When Complex Event Recognition Meets Cloud-Native Architectures

Shizhe Liu[1], Haipeng Dai[1], Meng Li[1], Yuemeng Zhang[1], Shaoxu Song[2],
Zhifeng Bao[3], Hancheng Wang[1], Xiaofeng Gao[4], Guihai Chen[1]
[1]Nanjing University, [2]Tsinghua University, [3]The University of Queensland, [4]Shanghai Jiao Tong University
{shizheliu, yuemengzhang, hanchengwang}@smail.nju.edu.cn, {haipengdai, meng}@nju.edu.cn,
sxsong@tsinghua.edu.cn, baozhifeng.cs@gmail.com, gao-xf@cs.sjtu.edu.cn, gchen1960@163.com

***Abstract*—Complex Event Recognition (CER) aims to detect a predefined pattern composed of multiple primitive events. With the growing adoption of cloud-native techniques (*i.e.*, computing and storage separation), which offer elasticity, availability, and cost efficiency, many database vendors are migrating their products to such architectures. However, when CER operates in cloud-native architectures, network becomes a performance bottleneck. To mitigate network-induced performance degradation, our key insight is to identify shorter time intervals that contain matches and transmit only the events within those intervals, hence reducing the transfer of irrelevant events. Upon this insight, we first propose a dual-filtering strategy that leverages both temporal and predicate constraints under multiple round-trips to incrementally shrink the time intervals. Then, we design two specialized filters: the shrinking window filter which reduces the complexity of time intervals maintenance from $O(N \log N)$ to $O(N)$, and the window-wise join filter, which enables low-cost round-trips for processing equality conditions. Furthermore, we propose a cost model to eliminate detrimental round-trips and prevent inefficiencies caused by excessively fine-grained round-trips. To the best of our knowledge, this is the first study to investigate CER in cloud-native architectures. Extensive evaluations demonstrate that our approach reduces transmission cost by over 70% and achieves a 1.2× to 25× end-to-end query speedup across various evaluation engines (*e.g.*, Flink and Esper) on the real-world and synthetic datasets, compared with the state-of-the-art approaches.**

I. Introduction

Complex event recognition (CER) aims to detect a predefined pattern composed of multiple primitive events. Here, the pattern commonly specifies the constraints on event attribute values, the event temporal order, and the maximum distance between the earliest and latest events [1], [2]. While most prior work [1]–[7] focuses on online CER for real-time monitoring of event streams, offline CER has recently gained increasing importance in analytical workloads that require complex pattern discovery over large volumes of historical data. Typical offline CER applications include cluster failure diagnosis [5], [8], network forensics [9], [10], and financial auditing [4], [11]. Unlike online CER, which operates incrementally with limited state, offline CER must scan and filter massive historical event logs efficiently, making it challenging to scale under modern cloud-native architectures due to heavy I/O and network costs. To support CER in data analytics, the SQL:2016 standard introduced the `MATCH_RECOGNIZE` clause [12], now widely adopted by systems such as Oracle, Trino, Snowflake, and Flink [13]–[16]. This work focuses exclusively on accelerating offline CER over historical datasets.

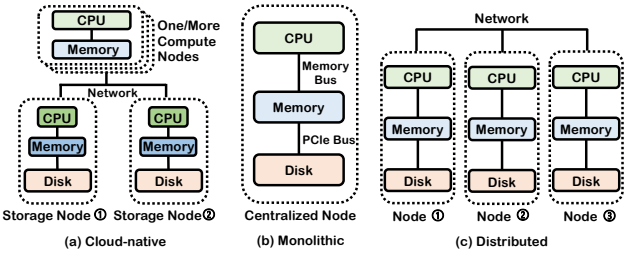


Fig. 1: Three mainstream architectures for CER, where cloud-native architectures have greater elasticity and economy.

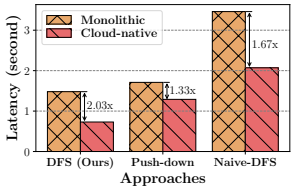


Fig. 2: Comparison of latency under two architectures.

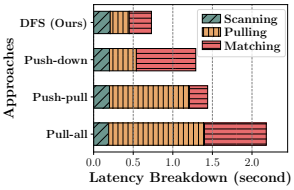


Fig. 3: Latency breakdown in cloud-native architectures.

Cloud-native architectures (as shown in Fig. 1a) separate computing and storage into distinct server layers connected via a network, enabling independent scaling [17]. Compared with traditional monolithic (as shown in Fig. 1b) and distributed architectures (as shown in Fig. 1c), cloud-native architectures are particularly suitable for offline CER workloads for three reasons: ① *Elasticity*: It can easily scale up or down the number of compute nodes (CNs) and storage nodes (SNs) to process dynamic query workloads [18]. ② *High resource utilization*: It allows users to freely select suitable hardware configurations (*e.g.*, memory size, SSD or HDD) for each workload, thereby improving resource utilization [19]. ③ *High parallel loading*: Each SN can read events from the disk and filter them in parallel, reducing the latency of data loading [20]. For example, Fig. 2 shows the average query latency for 150 queries on CRIMES dataset [21] under monolithic and cloud-native architectures (the latter contains one CN and three SNs, which follows the same setting as prior work [22]). Since parallel loading, the latter achieves a query speedup of 1.33× to 2.03× that of monolithic architectures.

Cloud-native architectures face two key challenges for offline CER: events are often out-of-order across SNs due to load balancing, and network bandwidth between CN and SNs is limited, creating a major performance bottleneck. Specifically, events are distributed across SNs in batches, so intervals

2375-026X/26/$31.00 ©2026 IEEE
DOI 10.1109/ICDE65706.2026.00172

TABLE I: Partial events from CRIMES dataset [21].

	primary_type	id	district	date
e_1	ROBBERY	13664891	10	18:45
e_2	BATTERY	13666324	19	18:50
e_3	MOTOR VEHICLE THEFT	13665019	14	18:50
e_4	ASSAULT	13665023	6	18:53
e_5	MOTOR VEHICLE THEFT	13664970	19	19:00
e_6	ROBBERY	13664994	15	20:15
e_7	BATTERY	13664984	7	20:43
e_8	ROBBERY	13665022	24	21:43
e_9	BATTERY	13665004	14	21:50
e_{10}	MOTOR VEHICLE THEFT	13665245	8	22:00

spanning multiple SNs cannot be filtered without collecting metadata from all nodes. Meanwhile, network communication is much slower than local disk access or PCIe transfers [17], [22]–[25]. For example, Fig. 3 shows that pulling all events from SNs over the network ("Pull-all") takes about 1.2s, while scanning the events from disk takes only 0.2s. When network latency outweighs the benefit of parallel loading, the advantages of parallel loading are largely lost.

Without considering caching on CN, several solutions for reducing network overhead include *one-round predicate push-down* [24]–[27] (push-down for short, note that it is not specifically proposed for CER, but rather a universal solution) and *multiple-round predicate push-pull* (push-pull) [2], [28], [29], which offload partial computation to SNs to alleviate network transmission burdens. Unfortunately, (1) push-down still transmits numerous irrelevant events. The reason is that it only pushes independent conditions to SNs for filtering, overlooking the window condition and dependent conditions inherent in the complex event query. (2) Push-pull maintains partial matches and runs costly matching algorithms to filter irrelevant events, which consumes significant CPU and RAM resources (note that the complexity of matching may reach exponential level [5]). As a result, both approaches cannot simultaneously achieve low network overhead and low filtering overhead when processing CER in cloud-native architectures.

Example. Let us consider an example to understand the above two approaches. Table I gives 10 events from CRIMES dataset, where 2024/11/15 is omitted in the date column, each representing a crime report. Suppose 10 events are stored in two SNs, and a user wants to identify three types of criminal activities that may be linked to the same individual within 30 minutes. Listing 1 gives the target query.

Then, for push-down, CN sends the independent conditions (*i.e.*, the constraints in `primary_type`) to each SN. Then, each SN retrieves events from the disk, filters out the event e_4, and sends the remaining 9 events to CN.

For push-pull, CN pulls the events of variables {`R`, `B`, `M`} one by one. Suppose the pulled order is `R`→`B`→`M`. Then, for `R`, SNs send events $\{e_1, e_6, e_8\}$ to CN. For `B`, SNs run a matching algorithm to identify that e_2, e_7, and e_9 can form partial matches $\{(e_1, e_2), (e_6, e_7), (e_8, e_9)\}$ with `R`'s events. SNs then cache the partial matches and send $\{e_2, e_7, e_9\}$ to CN. Similarly, for `M`, SNs identify that only e_5 can form a full match (e_1, e_2, e_5) with the previous partial matches under temporal and `district` attribute constraints, and then send e_5 to CN. As a result, for push-pull, CN receives 7 events.

Listing 1: Complex event query Q_1.

```
SELECT * FROM CRIMES MATCH_RECOGNIZE(
    ORDER BY date
    ALL ROWS PER MATCH
    AFTER MATCH SKIP TO NEXT ROW
    PATTERN (R {-N*-} B {-N*-} M) WITHIN '30'
        MINUTE
    DEFINE
        R AS R.primary_type = 'ROBBERY',
        B AS B.primary_type = 'BATTERY',
        M AS M.primary_type = 'MOTOR VEHICLE
            THEFT' AND M.district = B.district)
```

Our key observation. Actually, the query Q_1 generates only one match, *i.e.*, (e_1, e_2, e_5) (technique details of matching can refer to [16]), which involves three events. The shortest time interval set containing (e_1, e_2, e_5) is {[18:45, 19:00]}, which is a small fraction of the total period [18:45, 22:00] covered by the entire event set. Thus, an intuitive idea is *to identify all time intervals that contain matches and only transmit events with timestamps within these intervals.*

Challenges. ① Identifying the shortest intervals containing matches is non-trivial and typically requires running costly matching algorithms, which increases overall query latency. ② Relying on a single round-trip to identify these intervals still inevitably collects event information unrelated to matching. ③ Conversely, using multiple round-trips can lead to frequent and costly communications. This issue is especially pronounced when dealing with out-of-order event storage, processing equality conditions, or variables with high selectivity, making round-trip costs significantly expensive.

To the best of our knowledge, this is the first study to investigate CER in cloud-native architectures, and our key contributions are summarized as follows:

- We propose a lightweight dual-filtering strategy to identify time intervals that may contain matches without running costly matching algorithm. This approach leverages both temporal and predicate constraints under multiple round-trips to incrementally shrink the intervals.
- We propose two specialized filters: the shrinking window filter (SWF) which reduces the complexity of time intervals maintenance from $O(N \log N)$ to $O(N)$, and the window-wise join filter (WJF), which enables low-cost round-trips for processing equality predicate conditions.
- We provide a cost model to evaluate the benefits of a round-trip and eliminate the detrimental round-trips accordingly. Besides, we provide a theoretical analysis of the false positive rates and space consumption for SWF and WJF.
- We conduct extensive experiments validating the effectiveness of our approach. Specifically, our approach reduces transmission cost by over 70% and achieves a 1.25× to 25× end-to-end query speedup compared with the state-of-the-art approaches.

