.29×, 1.28×, 1.31× better than hSPICE, AthenaCEP, and DARLING. We attribute SHARP's superior scalability to its PSD-based indexing and the cost model that captures the state sharing schemes.

5.8 Integration/Comparison with Neo4j-GraphRAG

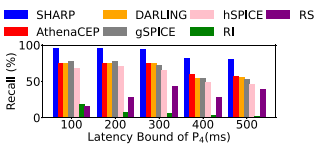
We study how SHARP (as a stand-alone state reduction library) improves a leading industrial GraphRAG system, Neo4j-GraphRAG [42]. We integrate SHARP into Neo4j-GraphRAG, coined as Neo4j-GraphRAG +Sharp (see our technical report [76]), and compare it against four Neo4j-GraphRAG baselines. We build the pipeline with Qwen3-8B [70] and evaluate 14,273 *3-hop* path queries from KG-MetaQA [79].

The baselines are: (i) Neo4j-GraphRAG: default execution without shared state or reduction. (ii) Neo4j-GraphRAG-Shared: shared state without reduction. (iii) Neo4j-GraphRAG-RS: random reduction without shared state. (iv) Neo4j-GraphRAG-Shared-RS: shared state with random reduction.

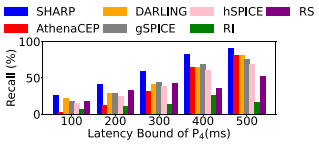
Fig. 16 reports accuracy, recall, and throughput. Without state reduction, Neo4j-GraphRAG reaches 100% accuracy and recall but incurs 1,429 ms latency. Sharing state reduces latency to 242 ms in Neo4j-GraphRAG-Shared. We also evaluate state reduction under latency bounds 10-100 ms. Neo4j-GraphRAG+Sharp achieves the best accuracy, recall, and throughput. At 100 ms, it reaches 91.2% accuracy and 90.8% recall: 1.84×, 1.63× above Neo4j-GraphRAG-Shared-RS, and 4.21×, 5.30× above Neo4j-GraphRAG-RS. The largest gap is at 40 ms: Neo4j-GraphRAG+Sharp achieves 84.9% accuracy and 87.3% recall, versus 28.7%/37.3% for Neo4j-GraphRAG-Shared-RS and 12.3%/10.8% for Neo4j-GraphRAG-RS. Overall, SHARP helps Neo4j-GraphRAG meet tighter latency bounds with minimal quality loss.

5.9 Optimality of SHARP's State Selection

We investigate the performance of SHARP's cost model in state selection compared to the optimal dynamic programming (DP) oracle. To this end, we compare SHARP's state selection to a leading industrial DP solver, Google OR-Tools (CP-SAT) [50], when selecting 10

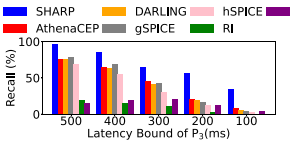


(a) Recall of P_3 under fixed bound

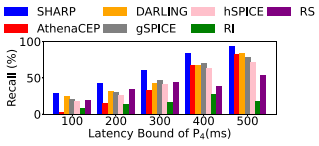


(b) Recall of P_4 under varying bounds

Fig. 13: Impact of pattern interactions as P_4's latency bound varies from 100-500 ms, with P_3's bound fixed at 500 ms.

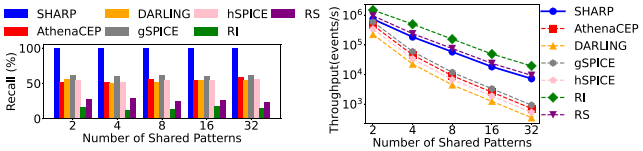


(a) Recall of P_3 under varying bounds



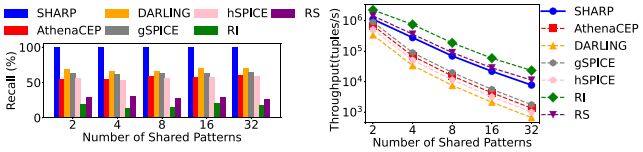
(b) Recall of P_4 under varying bounds

Fig. 14: Impact of pattern interactions as P_4's latency bound increases from 100-500 ms and P_3's decreases from 500-100 ms.