II. Preliminaries

A. Complex Event Query

In 2016, the `MATCH_RECOGNIZE` clause was introduced in SQL [12] to provide support for extracting complex event patterns. Its syntax is as follows:

```
SELECT <select list>
FROM <source table>
MATCH_RECOGNIZE(
  [ PARTITION BY <partition list> ]
  ORDER BY <order by list>
  [ MEASURES <measure list> ]
  [ONE ROW | ALL ROWS] PER MATCH
  [ AFTER MATCH SKIP <option>]
  PATTERN (<row pattern>) [ WITHIN <window>]
  DEFINE <condition definition list>
) AS <table alias>;
```

where PARTITION BY defines logical partitioning (similar to GROUP BY); ORDER BY specifies row ordering for pattern matching; MEASURES defines output columns; ONE ROW PER MATCH defines the output mode, *i.e.*, produces a single summary row for each match of the pattern, while ALL ROWS PER MATCH produces one row for each row of each match [30]; AFTER MATCH SKIP determines the next match starting position after finding a full match; PATTERN defines the searched pattern in a regular expression-like syntax; WITHIN[1] specifies the distance between the first and the last row in the pattern; DEFINE defines the predicate conditions that pattern variables must hold.

The pattern in the MATCH_RECOGNIZE clause is composed of multiple variables, and different quantifiers may bind these variables [12]. For example, variable N in the query Q_1 is bound by the '*' quantifier, which allows this variable to match zero or more rows. In addition, the variable N is also bound by the exclusion operator '{- -}'. This operator will exclude events matched by variable N from the output of ALL ROWS PER MATCH. For query Q_1, since the three events of interest to the user do not require strict continuity, the '{-N*-}' is placed between variables R and B, and between B and M to support the semantics of skipping irrelevant events. Without '{-N*-}', events within the match must be strictly contiguous. Then, applying independent filtering conditions alone would break this contiguity requirement, thereby invalidating the push-down optimization. Besides, push-pull is limited to queries with the semantics of skipping irrelevant events [2]. In contrast, our filtering strategy leverages variable predicates and time windows to shrink the temporal range, transmitting only events within the refined intervals. This strategy has been proven to yield query results equivalent to retrieving the entire event set under strict-contiguous semantics [31].

For the conditions in the DEFINE clause, they can be categorized into independent and dependent conditions [31]. Both conditions are Boolean expressions: the former is evaluated on a single event, and the latter on multiple events. For convenience, we further divide the dependent conditions into equality and non-equality conditions. Equality conditions involve two variables, with their corresponding attributes bound by '=' operator to compare attribute values. Any condition not meeting this criterion is classified as a non-equality condition. In this paper, we omit using non-equality conditions for filtering due to their diversity, inherent complexity, and the lack of a generalized, low-overhead communication strategy.

[1]Notably, the standard SQL does not allow window conditions to be specified directly in the PATTERN clause; instead, they should be defined in the DEFINE clause. Here, we use the WITHIN clause to simplify queries.

B. Problem Formulation

Definition 1 (Event Set)**.** *An event set E contains N unordered events $\{e_1, ..., e_N\}$, each event following the same schema $(\mathcal{A}^1, ..., \mathcal{A}^d, \mathcal{T})$. Here, $\mathcal{A}^i$ denotes the i-th attribute of events, and $\mathcal{T}$ refers to an event occurrence timestamp [32].*

Definition 2 (Node)**.** *Typically, cloud-native architectures have two types of nodes: compute nodes (CNs) and storage nodes (SNs). CN provides the central query processing capability. It consists of CPU and memory resources and is responsible for parsing queries, generating execution plans, pulling data from SNs, and producing the final query results. SN manages persistent data and serves data retrieval requests from CN. It is equipped with disk storage and limited CPU and memory resources for lightweight operations. When multiple SNs are deployed, data are partitioned across them via hashing, and SNs do not communicate with each other [26].*

Definition 3 (Pull Strategy)**.** *A pull strategy is an event retrieval approach in which CN requests specific events from multiple SNs to ensure that the events participating in the matching are included in the requested event set.*

Definition 4 (Query Latency)**.** *Query latency is the time from CN receiving a query to producing all results.*

Then, the problem studied in this paper is **in computing and storage separation environments, how to design an efficient pull strategy to pull events from multiple SNs, so that CN can minimize query latency?**

III. High-level Idea and Workflow

A. High-level Idea

As mentioned in Section II-B, recall that the unordered historical event set is randomly partitioned across multiple SNs. Intuitively, if we can identify the shortest time interval set (TIS) that contains matches[2] and only transmit events within TIS (note that independent conditions will be used to filter when applicable), CN will pull the fewest events, minimizing event transfer cost. However, identifying the shortest TIS requires running a costly matching algorithm on SNs, which increases pulling cost and overall query latency. To address this trade-off, we propose a lightweight dual-filtering strategy to identify shorter TIS instead of the shortest ones for pulling events, which involves the following three core ideas:

Multiple round-trips for identifying the time interval set are sufficient to reduce network overhead (Section IV-A). Without gathering time interval and attribute information, SNs cannot apply the window-based pruning strategy. Even with one additional round-trip to gather such information, irrelevant data may still inevitably be transmitted. In this paper, we employ multiple round-trips between CN and SNs to progressively shrink TIS. Specifically, CN first obtains an

[2]For a match result $(e_{i_1}, ..., e_{i_n})$ of the query, the time interval of this match is $\left[\min\{\mathcal{T}[e_{i_1}], ..., \mathcal{T}[e_{i_n}]\}, \max\{\mathcal{T}[e_{i_1}], ..., \mathcal{T}[e_{i_n}]\}\right]$, where $\mathcal{T}[e_i]$ denotes the timestamp of e_i. Then, the shortest TIS is the union of all time intervals of matches.

initial TIS from SNs, then sends requests to obtain updated TIS for unprocessed individual variables. Upon receiving an update request, SNs first filter out events outside received TIS (temporal-based event pruning, *i.e.*, for any event pair (e_i, e_j) to form a partial match, their timestamps must satisfy $|\mathcal{T}[e_i] - \mathcal{T}[e_j]| \leq w$, where w is the window size of the pattern). Then, SNs check the equality condition and prune time intervals without matches by predicate-based temporal pruning (*i.e.*, any time interval containing no events satisfying the variable's predicates can be safely pruned), and send the updated TIS to CN. CN merges all received TISs from SNs to obtain the updated TIS. This process reduces the transmission of irrelevant time intervals and attribute information.

Membership testing data structures are space-efficient and provide low communication latency (Section IV-B). We observe that using an array to maintain TIS has $O(N \log N)$ complexity, where N is the number of events. This high complexity leads to significant latency of round-trips when N is large. We propose the shrinking window filter (SWF) to approximately store the time intervals, which provides $O(1)$ complexity for insertion/lookup, thereby reducing the maintenance complexity of TIS to $O(N)$. Besides, processing equality conditions requires CN to gather attribute and timestamp information and push them to SNs, leading to an expensive transmission cost. We propose a space-efficient window-wise join filter (WJF) to store this information, thereby reducing network overhead and alleviating the latency of round-trips.

Detrimental round-trips can be avoided via a cost model (Section IV-C). Employing fine-grained round-trips may negatively impact overall latency. For example, variables with extremely high selectivity often cannot efficiently shrink TIS, while incurring substantial computational overhead. As a result, corresponding round-trips are detrimental to the overall latency. Fortunately, such negative round-trips can be eliminated via the cost model, *i.e.*, any round-trip expected to yield no overall benefits will be proactively removed.

B. Workflow

Combining these three high-level ideas, our approach follows this workflow: CN initially determines the variable processing order based on the selectivity of variables in the query pattern and pulls the TIS generated by the first processed variable from SNs. Next, the CN builds SWF to approximately store TIS, enabling efficient filtering and updates during subsequent processes. Although we use SWF, we refer to this as TIS for consistency. Before pulling a new TIS for an unprocessed variable, CN uses a cost model to estimate whether initiating new round-trip(s) would reduce overall latency. If yes, CN sends the local TIS to SNs for filtering and pulls the updated TIS (if the current variable has equality conditions with previously processed ones, an extra round-trip is initiated to collect attribute and timestamp information). Otherwise, it skips generating a new TIS for that variable and moves on to other unprocessed variables.

For SNs, when they receive TIS from CN, they filter out events outside TIS. Then, they generate a new TIS for the remaining events bound to the variable, intersect it with the received TIS, and send the updated TIS to CN. CN merges all TISs received from SNs to update its local TIS, completing the round-trip(s) of generating a new TIS for that variable.

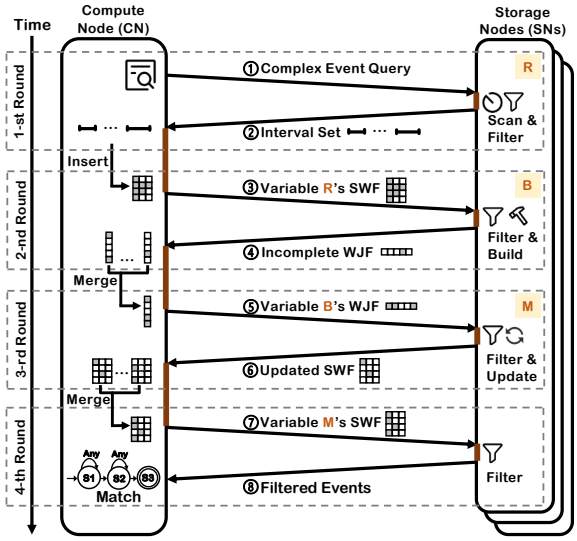


Fig. 4: Workflow between CN and SNs for the query Q_1.

Once TIS generation is complete for all variables, CN sends the local TIS to SNs and pulls events within TIS (the independent conditions will be used to filter when applicable). Finally, CN executes the matching algorithm to extract matches from the pulled events. Overall, for a query involving η variables, this workflow entails at least 2 round-trips between CN and SNs, and at most $2\eta + 1$ round-trips.

Example. Fig. 4 illustrates the workflow for the query Q_1, with the variable processing order `R`→`B`→`M` (variable `N` is ignored since it can match any row and has no impact on the output result). Initially, CN obtains the TIS generated by `R` during the first round-trip. Then, the cost model deems the round-trip for `B` detrimental, CN skips the TIS generation for B. Because `M` has the equality condition `M.distric=B.district` with `B`, the second round-trip aims to gather attribute and timestamp information specific to this condition for filtering, which is stored in WJF. In the third round-trip, CN pushes the complete WJF to SNs, which use TIS and WJF to filter events and update TIS. Finally, CN sends the final TIS to SNs for pulling events within TIS to match. In the next section, we will describe the specific operation of generating TIS on SNs.

IV. Design

A. Identification of Time Interval Set

For a given complex event query, let v_i denote the i-th variable in the pattern and η denote the number of variables (*e.g.*, $\eta = 5$ for the query Q_1). Initially, CN sends the query to SNs to obtain the initial TIS. When receiving the query, SNs retrieve relevant events from disk, identify the event set E_{v_i} that holds the independent conditions of variable v_i, and cache them in memory. Then, SNs generate TIS that may

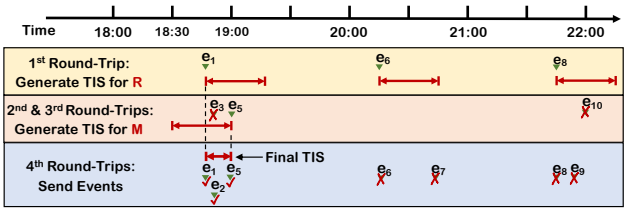


Fig. 5: TIS generation over varying round-trips.

contain matches for the first processed variable and send the generated TIS to CN. Specifically, for an event e_j in E_{v_i}, the time interval δ_{e_j} that e_j generates is

$$\delta_{e_j} = \begin{cases} [\mathcal{T}[e_j],\ \mathcal{T}[e_j] + w]\,, & i = 1, \\ [\mathcal{T}[e_j] - w,\ \mathcal{T}[e_j]]\,, & i = \eta\ , \\ [\mathcal{T}[e_j] - w,\ \mathcal{T}[e_j] + w]\,, & \text{otherwise.} \end{cases} \tag{1}$$

When receiving all TISs from SNs, CN merges them to generate the initial TIS. Then, CN uses the cost model to determine whether to initiate new round-trip(s) to generate a new TIS for the remaining unprocessed variables. If new round-trip(s) can reduce the overall query latency, CN sends the local TIS to SNs. Then, for each e_j in E_{v_i}, SNs start the following two-phase verification:

Phase 1: Timestamp verification. SNs first check whether $\mathcal{T}[e_j]$ falls within TIS. If yes, SNs proceed to the second phase verification. Otherwise, SNs filter out e_j since it cannot be involved in matching (*temporal-based event pruning*).

Phase 2: Attribute verification. SNs check whether v_i has equality conditions with previously processed variables. If none exist, SNs generate a time interval for e_j. Otherwise, SNs calculate the window ID of e_j by $\mathcal{W}_{e_j} = \lfloor \mathcal{T}[e_j]/w \rfloor$ and check whether the corresponding attribute of e_j exists in the current or adjacent windows (either the previous or the next window, depending on the temporal order of variables). If yes, SNs generate a time interval for e_j. Otherwise, SNs filter out e_j. Notably, WJF is responsible for this verification (it has already been generated in the previous round-trip).

After generating a time interval, SNs insert it into the newly generated TIS, ensuring that all intervals are non-overlapping. Once all events are processed, SNs intersect the new generated TIS with the previous TIS to obtain an updated TIS (*predicate-based temporal pruning*) and send the updated TIS to CN. When receiving all updated TISs from SNs, CN merges them to update its local TIS. After completing TIS generation for all variables, the local TIS of CN is the final TIS that we identify, and this final TIS is sent to SNs to pull events.

Example. Following the workflow in Section III-B, Fig. 5 illustrates the TIS generation over varying round-trips for the events listed in Table I. Initially, SNs identify events that hold the independent conditions of all variables and generates TIS, *i.e.*, {[18:45, 19:15], [20:15, 20:45], [21:43, 22:13]}, for variable R' events $\{e_1, e_6, e_8\}$. In the third round-trip, since e_3 and e_{10} fail attribute verification, they are filtered out. As a result, SNs only generate a new time interval [18:30, 19:00] for e_5. Then, the newly generated TIS is intersected with the previous TIS to shrink it, and the updated TIS is sent to the CN. In the last round-trip, SNs receive the final TIS [18:45, 19:00] from CN and transmits only the events $\{e_1, e_2, e_5\}$ with timestamps within the TIS.

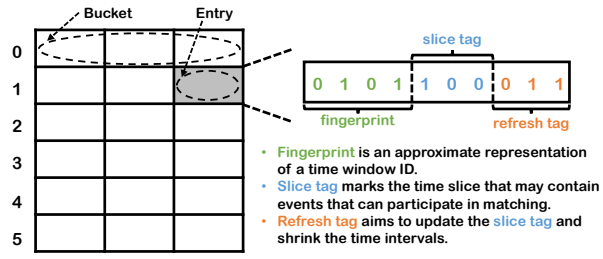


Fig. 6: Illustration for the structure of SWF.

B. Design of Membership Testing Data Structures

1) Storage of Time Interval Set: We propose the shrinking window filter (SWF) to approximately store time intervals. Fig. 6 illustrates the structure of SWF. SWF consists of an array with m buckets, each containing d entries. Each entry includes three fields: a fingerprint (denoted as f), a slice tag (denoted as q), and a refresh tag (denoted as r). Specifically, ❶ f is an approximate representation of the window ID. ❷ q represents multiple time slices (a time slice is the smallest time range stored in SWF, and its length is w/l_t, where l_t is the bit length of q). If a bit bound by a time slice is set to 1, it indicates that the corresponding time range is included in SWF; otherwise, it is not. ❸ r is used for updating the slice tag to shrink the time intervals.

Essentially, SWF stores the time interval via fingerprint-based hashing and bitmap techniques. A time range is partitioned into same-sized windows (the windows are stored by fingerprints to save space), each of which is subdivided into fixed-size time slices, and the bitmap of time slices is used to mark the range of time intervals that can be covered. For insertion, since each generated time interval of an event spans at most 3 windows by Equation 1, SWF only needs a constant number of hash computations to insert a time interval. For timestamp lookup, SWF only needs to calculate the window ID where an event is located and check whether the corresponding slice bitmap is 1 to determine whether it is in TIS. Thus, SWF achieves $O(1)$ complexity for insertion and lookup.

Besides, we provide two key optimizations for SWF to reduce the transmission cost, *i.e.*, ❶ Tag Separation: We observe that the refresh tag remains at 0 during the transmission of SWF between CN and SNs. Besides, for SWFs transmitted by CN and SNs, entries at the same location always have the same fingerprint. To alleviate unnecessary bit transmission, we propose a tag separation mechanism, *i.e.*, from CN to SNs, transmitted SWF excludes the refresh tags; while from SNs to CN, transmitted SWF excludes the fingerprints and slice tags. ❷ Compaction: Inspired by prior work [33], we introduce a compaction operation for SWF when its load factor is low. Specifically, after obtaining the merged SWF, we clear the entries with slice tags of zero and check whether the load factor of SWF is below a predefined threshold. If yes, we repeatedly halve the number of buckets and reinsert entries from the removed buckets into the retained buckets until the

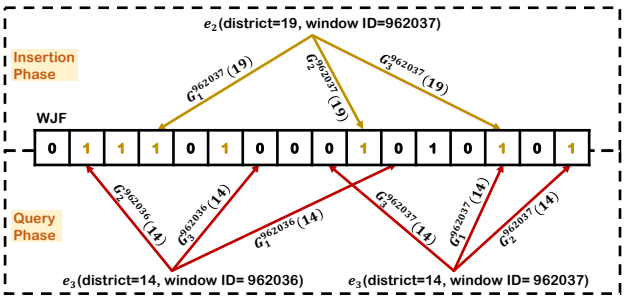


Fig. 7: Illustration for the structure of WJF and its insert/query operations for the variables `B` and `M`, where $k = 3$.

load factor exceeds the threshold (if the insertion fails, we will roll back to the previous state).

2) Storage of Attribute and Timestamp Information: We observe that transmitting the set of (attribute, window ID) pairs of events suffices instead of their (attribute, timestamp) pairs. The reason is that SWF can verify window constraints between events, and equality conditions provide implicit constraints on the window IDs. For example, in query Q_1, variables `B` and `M` have an implicit window condition `B.windowID=M.windowID OR B.windowID+1=M.windowID`.

Inspired by this observation, we design a space-efficient window-wise join filter based on BF [34], to store the set of (attribute, window ID) pairs. Specifically, WJF contains a bit array and k independent hash functions $G_1^{\xi}(\cdot), \cdots, G_k^{\xi}(\cdot)$ obeying uniform distribution, where ξ is the hash seed. Initially, all bits in the bit array are set to 0. When inserting a (attribute, window ID) pair to WJF, WJF uses the window ID as the hash seed and the attribute value as the key to calculate k hash positions. Then, WJF sets the bit values of k mapped position to 1. During attribute verification for event e_j in Section IV-A, we calculate e_j's window ID $\mathcal{W}_{e_j} = \mathcal{T}[e_j]/w$ and the adjacent window ID for querying. Then, for each window ID, we calculate k bit positions based on the attribute value. If there is no window where all mapped bits are 1, e_j cannot satisfy the equality condition and can be filtered out.

Example. Fig. 7 illustrates the structure of WJF, along with examples of insertion and lookup for the variables `B` and `M` in the query Q_1. When inserting variable `B`'s event e_2's (attribute, window ID) pair, *i.e.*, (19, 962037) into WJF, we calculate 3 mapped positions $G_1^{962037}(19), G_2^{962037}(19)$, and $G_3^{962037}(19)$ for the pair, and set the value of mapped bits to 1. When checking variable `M`'s event e_3, since `M` occurs after `B`, we verify whether the 962036th or 962037th window has the attribute value 14. Since not all 3 mapped positions are 1, event e_3 can be filtered out.

C. Cost Model of Round-trip

After CN obtains the initial TIS, it uses the cost model to determine whether generating a new TIS based on the variable v_i to shrink the TIS. Initiating new round-trip(s) to generate a new TIS offers two benefits: reducing event transmission overhead and decreasing the matching cost on CN. However, it also incurs additional costs: (1) two-phase verification, (2) network transmission in SWF and WJF, and (3) construction overhead for WJF. Here we use $\mathcal{L}_{benefit}$ to denote the latency benefit of initiating a new round-trip, and $\mathcal{L}_{harm}$ to denote the latency harm incurred by initiating new round-trip(s).