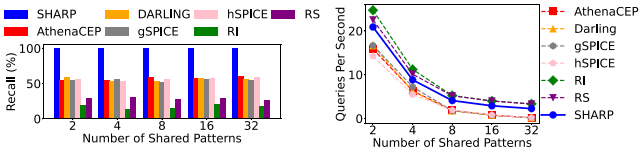
(a) CEP Recall

(b) CEP Throughput (log scale)



(c) MATCH_RECOGNIZE Recall

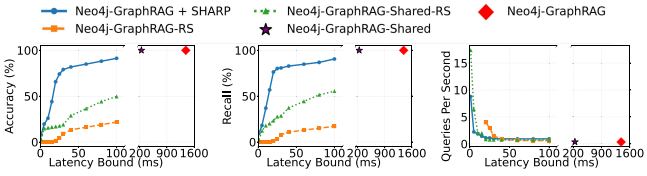
(d) MATCH_RECOGNIZE Throughput (log scale)



(e) GraphRAG Recall

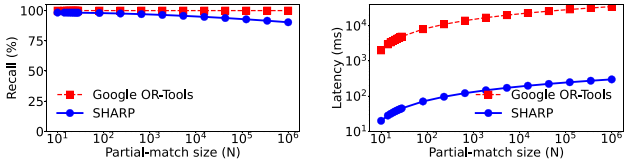
(f) GraphRAG Throughput

Fig. 15: Scalability of SHARP with shared CEP and MATCH_RECOGNIZE patterns P_7–P_{38} and GraphRAG patterns from Meta-QA [79]



(a) Accuracy **(b)** Recall **(c)** Throughput

Fig. 16: SHARP-enhanced Neo4j GraphRAG compared to baselines



(a) Recall changing with PM size **(b)** Latency changing with PM size

Fig. 17: SHARP's state selection compared to dynamic programming

to 10^6 PMs. Fig. 17a and Fig. 17b show the recall and latency. At 10^2 PMs, SHARP achieves 99.3% recall and 70.9 ms latency. In contrast, CP-SAT keeps 100% recall but with 3.13×10^3 ms latency (44.1× slower than SHARP). When selecting 10^6 PMs, SHARP maintains 90.3% recall and 313 ms latency. CP-SAT still keeps 100% recall but the latency becomes unacceptably high, 3.45×10^4 ms (110× slower).

The results show that SHARP's state selection achieves comparable quality with the theoretically optimal solution at much lower computational overhead (at least two orders of magnitude faster). In contrast, the theoretically optimal solution is impractical for low-latency constraints due to its high computational complexity.

6 RELATED WORK

Multi-pattern sharing optimizations [26, 35, 51–54, 78] reuses shared sub-patterns to reduce memory. SPASS [54] estimates sharing benefits from intra-/inter-query correlations, while Sharon [53] and HAMLET [51] support aggregation. MCEP [26], GRETA [52], and GLORIA [35] enable sharing in Kleene closure. These approaches complement SHARP by optimizing utilization; SHARP targets best-effort processing under latency bounds and integrates them.

Load shedding discards data elements or state [12, 16, 22, 61–63, 69, 81]. Input-based methods [12, 62, 63] drop inputs by estimated result importance, while state-based schemes [61, 80] discard partial matches via probabilistic utilities. Hybrid shedding [81] combines both with a cost model. Unlike SHARP, these schemes ignore interactions across patterns via shared state.