Estimation of $\mathcal{L}_{benefit}$. For the event set E, its average event generation rate is $r = \frac{|E|}{\max\{T[e_i]|e_i \in E\} - \min\{T[e_i]|e_i \in E\}}$. Suppose that initiating new round-trip(s) to generate new TIS can decrease the covered length of the local TIS of CN by $\Delta\ell$ and the matching latency is proportional to the number of events for a given query [32]. Then, the benefit of initiating new round-trip(s) for v_i is estimated by

$$\mathcal{L}_{benefit} = \frac{\Delta\ell \mathcal{S}_e r}{\mathcal{M}\mathcal{R}} + \beta\Delta\ell r, \tag{2}$$

where $\mathcal{R}$ is the data transfer rate in the network, $\mathcal{M}$ is the number of SNs, and $\mathcal{S}_e$ is the serialization size of an event. During system initialization, we proactively transmit a 10MB payload to estimate the initial $\mathcal{R}$. During runtime, $\mathcal{R}$ is continuously updated based on the transmission latency and the data volume observed in the most recent query execution. To measure β, when receiving a new query, CN will run the matching algorithm on a sample of 5000 events to estimate the average matching cost per event.

Since the current covered length ℓ of TIS is known before initiating new round-trip(s), when the selectivity of variable v_i is sel_i and event arrival follows a Poisson process (we adopt this assumption primarily because prior work [35]–[37] has made the same assumption for the real-world datasets used in our evaluation), we can estimate $\Delta\ell$ by the following equation:

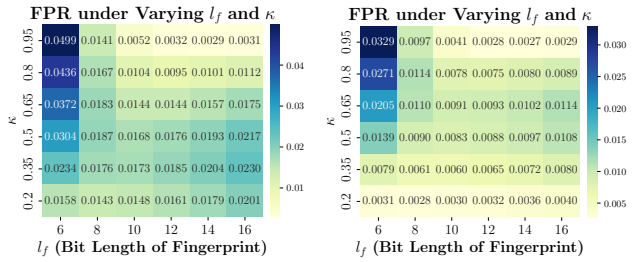
$$\Delta\ell = \begin{cases} \ell e^{-\lambda_i w}, & i \in \{1, \eta\}, \\ \ell e^{-2\lambda_i w}, & \text{otherwise}, \end{cases} \tag{3}$$

where $\lambda_i = \frac{sel_i|E|}{\max\{T[e_i]|e_i \in E\} - \min\{T[e_i]|e_i \in E\}}$, which represents the average occurrence rate of the event for variable v_i. Here we omit the proof to save space.

Estimation of $\mathcal{L}_{harm}$. Let C_{p_1} denote the average per-event cost of validation with SWF in the first phase, C_{p_2} denote the average per-event cost of validation with WJF in the second phase, and C_b denote the average per-event cost of constructing the WJF. Then, the detrimental latency incurred by initiating new round-trip(s) is estimated by

$$\mathcal{L}_{harm} = \frac{1}{\mathcal{M}}(sel_i|E|C_{p_1} + \ell\lambda_i\mathcal{H}C_{p_2} + \mathcal{K}C_b) + \frac{\mathcal{S}_t}{\mathcal{R}}, \tag{4}$$

where $\mathcal{H}$ is the number of equality conditions between the already processed variables and the current variable v_i, $\mathcal{K}$ is the number of (attribute, window ID) pairs involved in constructing the WJF, and $\mathcal{S}_t$ is the space consumption of SWF and WJF (their estimated values will be provided in § V). Clearly, $\mathcal{H}$ can easily be determined based on the query feature. To determine the value of $\mathcal{K}$, we need to record both the number of events corresponding to the processed variables and the coverage length of the TIS at that time. Then, we can proportionally scale down the number of events based on the current coverage length of the TIS to obtain the count of (attribute, window ID) pairs required for constructing WJF



(a) $p_1 = 1$ or $p_1 = \eta$ (b) $p_1 \neq 1$ and $p_1 \neq \eta$

Fig. 8: FPR v.s. varying parameter settings, where $\alpha = 0.8$.

for the relevant variable. Since C_{p_1} is primarily determined by the number of buckets in SWF, and the number of buckets is powers of 2 [38], we use a pre-measured lookup table over these discrete bucket configurations to obtain the corresponding cost. In contrast, C_{p_2} varies with both the bit-array length and the attribute byte size. Because this parameter space is too large to exhaustively profile, we employ piecewise-linear interpolation to estimate C_{p_2}. Since the construction cost of the WJF also depends on the bit-array length and attribute byte size, C_b is estimated in the same strategy as C_{p_2}.

After estimating $\mathcal{L}_{benefit}$ and $\mathcal{L}_{harm}$, if $\mathcal{L}_{benefit} \geq \mathcal{L}_{harm}$, we initiate new round-trip(s) for variable v_i; otherwise, we skip the TIS generation for v_i to avoid detrimental round-trips.

V. Theoretical Analysis

In this section, we provide a theoretical analysis of the false positive rate (FPR) and space consumption for SWF and WJF. Following prior work [28], [35]–[37], we model the occurrence of events that satisfy the independent conditions of v_i follows a Poisson process. Let v_{p_1} denote the first processed variable and λ_1 the average arrival rate of events associated with it.

A. Theoretical Analysis of SWF

1) FPR: The false positive of SWF refers to SWF mistakenly classifying events outside TIS as being within TIS. Since SWF on CN varies across round-trips, we focus on analyzing the initial SWF's FPR to determine its optimal parameters.

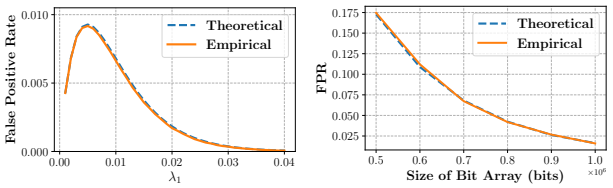
SWF can produce false positives in two cases: (1) *hash collision*: Different window IDs produce identical fingerprints mapped to the same buckets. In this case, we continue to verify whether the corresponding bit value in the slice tag is set to 1. If yes, a false positive has occurred. And (2) *longer range*: Coarser time slices represent a longer time range than the actual. Based on the two cases of false positives, Theorem 1 gives the theoretical FPR of the initial SWF.

Theorem 1. *For the initial SWF, its upper bound of FPR ϵ is*

$$\epsilon = \begin{cases} \kappa^2 \cdot \frac{2d\alpha}{2^{l_f}} \cdot \frac{l_t(1-\kappa) - \kappa \ln \kappa}{l_t(1-\kappa^2)} + \frac{\kappa \ln \kappa^{-1}}{l_t}, & p_1 \in \{1, \eta\}, \\ \kappa^3 \cdot \frac{2d\alpha}{2^{l_f}} \cdot \frac{l_t(1-\kappa) - \kappa^2 \ln \kappa}{l_t(1-\kappa^3)} + \frac{\kappa^2 \ln \kappa^{-1}}{l_t}, & \text{otherwise}, \end{cases} \tag{5}$$

where $\kappa = e^{-\lambda_1 w}$, α is the load factor of SWF, and l_f, l_t are the bit length of fingerprint and slice tag.

Typically, the bit length of an entry in SWF is 32. When enabling the optimization of tag separation in Section IV-B1, we have $l_f + l_t = 32$. Fig. 8 shows the FPR of the initial SWF under varying parameter settings, where the load factor of SWF is set to 0.8. Clearly, no specific l_f has a minimum FPR at any κ value. When l_f is 10 or 12, we can obtain a generally lower FPR.



(a) FPR of SWF (b) FPR of WJF

Fig. 9: Theoretical vs. empirical FPR value.

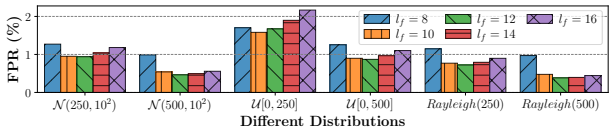


Fig. 10: FPR variations of SWF across different distributions.

Verification. Fig. 9a shows that the theoretical FPR of initial SWF fits well with the empirical FPR when $d = 4, l_f = 12, l_t = 20, w = 100$, $p_1 = 1$, and λ_1 increases from 0.001 to 0.04, where ground truth comes from the interval array.

Discussion. Considering that the inter-arrival times between consecutive events may not follow a Poisson process, we further evaluate FPR of SWF under Gaussian, Uniform, and Rayleigh distributions. As shown in Fig. 10, when l_f is set to 10 or 12 (note that $l_f + l_t = 32$), SWF still achieves a generally lower FPR.

2) Space Consumption: When enabling the optimization of tag separation, the space consumption of SWF is $md(l_f + l_t)$ bits, where m is the number of buckets, and d is the number of entries. To avoid insertion failure, we need to set an appropriate m. Note that when d is fixed, the cuckoo hashing mechanism has a maximum load factor α_d [38]. Then, after obtaining the initial TIS, CN estimates the number of windows it spans, denoted as n_e, and sets m to $2^{\lceil \log_2 \frac{n_e}{d\alpha_d} \rceil}$.

B. Theoretical Analysis of WJF

1) FPR: WJF uses the window ID as the hash seed and the attribute value as the key to calculate k hash positions. When we assume that hash functions with varying seeds exhibit uniform and independent distributions, the probability of a bit being set to 1 is $p = 1 - (1 - \frac{1}{m_b})^{kn_p}$ [34], where m_b is the bit array size and n_p is the number of unique pairs. Then, the probability of WJF mistakenly reporting yes for a non-existent pair is p^k. For the attribute value of an event, we will check two window IDs, thereby the overall FPR of WJF is $p^k + (1 - p^k) \cdot p^k$. Note that when $k = \frac{m_b}{n_p} \ln 2$, WJF achieves the lowest FPR, which is 2^{-k} [34]. According to Section IV-A, when using WJF to filter, an attribute value is checked for existence in two windows, equivalent to performing two queries. Thus, the FPR of WJF is

$$\epsilon^* = 2^{-k} + \left(1 - 2^{-k}\right) \cdot 2^{-k} \approx 2^{1 - \frac{m_b}{n_p} \ln 2}. \tag{6}$$

Verification. Fig. 9b shows that the theoretical FPR value of WJF fits well with the empirical FPR value when $n_p = 10^5$

and m_b increases from 5×10^5 to 10^6, where ground truth comes from a hashmap storing (attribute, window ID) pairs.

2) Space Consumption: Given a desired FPR ϵ^* and the number of unique pairs n_p for WJF, the space consumption of the bit array is $m_p = \frac{n_p}{\ln^2 2}(\ln 2 - \ln \epsilon^*)$.

VI. Experimental Evaluation

A. Experimental Setup

1) Implementation: Guided by the core principles of cloud-native architectures, we implement a system with disaggregated compute and storage resources for CER, in which SNs leverage EBS or S3 as the backend to provide elastic storage capacity, and the number of CNs can be scaled according to query concurrency. Moreover, we integrate three evaluation engines on CNs, including SASE-E (a lightweight non-deterministic finite automata based on SASE [39]), Esper [40], and Flink [14], to process complex event queries. Note that we choose local Flink over serverless Flink because local Flink achieves performance comparable to serverless Flink while avoiding the additional extract–transform–load overhead. SNs adopt a row-oriented storage format with a page size of 4 KB. For pulling strategies that involve multiple round-trip, SNs cache intermediate results. All SNs respond to CN requests in parallel. Finally, our source code is publicly available [41].

2) Environment: We conduct all experiments on AWS EC2 instances. By default, we select a c5.2xlarge instance (launched in us-east-1b, with 8 vCPU, 16 GB memory, and a 10 Gbps network bandwidth) as CN, with the default `Xmx` setting of 3.4 GB by the system (`Xmx` specifies the maximum memory allocation pool of JVM). Besides, we select three c5a.xlarge instances (launched in us-east-1a, with 4 vCPU, 8 GB memory, and a 10 Gbps network bandwidth) as SNs, and `Xmx` is set to 1.7GB. We use Amazon S3 [42] or Amazon Elastic Block Store (EBS) [43] as the storage backend for SNs to store partitioned events. Unless otherwise specified, EBS is used by default for three reasons: ① it provides higher I/O throughput than S3; ② it supports on-demand provisioning and independent scaling [44]; ③ it is widely adopted by production cloud databases, including Amazon RDS [45], MongoDB Atlas [46], TiDB Cloud [47].

3) Baselines: ① Push-down pushes independent conditions down to SNs, and SNs filter irrelevant events before sending them to CN. ② Push-pull employs multiple round-trips to retrieve events. In each round, it pushes partial received events to each SN, where matching algorithms are executed to perform filtering by leveraging all predicate conditions. ③ Muse [3] operates in fully distributed architectures, requiring data shuffling and communication between every node, which conflicts with cloud-native architectures. Thus, in our experiments, we only report Muse's transmission cost with four nodes. ⑤ Pull-all requests all events to match, and we only use it for the queries with strict-contiguous semantics. ⑤ Besides, we also add a naive DFS (*i.e.*, Naive-DFS) approach as a baseline, which uses arrays to store TIS and hash tables to store (attribute, window ID) pairs. Naive-DFS relies on a dual-filtering strategy to identify TIS containing matches in a single round-trip. Regarding parameters for our DFS, in SWF, we set $d=4, l_f=12$, and $l_t=20$; and in WJF, we set $\epsilon^*=0.02$.

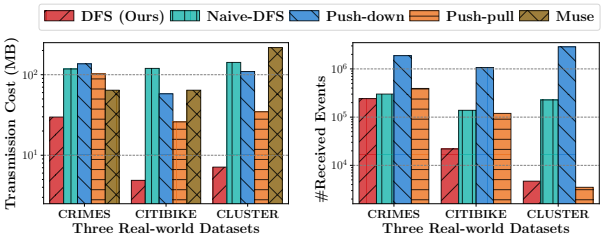


Fig. 11: Comparison of average transmission cost.

Fig. 12: Comparison of average #received events.

4) Datasets: We use three real-world datasets from existing work [2], [32], [48]: CRIMES [21], CITIBIKE [49], and CLUSTER [50]. ① CRIMES dataset records crime reports in Chicago from 2001-01 to 2024-12, containing about 8M events. ② CITIBIKE dataset describes the information on each ride in 2023, containing about 35M events. ③ CLUSTER dataset records the CPU, RAM, and disk resource consumption of each task, containing about 144M events. Note that CRIMES dataset is sensitive and may reflect reporting and policing biases; it is used here solely as a technical benchmark.

Besides, we further generate synthetic data to achieve a controlled setup. The synthetic event follows the schema: *(char(4) type, int a1, char(8) a2, char(16) a3, long timestamp)*, where the event type obeys a Zipf distribution *Zipf*$(1.2, 20)$ [51], a1 obeys uniform distribution $\mathcal{U}[1, 1000]$, a2 and a3 obey *Zipf*$(0.255, 50)$ and *Zipf*$(0.7, 50)$, respectively. In the synthetic dataset, events are generated according to a Poisson process with an arrival rate of one event per second.

5) Query: Unless otherwise specified, for the query workloads of each dataset, all queries adopt the semantics of skipping irrelevant events, as this semantics is the most common in real-world query scenarios [11], [31], [52]–[54]. To save space, we only report the features of patterns in the query workloads. More details can be found in our code repository [41].

- On CRIMES dataset, we search for sequences of `ROBBERY`, `BATTERY`, and `MOTOR VEHICLE THEFT` events occurring within 30 minutes and close spatial proximity, which are widely used in prior work [31], [32], [55].
- On CITIBIKE dataset, to avoid out-of-memory issues with Push-pull, we shorten the window size and add equality conditions based on prior queries [2], [51]. Specifically, we search for patterns involving three ride events that occur within 5 minutes, where the end station of one ride matches the start station of the next.
- On CLUSTER dataset, we search for patterns in which tasks sharing the same job_id are submitted, scheduled, and finished/killed within 3 seconds [48].
- On the synthetic dataset, we search for the pattern containing η variables $v_1, ..., v_\eta$. For each variable v_{2i}, it can match any row. The probability that any two variables v_{2j-1} and v_{2j+1} have an equality condition is $\min\{1, \frac{3}{\eta-1}\}$. Lastly, the selectivity of attributes for each variable v_{2j+1} is 20%.

TABLE II: Comparison of average latency on the three real-world datasets, where TO represents the latency $>$ 5 minutes.

Storage backend	Dataset	Engine	1 CN (us-east-1b) + 3 SNs (us-east-1a)				1 CN (us-west-1a) + 3 SNs (us-east-1a)			
			DFS (Ours)	Naïve-DFS	Push-down	Push-pull	DFS (Ours)	Naïve-DFS	Push-down	Push-pull
S3	CRIMES	SASE-E	**1.05s**	2.47s	2.03s	1.84s	**1.83s**	3.65s	3.66s	3.30s
		Esper	**1.39s**	2.85s	3.79s	2.26s	**2.19s**	4.01s	5.39s	3.69s
		Flink	**5.83s**	9.74s	33.57s	9.02s	**7.01s**	10.92s	35.20s	10.45s
	CITIBIKE	SASE-E	**1.56s**	3.42s	2.19s	1.89s	**2.61s**	4.23s	3.12s	3.10s
		Esper	**1.89s**	3.87s	3.41s	2.25s	**2.91s**	4.79s	4.31s	3.45s
		Flink	**3.36s**	18.37s	67.35s	5.01s	**4.41s**	19.18s	68.28s	6.22s
	CLUSTER	SASE-E	**4.59s**	7.59s	8.03s	6.05s	**5.82s**	8.93s	9.31s	7.89s
		Esper	**4.94s**	8.93s	160.01s	6.37s	**6.17s**	10.14s	164.01s	8.24s
		Flink	**6.71s**	18.75s	TO	8.31s	**7.53s**	20.09s	TO	10.15
EBS	CRIMES	SASE-E	**0.73s**	2.07s	1.29s	1.44s	**1.33s**	3.25s	3.24s	2.78s
		Esper	**1.06s**	2.50s	3.02s	1.89s	**1.65s**	3.40s	4.98s	3.20s
		Flink	**5.52s**	9.31s	32.81s	8.63s	**6.13s**	10.54s	34.78s	9.96s
	CITIBIKE	SASE-E	**0.82s**	2.61s	1.30s	1.17s	**1.44s**	3.70s	2.30s	2.41s
		Esper	**1.22s**	3.15s	2.53s	1.53s	**1.79s**	4.16s	3.54s	2.75s
		Flink	**2.62s**	17.65s	66.46s	4.28s	**3.23s**	18.65s	67.46s	5.53s
	CLUSTER	SASE-E	**2.87s**	6.63s	6.07s	5.37s	**3.96s**	8.71s	7.57s	5.61s
		Esper	**3.24s**	8.01s	158.7s	5.68s	**4.32s**	9.04s	159.12s	5.92s
		Flink	**4.83s**	17.86s	TO	7.63s	**5.08s**	18.87s	TO	7.87s

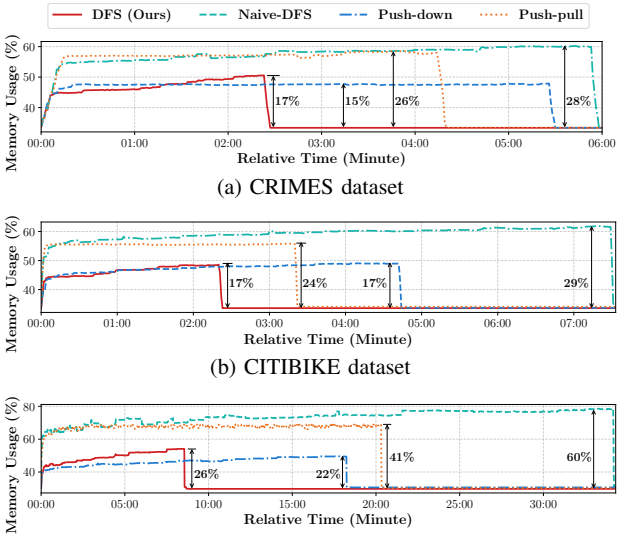
B. Results for Real-world Datasets

In this subsection, we compare the performance of different approaches in real query scenarios. We generate 150 queries for each dataset. To enhance diversity, we vary attribute value ranges and incorporate non-equality conditions randomly.

1) Comparison of Transmission Cost and Number of Received Events: ① Fig. 11 shows the average transmission cost of queries between CN and SNs. Our proposed approach, *i.e.*, DFS, consistently has the lowest transmission cost. Compared with Push-down, DFS reduces transmission cost by around 78%, 92%, and 93% on the three real-world datasets. Compared with Push-pull, DFS reduces it by around 71%, 81%, and 79% on the same datasets. As for Muse, it pulls events or partial matches from all other nodes for pattern matching, which causes significant information redundancy and high transmission cost. ② Fig. 12 shows the number of events received by CN from SNs. When using additional windows and equality conditions, DFS reduces the number of events transmitted by 77%, 97%, and 99% compared with Push-down. Although Push-pull can use all conditions for filtering, events that are sent to CN early do not benefit from these conditions. As a result, CN may receive more events from Push-pull than from DFS.

2) Comparison of Query Latency: Table II reports the average query latency across different evaluation engines and storage backends. Overall, DFS achieves a 1.2$\times$ to 25$\times$ speedup across various evaluation engines and storage backends compared with the baselines. The reasons for the low latency of DFS are twofold: ① it incurs relatively low communication overhead when identifying the final TIS; and ② it significantly reduces the number of events sent to the CN, thereby lowering both network transmission and engine matching costs. Notably, when using SASE-E as the evaluation engine, all approaches achieve the lowest query latency due to its lightweight implementation, whereas Flink exhibits higher matching costs, substantially increasing overall latency. On CITIBIKE and CLUSTER datasets, large numbers of events per window generate many partial matches, causing Push-down to incur prohibitively high latency when Flink is used.