GraphRAG [49] improves RAG by retrieving relationships from knowledge graphs [17, 18, 58, 72]. Neo4j [44] and Memgraph [37] speed up single-query execution via planning and caching, but lack state sharing across path queries and best-effort processing under strict latency bounds. These gaps motivate SHARP state reduction for GraphRAG systems.

7 CONCLUSIONS

We presented SHARP, a state management library for best-effort shared pattern matching. It aims to satisfy strict latency bounds while maximizing result quality. SHARP introduces *pattern-sharing degree* (PSD) to capture sharing schemes and cross-pattern interactions, and a *cost model* to quantify shared-state importance and overhead. Its *state selector* chooses state in constant time via PSD indexing and partially ordered cost models. We comprehensively evaluated SHARP on real-world CEP, OLAP, and GraphRAG workloads against several SOTA baselines.

8 ACKNOWLEDGMENT

This work is funded by Research Council of Finland (grant number 362729) and Business Finland (grant number 169/31/2024). We also acknowledge the computational resources provided by the Aalto Science-IT project and LUMI supercomputer owned by the EuroHPC Joint Undertaking, hosted by CSC Finland.

REFERENCES

[1] 2024. Citi Bike. http://www.citibikenyc.com/system-data. Accessed: July 16, 2024.

[2] Aalto University, SciComp. [n. d.]. Triton HPC. https://scicomp.aalto.fi/triton/. Accessed: July 31, 2025.

[3] Serge Abiteboul and Victor Vianu. 1997. Regular path queries with constraints. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 122–133.

[4] Zahid Abul-Basher. 2017. Multiple-query optimization of regular path queries. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 1426–1430.

[5] Zahid Abul-Basher, Nikolay Yakovets, Parke Godfrey, and Mark H Chignell. 2016. SwarmGuide: Towards Multiple-Query Optimization in Graph Databases.. In *AMW*.

[6] Samira Akili, Steven Purtzel, and Matthias Weidlich. 2024. DecoPa: Query Decomposition for Parallel Complex Event Processing. *Proc. ACM Manag. Data* 2, 3, Article 132 (May 2024), 26 pages. doi:10.1145/3654935

[7] Rodrigo Alves. 2019. *Azure Stream Analytics now supports MATCH_RECOGNIZE.* https://azure.microsoft.com/en-us/blog/azure-stream-analytics-now-supports-match-recognize/

[8] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)* 50, 5 (2017), 1–40.

[9] AWS Machine Learning Blog. 2024. Improving retrieval-augmented generation accuracy with GraphRAG. https://aws.amazon.com/blogs/machine-learning/improving-retrieval-augmented-generation-accuracy-with-graphrag/

[10] Neo4j Developer Blog. 2024. Knowledge Graph RAG Application. https://neo4j.com/developer-blog/knowledge-graph-rag-application/

[11] Sharma Chakravarthy and Deepak Mishra. 1994. Snoop: An expressive event specification language for active databases. *Data & Knowledge Engineering* 14, 1 (1994), 1–26.

[12] Koral Chapnik, Ilya Kolchinsky, and Assaf Schuster. 2021. DARLING: data-aware load shedding in complex event processing systems. *Proceedings of the VLDB Endowment* 15, 3 (2021), 541–554.

[13] Gianpaolo Cugola and Alessandro Margara. 2010. TESLA: a formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. 50–61.

[14] Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)* 44, 3 (2012), 1–62.

[15] Carlos Gomes Da Silva, João Clímaco, and José Rui Figueira. 2008. Core problems in bi-criteria {0, 1}-knapsack problems. *Computers & Operations Research* 35, 7 (2008), 2292–2306.

[16] Nihal Dindar, Peter M Fischer, Merve Soner, and Nesime Tatbul. 2011. Efficiently correlating complex events over live and archived data streams. In *Proceedings of the 5th ACM international conference on Distributed event-based system*. 243–254.

[17] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. 2025. From Local to Global: A Graph RAG Approach to Query-Focused Summarization. arXiv:2404.16130 [cs.CL] https://arxiv.org/abs/2404.16130

[18] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. 2025. From Local to Global: A Graph RAG Approach to Query-Focused Summarization. arXiv:2404.16130 [cs.CL] https://arxiv.org/abs/2404.16130

[19] Kasia Findeisen. 2021. *Row pattern recognition with MATCH_RECOGNIZE.* https://trino.io/blog/2021/05/19/row_pattern_matching.html

[20] Lars George, Bruno Cadonna, and Matthias Weidlich. 2016. IL-Miner: Instance-Level Discovery of Complex Event Patterns. *Proc. VLDB Endow.* 10, 1 (2016), 25–36. doi:10.14778/3015270.3015273

[21] Saif Gunja. 2023. SLO examples for faster, more reliable apps. https://www.dynatrace.com/news/blog/service-level-objective-examples-5-slo-examples/.

[22] Yeye He, Siddharth Barman, and Jeffrey F. Naughton. 2014. On Load Shedding in Complex Event Processing. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy (Eds.). OpenProceedings.org, 213–224. doi:10.5441/002/ICDT.2014.23

[23] Silu Huang, Erkang Zhu, Surajit Chaudhuri, and Leonhard Spiegelberg. 2023. T-rex: Optimizing pattern search on time series. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.

[24] ISO/IEC JTC 1/SC 32 Data management and interchange. 2016. *ISO/IEC TR 19075-5:2016 Information technology – Database languages – SQL Technical Reports – Part 5: Row Pattern Recognition in SQL.* Technical Report. International Organization for Standardization (ISO). https://www.iso.org/standard/65143.html

[25] Jiho Kim, Yeonsu Kwon, Yohan Jo, and Edward Choi. 2023. KG-GPT: A General Framework for Reasoning on Knowledge Graphs Using Large Language Models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 9410–9421. doi:10.18653/v1/2023.findings-emnlp.631

[26] Ilya Kolchinsky and Assaf Schuster. 2019. Real-time multi-pattern detection over event streams. In *Proceedings of the 2019 International Conference on Management of Data*. 589–606.

[27] Michael Körber, Nikolaus Glombiewski, and Bernhard Seeger. 2021. Index-accelerated pattern matching in event stores. In *Proceedings of the 2021 International Conference on Management of Data*. 1023–1036.

[28] Keith Laker. 2017. *MATCH_RECOGNIZE and predicates - everything you need to know.* https://blogs.oracle.com/datawarehousing/post/match_recognize-and-predicates-everything-you-need-to-know

[29] Zheng Li and Tingjian Ge. 2016. History is a mirror to the future: Best-effort approximate complex event matching with insufficient resources. *Proc. VLDB Endow.* 10, 4 (2016), 397–408. doi:10.14778/3025111.3025121

[30] Leonid Libkin and Domagoj Vrgoč. 2012. Regular path queries on graphs with data. In *Proceedings of the 15th International Conference on Database Theory*. 74–85.

[31] Xunyun Liu and Rajkumar Buyya. 2020. Resource management and scheduling in distributed stream processing systems: a taxonomy, review, and future directions. *ACM Computing Surveys (CSUR)* 53, 3 (2020), 1–41.

[32] LUMI Consortium. [n. d.]. LUMI Supercomputer. https://lumi-supercomputer.eu/. Accessed: July 31, 2025.

[33] Linhao Luo, Yuan-Fang Li, Gholamreza Haffari, and Shirui Pan. 2024. Reasoning on Graphs: Faithful and Interpretable Large Language Model Reasoning. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=ZGNWW7xZ6Q

[34] Thibaut Lust and Jacques Teghem. 2012. The multiobjective multidimensional knapsack problem: a survey and a new approach. *International Transactions in Operational Research* 19, 4 (2012), 495–520.