Furthermore, when SNs use S3 as the storage backend, retrieving events incurs about 0.3s to 2s higher latency than retrieving them from EBS, while the matching cost on CN and the communication overhead between CN and SNs remain largely unchanged. This increase is mainly due to S3's RESTful, object-based interface, which introduces higher software stack overhead and per-request latency compared with EBS's block-level interface. Additionally, deploying CN and SNs across distant regions further increases the query latency, since filtered events need to traverse longer network paths.



(a) CRIMES dataset

(b) CITIBIKE dataset

(c) CLUSTER dataset

Fig. 13: Memory usage under sequential query workloads.

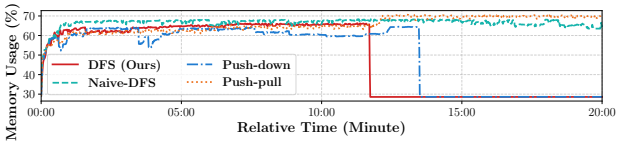


Fig. 14: Memory usage under concurrent query workloads.

3) Comparison of Memory Usage: We first monitor the memory usage of SNs using CWAgent [56] when executing sequential query workloads with a single thread. Fig. 13 shows the results. Since Push-down only holds events that satisfy independent conditions in memory, it has the lowest memory footprint. For DFS, it not only holds events but also maintains

TABLE III: Pattern generation templates.

ID	Generation templates with strict continuous semantics
P1	`PATTERN (A B C) WITHIN #window_size MINUTE`
P2	`PATTERN (A B C D) WITHIN #window_size MINUTE`
P3	`PATTERN (A B C D E) WITHIN #window_size MINUTE`

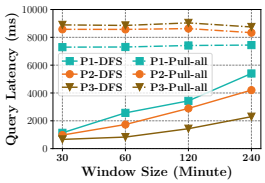


Fig. 15: Query latency of varied patterns on CRIMES.

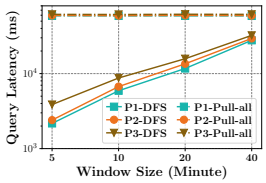


Fig. 16: Query latency of varied patterns on CITIBIKE.

SWF and WJF. Fortunately, due to their relatively low memory consumption, the peak memory usage of DFS is less than 4% higher than that of Push-down across all real-world datasets. In contrast, Push-pull additionally holds partial matches in memory. Although we use event references rather than raw events to represent matches, the large number of matches still causes Push-pull to exhibit higher peak memory usage than Push-down. For Naive-DFS, it needs to build multiple arrays to store the time intervals for each variable, and build the hash table to store the set of (attribute, window ID) pairs in one round-trip. Since most events are not within the final TIS, Naive-DFS consumes a significant amount of memory space.

Then, we add two additional CNs (three in total). Each CN hosts five tenants, which submit queries in parallel. We monitor the memory usage of SNs under this concurrent workload (500 queries across varied datasets). Note that each SN maintains an intermediate state for each query and adopts a pessimistic admission policy: if available memory drops below 20% of the usable maximum, new queries are paused until sufficient memory is freed to avoid out-of-memory errors. Fig. 14 reports the results. Due to DFS's additional space overhead in SWF and WJF, it is more likely than Push-down to trigger query pauses. As a result, its concurrency advantage is less pronounced than under single-threaded sequential execution. Naive-DFS and Push-pull fail to complete all queries within 20 minutes due to high memory consumption and low concurrent processing throughput on SNs.

4) Latency Evaluation for Queries with Strict-contiguous: We use the generation templates from Table III to construct query patterns with strict-contiguous semantics on CRIMES and CITIBIKE datasets across varied window sizes. Note that under the semantics, the baseline is Pull-all. Fig. 15 and Fig. 16 show the latency variation with window size for different approaches and query patterns, with each query executed five times. Clearly, for strict-contiguous patterns, DFS consistently outperforms Pull-all. For Pull-all, it shows near-constant latency for a fixed pattern because it retrieves all events from SNs regardless of window size. For DFS, on the sparse CRIMES dataset, increasing the number of variables sharply reduces the likelihood of matches within a window, leading to a shorter final TIS, fewer events pulled by CN, and thus lower latency. In contrast, on the dense CITIBIKE dataset, more variables do not significantly shrink TIS; instead,

TABLE IV: Comparison of average query latency across different node configurations on the synthetic dataset.

	Default settings for CN & SNs	Cross-region for CN & SNs	Heterogeneous instances for SNs
DFS (Ours)	1610.32ms	2583.56ms	4431.96ms
Naive-DFS	2272.32ms	3407.98ms	5174,12ms
Push-down	2788.72ms	4546.22ms	5334.46ms
Push-pull	3222.72ms	4759.5ms	6321.90ms

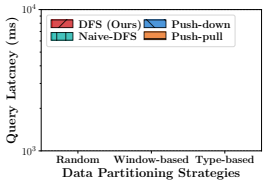


Fig. 17: Average latency versus partitioning strategies.

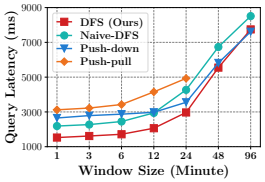


Fig. 18: Average latency versus window size.

they increase the number of pulled events, resulting in higher latency. On both datasets, the latency of DFS increases as the query window size grows.

C. Results for Synthetic Datasets

In this section, we evaluate the performance of different approaches under varying settings. Unless otherwise specified, we use a dataset of 50 million events, a workload of 50 queries, and a 3-minute query window. To reduce EC2 costs, we employ SASE-E as the evaluation engine.

1) Latency Evaluation on Different Node Configurations: Table IV reports the average query latency on the synthetic dataset under different CN and SN configurations. For the cross-region setting, CN is deployed on a c5.2xlarge instance in us-west-1a. For the heterogeneous SNs setting, three SNs are deployed on c5a.large (2 vCPU, 4GB memory), c5a.xlarge (4 vCPU, 8GB memory), and c5a.2xlarge (8 vCPU, 16GB memory) instances, respectively. When the network path between the CN and SNs becomes longer, DFS incurs the smallest additional latency compared with the other baselines. Under heterogeneous SN configurations, the latency is dominated by the weakest SN, *i.e.*, the c5a.large instance, whose limited 2 vCPU substantially reduces asynchronous read-and-filter throughput and significantly increases retrieval latency. Despite this, DFS remains about 0.7s faster than the best baseline. Overall, across all node configurations, DFS consistently achieves the best query performance.

2) Latency Evaluation on Varying Data Partition Strategies: By default, we use random event partitioning, which provides good load balancing. For comparison, we also evaluate window-based (time-range) and type-based partitioning. In the window-based strategy, events are grouped into 30-minute windows; all events in a window are stored on a single SN, and different windows are randomly assigned to SNs. In the type-based strategy, events of the same type are co-located on a single SN. Because event types are skewed, this can cause some SNs to hold far more events than others. For the type-based strategy, we configure queries to access only one partition to measure average query latency. Fig. 17 reports the latency under different partitioning schemes. Window-based partitioning yields a latency profile similar to random

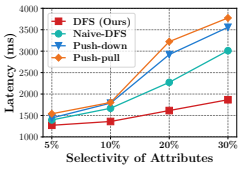


Fig. 19: Average latency vs. attribute selectivity.

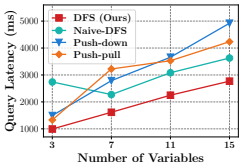


Fig. 20: Average latency vs. number of variables.

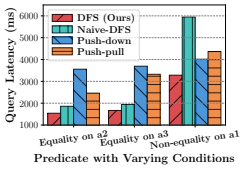


Fig. 21: Average latency vs. varied conditions.

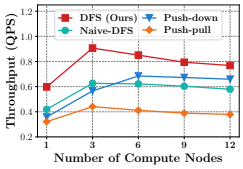


Fig. 22: Query throughput vs. number of CNs.

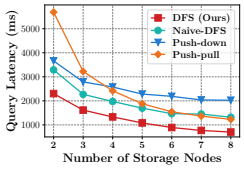


Fig. 23: Average latency vs. number of SNs.

partitioning, as both effectively distribute the load and avoid data skew. In contrast, type-based partitioning concentrates queries on a single SN, substantially increasing data loading and filtering overhead on that node, and thus query latency.

3) Sensitivity Analysis on Window Size, Attribute Selectivity, and Number of Variables: Fig. 18 shows the impact of window size on average query latency. As the window size increases, the average query latency of various approaches also increases. The reason is that longer windows increase the number of events involved in matching, thereby reducing the effectiveness of pruning and increasing the matching time. When the window size is very large (*e.g.*, $\geq$ 96 minutes), DFS cannot effectively prune events using window and equality conditions, and its multi-round communication results in slightly higher latency than Push-down. For Push-pull, enlarging the window leads to a rapid growth of partial matches, eventually exhausting memory on SNs. As a result, when the window exceeds 48 minutes, Push-pull fails to produce query results.

Next, we vary the attribute selectivity of each variable from 5% to 30% and report the query latency in Fig. 19. As attribute selectivity increases, more events are transmitted, increasing overall query latency. Nevertheless, DFS can still exploit window and equality conditions to effectively shrink the TIS and prune most events, consistently achieving the lowest latency. At 30% selectivity, DFS reduces query latency by 61% compared with the best baseline.

Finally, we vary the number of variables and report the average query latency in Fig. 20. As the number of variables increases, the cost of checking independent conditions rises, increasing scanning overhead for all approaches. However, more variables yield fewer matches, enabling DFS to filter more events. Thus, DFS consistently achieves the lowest query latency, with about 1.4$\times$ speedup over the best baseline. Notably, with three variables, an equality condition must exist between the first and last variables, and one of them has a high selectivity under the query setting in § VI-A5. As a result, Naive-DFS has higher network overhead when pulling (attribute, window ID) pairs than in the five-variable setting.

4) Latency Evaluation on Equality/Non-equality Conditions: Firstly, we evaluate query latency under different equality conditions by fixing independent conditions and binding equality conditions to varying attributes. Fig. 21 reports the results. Since attribute a3 is more skewed than a2, an equality condition on a3 has weaker filtering power, resulting in higher latency. Secondly, we remove all equality conditions from the default query and replace some low-selectivity independent conditions with highly selective non-equality conditions on attribute a1, and then measure the query latency. Among all approaches, only Push–pull can exploit such non-equality conditions; the others cannot. As shown in Fig. 21 (right), replacing independent conditions with non-equality conditions increases variable selectivity, which sharply enlarges the number of partial matches in Push-pull and thus increases its filtering latency. For DFS, higher variable selectivity and the absence of equality conditions substantially weaken its pruning ability, causing its latency to approach that of Push-down.