[35] Lei Ma, Chuan Lei, Olga Poppe, and Elke A Rundensteiner. 2022. Gloria: Graph-based Sharing Optimizer for Event Trend Aggregation. In *Proceedings of the 2022 International Conference on Management of Data*. 1122–1135.

[36] Yuan Mei and Samuel Madden. 2009. Zstream: a cost-based query processor for adaptively detecting composite events. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 193–206.

[37] Memgraph Ltd. 2025. *Memgraph: In-Memory Graph Database.* Zagreb, Croatia. https://memgraph.com Open-source, ACID-compliant, in-memory graph database written in C/C++:contentReference[oaicite:5]index=5:contentReference[oaicite:6]index=6.

[38] Alberto O Mendelzon and Peter T Wood. 1995. Finding regular simple paths in graph databases. *SIAM J. Comput.* 24, 6 (1995), 1235–1258.

[39] Microsoft. 2024. GraphRAG - Microsoft Research. https://microsoft.github.io/graphrag/

[40] Manas Ranjan Mishra and Pramod Kumar Meher. 2024. Abnormal Human Behaviour Detection by Surveillance Camera. In *2024 Asia Pacific Conference on Innovation in Technology (APCIT)*. IEEE, 1–6.

[41] Yuya Nasu, Hiroyuki Kitagawa, and Kosuke Nakabasami. 2019. Efficient Row Pattern Matching Using Pattern Hierarchies for Sequence OLAP. In *Big Data Analytics and Knowledge Discovery: 21st International Conference, DaWaK 2019, Linz, Austria, August 26–29, 2019, Proceedings* (Linz, Austria). Springer-Verlag, Berlin, Heidelberg, 89–104. doi:10.1007/978-3-030-27520-4_7

[42] Neo4j. [n. d.]. neo4j/neo4j-graphrag-python: Neo4j GraphRAG for Python. https://github.com/neo4j/neo4j-graphrag-python. Accessed: October 20, 2025.

[43] Neo4j. 2024. Global Automaker. https://neo4j.com/customer-stories/global-automaker/

[44] Neo4j Inc. 2025. *Neo4j Graph Database.* San Mateo, CA. https://neo4j.com ACID-compliant, disk-based native graph database:contentReference[oaicite:4]index=4.

[45] Thi Nguyen, Linhao Luo, Fatemeh Shiri, Dinh Phung, Yuan-Fang Li, Thuy-Trang Vu, and Gholamreza Haffari. 2024. Direct Evaluation of Chain-of-Thought in Multi-hop Reasoning with Knowledge Graphs. In *Findings of the Association for Computational Linguistics: ACL 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 2862–2883. doi:10.18653/v1/2024.findings-acl.168

[46] City of Chicago. 2024. Crimes - 2001 to Present. https://data.cityofchicago.org/Public-Safety/Crimes-2001-to-Present/ijzp-q8t2. Accessed: October 16, 2024.

[47] Anil Pacaci, Angela Bonifati, and M Tamer Özsu. 2020. Regular path query evaluation on streaming graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1415–1430.

[48] Marta Paes. 2019. *MATCH_RECOGNIZE: where Flink SQL and Complex Event Processing meet.* https://www.ververica.com/blog/match_recognize-where-flink-sql-and-complex-event-processing-meet

[49] Boci Peng, Yun Zhu, Yongchao Liu, Xiaohe Bo, Haizhou Shi, Chuntao Hong, Yan Zhang, and Siliang Tang. 2024. Graph Retrieval-Augmented Generation: A Survey. arXiv:2408.08921 [cs.AI] https://arxiv.org/abs/2408.08921

[50] Laurent Perron and Frédéric Didier. 2025. *CP-SAT.* Google. https://developers.google.com/optimization/cp/cp_solver/

[51] Olga Poppe, Chuan Lei, Lei Ma, Allison Rozet, and Elke A Rundensteiner. 2021. To share, or not to share online event trend aggregation over bursty event streams. In *Proceedings of the 2021 International Conference on Management of*

Data. 1452–1464.