5) Scalability Evaluation on Number of CNs and SNs: Fig. 22 reports query throughput scalability with respect to the number of CNs. When scaling from 1 to 3 CNs, DFS achieves the largest throughput gain, as it incurs relatively low filtering overhead and substantially reduces the volume of transmitted events, thereby lowering network bandwidth consumption. As the number of CNs increases further, rapid saturation of network bandwidth and memory resources diminishes the benefits of parallelism. When the number of CNs reaches six or more, the throughput of DFS, Push-pull, and Naive-DFS declines, as SNs maintain the intermediate state for each query, incurring high memory overhead that limits query concurrency. However, since DFS compacts SWF and releases SN memory as WJF is consumed, its throughput on 12 CNs remains higher than that of Push-down.

Fig. 23 shows the average query latency with different numbers of SNs. Increasing the number of SNs improves parallelism in data loading, leading to lower query latency. DFS consistently achieves the lowest latency across all deployment scales. Push-pull achieves lower latency than Push-down when more than three SNs are deployed. This is because the number of generated partial matches per node decreases with the increase of SNs, which reduces the matching cost. Fig. 24 shows the latency breakdown of SNs for varying numbers of SNs. The results show that the primary benefit of adding the number of SNs lies in the reduction of scanning cost.

6) Ablation Studies: We first compare three variants of DFS: ❶ SWF alone (disabling WJF in all round-trips), ❷ DFS-V1 (replacing WJF with a hash table), and ❸ DFS-V2 (replacing SWF and WJF with naive interval array and hash table, respectively). Fig. 25 reports their average query latency on the synthetic dataset. Disabling WJF increases latency by 22%, replacing WJF with a hash table increases it by about 7%, and replacing both SWF and WJF with naive solutions increases it by about 16%. These results confirm that both SWF and WJF contribute to latency reduction. Compared with Push-down, SWF alone reduces latency by 42%, and adding WJF on top of SWF provides an additional 13% improvement.

Next, we conduct a ablation study of the cost model, comparing four additional variants: always-off (never initiates

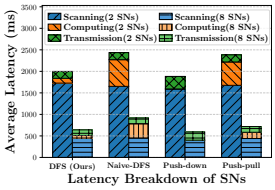


Fig. 24: Latency breakdown under varied numbers of SNs.

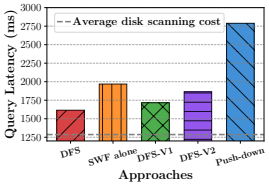


Fig. 25: Ablation studies of system components.

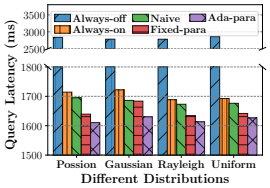


Fig. 26: Average latency vs. varied cost models.

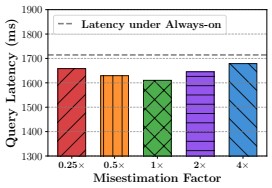


Fig. 27: Average latency under varied misestimation of $\mathcal{R}$.

new round-trips to filter), always-on (always initiates new round-trips), naive (initiates if the expected interval-shrink ratio $> 15\%$, otherwise closes), and fixed-para (uses fixed parameters initialized from warm-up queries). We run the default synthetic query under four inter-arrival distributions (Poisson, Gaussian, Rayleigh, and Uniform) and compare the query latency in Fig. 26. Across varying distributions, our adaptive cost model (ada-para) yields the lowest latency for DFS, reducing latency by about 7% compared with always-on.

Besides, we investigate the impact of misestimating the network transfer rate $\mathcal{R}$ on query latency. Since $\mathcal{R}$ in cloud environments is difficult to control, we log the cost savings observed in each round-trip and scale $\mathcal{R}$ used by the model with a multiplicative factor to simulate misestimation. As shown in Fig. 27, when the misestimation factor ranges from $0.25\times$ to $4\times$, the overall latency increases only slightly and remains lower than that of the always-on model.

VII. Discussion

For DFS, if a query pattern includes predicates with low selectivity (*i.e.*, independent or equality conditions that filter out a majority of candidate events), the resulting set of time intervals containing potential matches is substantially reduced. This allows DFS to prune more events, thereby reducing both network transmission costs and overall query latency.

However, the pruning effectiveness of DFS diminishes when it operates over very large windows or predicates with high selectivity. In such cases, its performance approaches that of the Push-down baseline. Moreover, due to the diversity and complexity of non-equality predicates (*e.g.*, `A.a1 + B.a1 > C.a1` or `A.a1 * A.a2 > B.a3`), designing a low-overhead communication strategy to exploit them for pruning is challenging. Consequently, the current DFS implementation does not leverage non-equality conditions. We consider this an important research direction and leave it for future work.

VIII. Related Work

Complex event recognition (CER). CER can be divided into two categories based on processing style: centralized and distributed. Centralized CER aims to process complex event queries in a monolithic machine. Various techniques have been proposed to enhance query efficiency in this setting, such as parallel-based techniques [51], [57]–[59], join-based techniques [7], [60], [61], sub-pattern sharing techniques [4], [62]–[64], and filter-based techniques [6], [31], [32], [52], [55], [65] (a comprehensive survey can be found in [66]). On the other hand, distributed CER [1]–[3], [28], [29] aims to divide a query into subqueries, processing them across multiple nodes. Commonly, existing distributed solutions are tailored for fully distributed architectures, which makes migrating them to cloud-native environments challenging.

Cloud-native architectures. Essentially, cloud-native architectures aim to decouple and pool computing and storage resources, enabling independent scaling [17], [18]. It has two typical implementation paradigms: disaggregated compute-storage and disaggregated compute-memory-storage [17]. In this paper, we focus on the processing of CER over disaggregated compute-storage. Note that disaggregated compute-storage encounters severe performance bottlenecks in the network [23], [25]. To alleviate this issue, push-down and caching approaches [19], [24]–[27], [67]–[70] are proposed. For CER, push-down still transmits numerous irrelevant events as it overlooks the window and dependent conditions inherent in the query. Caching approach stores partial data in the memory of the compute node to avoid fetching data from remote storage, which is commonly orthogonal to our solution.

IX. Conclusion

When CER operates in cloud-native architectures, network becomes a performance bottleneck. Our key insight is to identify shorter time intervals that contain matches and only transmit the events within those intervals. Upon this insight, we propose a lightweight DFS to identify the time interval set, which achieves a $1.2\times$ to $25\times$ query speedup compared with the state-of-the-art approaches.

Acknowledgements

Haipeng Dai is the corresponding author. This work was supported in part by the National Key R&D Program of China under Grant No. 2023YFB4502400, in part by the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China under Grant No. JYB2025XDXM901, in part by the National Natural Science Foundation of China under Grant No. 62272223, U22A2031, U24B20153, 62402212, U23A20309, 62272302, 62372296, in part by the New Generation Information Technology Innovation Project 2023 (2023IT196), in part by the Fundamental Research Funds for the Central Universities under Grant No. 2024300349 and 2025300309, in part by the Natural Science Foundation for Young Scientists of Jiangsu Province under Grant No. BK20241245, in part by the Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing University, in part by the Jiangsu High-level Innovation and Entrepreneurship (Shuangchuang) Program, and in part by the Program for Outstanding PhD Candidates of Nanjing University under Grant No. 2025A11.

AI-Generated Content Acknowledgement

We clarify that AI tools (*e.g.*, ChatGPT, Gemini, and DeepSeek) are used solely to refine grammar and wording, not to generate original ideas, technical frameworks, or experimental analysis. We assume full responsibility for the content of this work.

References

[1] S. Akili, S. Purtzel, and M. Weidlich, "INEv: In-Network Evaluation for Event Stream Processing," *Proc. ACM Manag. Data*, vol. 1, no. 1, pp. 101:1–101:26, 2023.

[2] S. Purtzel, S. Akili, and M. Weidlich, "Predicate-based Push-pull Communication for Distributed CEP," in *ACM International Conference on Distributed and Event-based Systems*, 2022, pp. 31–42.

[3] S. Akili and M. Weidlich, "MuSE Graphs for Flexible Distribution of Event Stream Processing in Networks," in *Proceedings of International Conference on Management of Data*, 2021, pp. 10–22.

[4] K. Chapnik, I. Kolchinsky, and A. Schuster, "DARLING: Data-Aware Load Shedding in Complex Event Processing Systems," *Proceedings of the VLDB Endowment*, vol. 15, no. 3, pp. 541–554, 2021.

[5] H. Zhang, Y. Diao, and N. Immerman, "On Complexity and Optimization of Expensive Queries in Complex Event Processing," in *Proceedings of International Conference on Management of Data*, 2014, pp. 217–228.

[6] B. Cadonna, J. Gamper, and M. H. Böhlen, "Efficient Event Pattern Matching with Match Windows," in *ACM SIGKDD conference on Knowledge Discovery and Data Mining*, 2012, pp. 471–479.

[7] I. Kolchinsky and A. Schuster, "Efficient Adaptive Detection of Complex Event Patterns," *Proceedings of the VLDB Endowment*, vol. 11, pp. 1346–1359, 2018.

[8] S. Song, R. Huang, Y. Gao, and J. Wang, "Why Not Match: On Explanations of Event Pattern Queries," in *Proceedings of International Conference on Management of Data*, 2021, pp. 1705–1717.

[9] S. Dharmapurikar and J. W. Lockwood, "Fast and Scalable Pattern Matching for Network Intrusion Detection Systems," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 10, pp. 1781–1792, 2006.

[10] M. Caselli, E. Zambon, and F. Kargl, "Sequence-aware Intrusion Detection in Industrial Control Systems," in *ACM Workshop on Cyber-Physical System Security*, 2015, pp. 13–24.

[11] S. Huang, E. Zhu, S. Chaudhuri, and L. Spiegelberg, "T-Rex: Optimizing Pattern Search on Time Series," *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, pp. 130:1–130:26, 2023.

[12] ISO, "ISO/IEC TR 19075-5:2016 Part 5: Row Pattern Recognition in SQL," 2016, https://www.iso.org/standard/65143.html.

[13] Oracle, "SQL for Pattern Matching," 2017, https://docs.oracle.com/database/121/DWHSG/pattern.htm.

[14] Flink, "Pattern Recognition," 2022, https://nightlies.apache.org/flink/flink-docs-release-1.20/docs/dev/table/sql/queries/match_recognize/.

[15] Snowflake, "Identifying Sequences of Rows That Match a Pattern," 2020, https://docs.snowflake.com/en/user-guide/match-recognize-intoduction.

[16] Trino, "Row pattern recognition with MATCH_RECOGNIZE," 2021, https://trino.io/blog/2021/05/19/row_pattern_matching.html.

[17] H. Dong, C. Zhang, G. Li, and H. Zhang, "Cloud-Native Databases: A Survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 36, no. 12, pp. 7772–7791, 2024.

[18] W. Cao, Y. Zhang, X. Yang, and et al, "PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers," in *Proceedings of International Conference on Management of Data*, 2021, pp. 2477–2489.

[19] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes, "Building An Elastic Query Engine on Disaggregated Storage," in *USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2020, pp. 449–462.

[20] D. Durner, V. Leis, and T. Neumann, "Exploiting Cloud Object Storage for High-Performance Analytics," *Proc. VLDB Endow.*, vol. 16, no. 11, pp. 2769–2782, 2023.

[21] C. D. Portal, "Crimes - 2001 to present," https://data.cityofchicago.org/Public-Safety/Crimes-2001-to-Present/ijzp-q8t2/about_data, 2024.

[22] X. Pang and J. Wang, "Understanding the Performance Implications of the Design Principles in Storage-Disaggregated Databases," *Proc. ACM Manag. Data*, vol. 2, no. 3, p. 180, 2024.

[23] J. Wang and Q. Zhang, "Disaggregated Database Systems," in *Companion of the 2023 International Conference on Management of Data*, 2023, pp. 37–44.

[24] Y. Yang, X. Yu, M. Serafini, A. Aboulnaga, and M. Stonebraker, "FlexpushdownDB: Rethinking Computation Pushdown for Cloud OLAP DBMSs," *The VLDB Journal*, vol. 33, no. 5, pp. 1643–1670, 2024.

[25] X. Yu, M. Youill, M. E. Woicik, A. Ghanem, M. Serafini, A. Aboulnaga, and M. Stonebraker, "PushdownDB: Accelerating a DBMS Using S3 Computation," in *IEEE International Conference on Data Engineering*. IEEE, 2020, pp. 1802–1805.

[26] Y. Yang, X. Yu, M. Serafini, A. Aboulnaga, and M. Stonebraker, "Enhancing Computation Pushdown for Cloud OLAP Databases," *CoRR*, vol. abs/2312.15405, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2312.15405

[27] J. Lu, A. Raina, A. Cidon, and M. J. Freedman, "Fusion: An Analytics Object Store Optimized for Query Pushdown," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2025, pp. 540–556.

[28] M. Akdere, U. Çetintemel, and N. Tatbul, "Plan-based Complex Event Detection Across Distributed Sources," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 66–77, 2008.

[29] S. Purtzel, S. Akili, and M. Weidlich, "Efficient Multi-query Evaluation for Distributed CEP through Predicate-based Push-pull Plans," *Inf. Syst.*, vol. 120, p. 102250, 2024.

[30] D. Petkovic, "Specification of Row Pattern Recognition in the SQL Standard and its Implementations," *Datenbank-Spektrum*, vol. 22, no. 2, pp. 163–174, 2022.

[31] E. Zhu, S. Huang, and S. Chaudhuri, "High-Performance Row Pattern Recognition Using Joins," *Proceedings of the VLDB Endowment*, vol. 16, no. 5, pp. 1181–1195, 2023.

[32] M. Körber, N. Glombiewski, and B. Seeger, "Index-Accelerated Pattern Matching in Event Stores," in *Proceedings of International Conference on Management of Data*, 2021, pp. 1023–1036.

[33] H. Wang, H. Dai, S. Chen, M. Li, R. Gu, H. Chai, J. Zheng, Z. Chen, S. Li, X. Deng, and G. Chen, "Bamboo Filters: Make Resizing Smooth and Adaptive," *IEEE/ACM Trans. Netw.*, vol. 32, no. 5, pp. 3776–3791, 2024.

[34] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[35] S. Di, D. Kondo, and F. Cappello, "Characterizing and Modeling Cloud Applications/Jobs on a Google Data Center," *J. Supercomput.*, vol. 69, no. 1, pp. 139–160, 2014.

[36] S. Tao and J. Pender, "A Stochastic Analysis of Bike-Sharing Systems," *Probability in the Engineering and Informational Sciences*, vol. 35, no. 4, p. 781–838, 2021.

[37] D. W. Osgood, "Poisson-based Regression Analysis of Aggregate Crime Rates," in *Quantitative methods in criminology*, 2017, pp. 577–599.

[38] B. Fan, D. G. Andersen, M. Kaminsky, and M. Mitzenmacher, "Cuckoo Filter: Practically Better Than Bloom," in *Proceedings of International Conference on Emerging Networking Experiments and Technologies*, 2014, pp. 75–88.

[39] H. Zhang, Y. Diao, and N. Immerman., "SASE Open Source System," 2013, https://github.com/haopeng/sase.

[40] "Esper," 2024, https://github.com/espertechinc/esper.

[41] "When Complex Event Recognition Meets Cloud-Native Architectures," 2025, https://github.com/Josehokec/CERMeetsDS.

[42] Amazon s3 object storage built to retrieve any amount of data from anywhere. [Online]. Available: https://aws.amazon.com/s3/.

[43] "Amazon Elastic Block Store," 2025, https://aws.amazon.com/ebs/.

[44] W. Zhang, E. Xu, Q. Wang, X. Zhang, Y. Gu, Z. Lu, T. Ouyang, G. Dai, W. Peng, Z. Xu *et al.*, "What's the Story in EBS Glory: Evolutions and Lessons in Building Cloud Block Store," in *22nd USENIX Conference on File and Storage Technologies*, 2024, pp. 277–291.

[45] Amazon rds db instance storage. [Online]. Available: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_Storage.html

[46] Amazon web services (aws) - atlas - mongodb docs. [Online]. Available: https://www.mongodb.com/docs/atlas/reference/amazon-aws/.

[47] A. Xie, B. Duan, F. Sun, P. Singh, and Y. Yu. Pingcap increased tidb cloud stability using amazon ebs detailed performance statistics. [Online]. Available: https://aws.amazon.com/blogs/storage/pingcap-increased-tidb-cloud-stability-using-amazon-ebs-detailed-performance-statistics/.

[48] B. Zhao, H. van der Aa, T. T. Nguyen, Q. V. H. Nguyen, and M. Weidlich, "EIRES: Efficient Integration of Remote Data in Event Stream

Processing," in *Proceedings of International Conference on Management of Data*. ACM, 2021, pp. 2128–2141.
[49] "Citi bike system data," 2025, http://www.citibikenyc.com/system-data.
[50] J. Wilkes, "Yet more Google compute cluster trace data," Google research blog, Apr. 2020, https://ai.googleblog.com/2020/04/yet-more-google-compute-cluster-trace.html.
[51] S. Akili, S. Purtzel, and M. Weidlich, "DecoPa: Query Decomposition for Parallel Complex Event Processing," *Proc. ACM Manag. Data*, vol. 2, no. 3, p. 132, 2024.
[52] A. Amir, I. Kolchinsky, and A. Schuster, "DLACEP: A Deep-Learning Based Framework for Approximate Complex Event Processing," in *Proceedings of International Conference on Management of Data*, 2022, pp. 340–354.
[53] M. Bucchi, A. Grez, A. Quintana, C. Riveros, and S. Vansummeren, "CORE: a COmplex event Recognition Engine," *Proceedings of the VLDB Endowment*, vol. 15, no. 9, pp. 1951–1964, 2022.
[54] E. Alevizos, A. Artikis, and G. Paliouras, "Complex Event Recognition with Symbolic Register Transducers," *Proc. VLDB Endow.*, vol. 17, no. 11, pp. 3165–3177, 2024.
[55] S. Liu, H. Dai, S. Song, M. Li, J. Dai, R. Gu, and G. Chen, "ACER: Accelerating Complex Event Recognition via Two-Phase Filtering under Range Bitmap-Based Indexes," in *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2024, pp. 1933–1943.
[56] Collect metrics, logs, and traces using the cloudwatch agent. [Online]. Available: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/Install-CloudWatch-Agent.html
[57] M. Yankovitch, I. Kolchinsky, and A. Schuster, "HYPERSONIC: A Hybrid Parallelization Approach for Scalable Complex Event Processing," in *Proceedings of International Conference on Management of Data*, 2022, pp. 1093–1107.
[58] I. Kolchinsky and A. Schuster, "Real-time Multi-Pattern Detection over Event Streams," in *Proceedings of International Conference on Management of Data*, 2019, pp. 589–606.
[59] S. Zhang, H. T. Vo, D. Dahlmeier, and B. He, "Multi-Query Optimization for Complex Event Processing in SAP ESP," in *IEEE International Conference on Data Engineering*, 2017, pp. 1213–1224.
[60] Y. Mei and S. Madden, "ZStream: A Cost-based Query Processor for Adaptively Detecting Composite Events," in *Proceedings of International Conference on Management of Data*, 2009, pp. 193–206.
[61] I. Kolchinsky and A. Schuster, "Join Query Optimization Techniques for Complex Event Processing Applications," *Proceedings of the VLDB Endowment*, vol. 11, p. 1332–1345, 2018.
[62] L. Ma, C. Lei, O. Poppe, and E. A. Rundensteiner, "Gloria: Graph-based sharing optimizer for event trend aggregation," in *Proceedings of International Conference on Management of Data*, 2022, pp. 1122–1135.
[63] O. Poppe, C. Lei, L. Ma, A. Rozet, and E. A. Rundensteiner, "To Share, or not to Share Online Event Trend Aggregation Over Bursty Event Streams," in *Proceedings of International Conference on Management of Data*, 2021, pp. 1452–1464.
[64] M. Ray, C. Lei, and E. A. Rundensteiner, "Scalable Pattern Sharing on Event Streams," in *Proceedings of International Conference on Management of Data*, 2016, pp. 495–510.
[65] M. Seidemann, N. Glombiewski, M. Körber, and B. Seeger, "ChronicleDB: A High-Performance Event Store," *ACM Trans. Database Syst.*, vol. 44, no. 4, pp. 13:1–13:45, 2019.
[66] N. Giatrakos, E. Alevizos, A. Artikis, A. Deligiannakis, and M. N. Garofalakis, "Complex Event Recognition in the Big Data Era: A Survey," *The VLDB Journal*, vol. 29, no. 1, pp. 313–352, 2020.
[67] V. Jalaparti, C. Douglas, M. Ghosh, A. Agrawal, A. Floratou, S. Kandula, I. Menache, J. S. Naor, and S. Rao, "Netco: Cache and I/O Management for Analytics over Disaggregated Stores," in *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*, 2018, pp. 186–198.
[68] D. Durner, B. Chandramouli, and Y. Li, "Crystal: A Unified Cache Storage System for Analytical Databases," *Proc. VLDB Endow.*, vol. 14, no. 11, pp. 2432–2444, 2021.
[69] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao, "Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases," in *Proceedings of International Conference on Management of Data*, 2017, pp. 1041–1052.
[70] Z. Ji, Z. Xie, Y. Wu, and M. Zhang, "LBSC: A Cost-Aware Caching Framework for Cloud Databases," in *Proceedings of International Conference on Data Engineering*, 2024, pp. 4911–4924.