[52] Olga Poppe, Chuan Lei, Elke A. Rundensteiner, and David Maier. 2017. GRETA: graph-based real-time event trend aggregation. *Proc. VLDB Endow.* 11, 1 (Sept. 2017), 80–92. doi:10.14778/3151113.3151120

[53] Olga Poppe, Allison Rozet, Chuan Lei, Elke A Rundensteiner, and David Maier. 2018. Sharon: Shared online event sequence aggregation. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 737–748.

[54] Medhabi Ray, Chuan Lei, and Elke A Rundensteiner. 2016. Scalable pattern sharing on event streams. In *Proceedings of the 2016 international conference on management of data*. 495–510.

[55] Pragmatics RE. 2024. SLO Examples. https://www.pragmaticsre.com/psre-guide/conclusion/slo-examples. Accessed: July 10, 2024.

[56] Microsoft Research. 2024. GraphRAG: Unlocking LLM Discovery on Narrative Private Data. https://www.microsoft.com/en-us/research/blog/graphrag-unlocking-llm-discovery-on-narrative-private-data/

[57] Poojashree Chandrashekar Pankaj M Sajjanar. 2025. Real-Time, Low-Latency Surveillance Using Entropy-Based Adaptive Buffering and MobileNetV2 on Edge Devices. *arXiv preprint arXiv:2506.14833* (2025).

[58] Bhaskarjit Sarmah, Benika Hall, Rohan Rao, Sunil Patel, Stefano Pasquali, and Dhagash Mehta. 2024. HybridRAG: Integrating Knowledge Graphs and Vector Retrieval Augmented Generation for Efficient Information Extraction. arXiv:2408.04948 [cs.CL] https://arxiv.org/abs/2408.04948

[59] Rebecca Sattler, Sarah Kleest-Meißner, Steven Lange, Markus L. Schmid, Nicole Schweikardt, and Matthias Weidlich. 2025. DISCES: Systematic Discovery of Event Stream Queries. *Proc. ACM Manag. Data* 3, 1, Article 32 (Feb. 2025), 26 pages. doi:10.1145/3709682

[60] Siemens. 2024. Artificial Intelligence: Industrial Knowledge Graph. https://www.siemens.com/global/en/company/stories/research-technologies/artificial-intelligence/artificial-intelligence-industrial-knowledge-graph.html

[61] Ahmad Slo, Sukanya Bhowmik, Albert Flaig, and Kurt Rothermel. 2019. pspice: Partial match shedding for complex event processing. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 372–382.

[62] Ahmad Slo, Sukanya Bhowmik, and Kurt Rothermel. 2019. eSPICE: Probabilistic Load Shedding from Input Event Streams in Complex Event Processing. In *Proceedings of the 20th International Middleware Conference* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 215–227. doi:10.1145/3361525.3361548

[63] Ahmad Slo, Sukanya Bhowmik, and Kurt Rothermel. 2020. hSPICE: state-aware event shedding in complex event processing. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems* (Montreal, Quebec, Canada) *(DEBS '20)*. Association for Computing Machinery, New York, NY, USA, 109–120. doi:10.1145/3401025.3401742

[64] Ahmad Slo, Sukanya Bhowmik, and Kurt Rothermel. 2023. gspice: Model-based event shedding in complex event processing. In *2023 IEEE International Conference on Big Data (BigData)*. IEEE, 263–270.

[65] Snowflake. 2021. *Identifying Sequences of Rows That Match a Pattern*. https://docs.snowflake.com/en/user-guide/match-recognize-introduction.html

[66] Google SRE. [n. d.]. Implementing SLOs. https://sre.google/workbook/implementing-slos/.

[67] Jiashuo Sun, Chengjin Xu, Lumingyuan Tang, Saizhuo Wang, Chen Lin, Yeyun Gong, Lionel M. Ni, Heung-Yeung Shum, and Jian Guo. 2024. Think-on-Graph: Deep and Responsible Reasoning of Large Language Model on Knowledge Graph. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=nnVO1PvbTv

[68] Xingyu Tan, Xiaoyang Wang, Qing Liu, Xiwei Xu, Xin Yuan, and Wenjie Zhang. 2025. Paths-over-Graph: Knowledge Graph Empowered Large Language Model Reasoning. arXiv:2410.14211 [cs.CL]

[69] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. 2003. Load shedding in a data stream manager. In *Proceedings 2003 vldb conference*. Elsevier, 309–320.

[70] Qwen Team. 2025. Qwen3-8B. https://huggingface.co/Qwen/Qwen3-8B. Hugging Face repository.

[71] Sarisht Wadhwa, Anagh Prasad, Sayan Ranu, Amitabha Bagchi, and Srikanta Bedathur. 2019. Efficiently Answering Regular Simple Path Queries on Large Labeled Networks. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1463–1480. doi:10.1145/3299869.3319882

[72] Xintao Wang, Qianwen Yang, Yongting Qiu, Jiaqing Liang, Qianyu He, Zhouhong Gu, Yanghua Xiao, and Wei Wang. 2023. KnowledGPT: Enhancing Large Language Models with Retrieval and Storage Access on Knowledge Bases. arXiv:2308.11761 [cs.CL] https://arxiv.org/abs/2308.11761

[73] J.W.J. Williams. 2025. Algorithm 232: Heapsort. *Commun. ACM* 7, 6 (May 2025), 347–348. doi:10.1145/512274.3734138

[74] Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 407–418.

[75] Wen-tau Yih, Matthew Richardson, Chris Meek, Ming-Wei Chang, and Jina Suh. 2016. The Value of Semantic Parse Labeling for Knowledge Base Question Answering. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, Katrin Erk and Noah A. Smith (Eds.). Association for Computational Linguistics, Berlin, Germany, 201–206. doi:10.18653/v1/P16-2033

[76] Cong Yu, Tuo Shi, Matthias Weidlich, and Bo Zhao. 2025. SHARP: Shared State Reduction for Efficient Matching of Sequential Patterns. *arXiv preprint arXiv:2507.04872* (2025). https://arxiv.org/abs/2507.04872

[77] Haopeng Zhang, Yanlei Diao, and Neil Immerman. 2014. On complexity and optimization of expensive queries in complex event processing. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 217–228.

[78] Shuhao Zhang, Hoang Tam Vo, Daniel Dahlmeier, and Bingsheng He. 2017. Multi-query optimization for complex event processing in SAP ESP. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 1213–1224.

[79] Yuyu Zhang, Hanjun Dai, Zornitsa Kozareva, Alexander Smola, and Le Song. 2018. Variational Reasoning for Question Answering With Knowledge Graph. *Proceedings of the AAAI Conference on Artificial Intelligence* 32, 1 (Apr. 2018). doi:10.1609/aaai.v32i1.12057

[80] Bo Zhao. 2018. Complex Event Processing under Constrained Resources by State-Based Load Shedding. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 1699–1703. doi:10.1109/ICDE.2018.00218

[81] Bo Zhao, Nguyen Quoc Viet Hung, and Matthias Weidlich. 2020. Load shedding for complex event processing: Input-based and state-based techniques. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1093–1104.

[82] Erkang Zhu, Silu Huang, and Surajit Chaudhuri. 2023. High-performance row pattern recognition using joins. *Proceedings of the VLDB Endowment* 16, 5 (2023), 1181–1195.

[83] Detlef Zimmer and Rainer Unland. 1999. On the semantics of complex events in active database management systems. In *Proceedings 15th International Conference on Data Engineering (Cat. No. 99CB36337)*. IEEE, 392–399.