ACER: Accelerating Complex Event Recognition via Two-Phase Filtering under Range Bitmap-Based Indexes

Shizhe Liu
shizheliu@smail.nju.edu.cn
State Key Laboratory for Novel Software Technology
Nanjing University, Nanjing, China

Haipeng Dai*
haipengdai@nju.edu.cn
State Key Laboratory for Novel Software Technology
Nanjing University, Nanjing, China

Shaoxu Song
sxsong@tsinghua.edu.cn
BNRist
Tsinghua University, Beijing, China

Meng Li
Jingsong Dai
meng@nju.edu.cn
jingsongdai@smail.nju.edu.cn
State Key Laboratory for Novel Software Technology
Nanjing University, Nanjing, China

Rong Gu
Guihai Chen
gurong@nju.edu.cn
gchen@nju.edu.cn
State Key Laboratory for Novel Software Technology
Nanjing University, Nanjing, China

ABSTRACT

Complex event recognition (CER) refers to identifying specific patterns composed of several primitive events in event stores. Since full-scanning event stores to identify primitive events holding query constraint conditions will incur costly I/O overhead, a mainstream and practical approach is using index techniques to obtain these events. However, prior index-based approaches suffer from significant I/O and sorting overhead when dealing with high predicate selectivity or long query window (common in real-world applications), which leads to high query latency. To address this issue, we propose ACER, a Range Bitmap-based index, to accelerate CER. Firstly, ACER achieves a low index space overhead by grouping the events with the same type into a cluster and compressing the cluster data, alleviating the I/O overhead of reading indexes. Secondly, ACER builds Range Bitmaps in batch (block) for queried attributes and ensures that the events of each cluster in the index block are chronologically ordered. Then, ACER can always obtain ordered query results for a specific event type through merge operations, avoiding sorting overhead. Most importantly, ACER avoids unnecessary disk access in indexes and events via two-phase filtering based on the window condition, thus alleviating the I/O overhead further. Our experiments on six real-world and synthetic datasets demonstrate that ACER reduces the query latency by up to one order of magnitude compared with SOTA techniques.

CCS CONCEPTS

• **Information systems** → **Information retrieval**.

*Haipeng Dai is the corresponding author of this paper.

KDD '24, August 25–29, 2024, Barcelona, Spain.

ACM ISBN 979-8-4007-0490-1/24/08
https://doi.org/10.1145/3637528.3671814

KEYWORDS

Complex Event Recognition, Bitmap Indexes, Filtering Algorithm

ACM Reference Format:
Shizhe Liu, Haipeng Dai, Shaoxu Song, Meng Li, Jingsong Dai, Rong Gu, and Guihai Chen. 2024. ACER: Accelerating Complex Event Recognition via Two-Phase Filtering under Range Bitmap-Based Indexes. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '24), August 25–29, 2024, Barcelona, Spain.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3637528.3671814

1 INTRODUCTION

Complex event recognition (CER) refers to identifying specific patterns composed of several primitive events in event stores. It is widely employed in event pattern mining applications like cluster monitoring [27, 47, 52], network intrusion detection [9, 15, 50], and financial services [21, 24, 45]. Notably, in 2016, the International Organization for Standardization issued new semantics in SQL to support CER [25]. Meanwhile, prominent database and data-stream systems, including Oracle, Snowflake, and Flink have already implemented CER interfaces [19, 38, 46]. To better understand CER, we give the following example.

***Example**. In the stock trade market, an investor identifies an unusual trading pattern involving two positively correlated stocks, GOOG and MSFT, as shown in Table 1. This pattern is characterized by a first increase in MSFT's opening price, and then GOOG's opening price decreases. Then, the investor submits query Q_1 (shown in Figure 1) to event stores to retrieve the historical frequency of this pattern. If the pattern is infrequent, it may suggest a potential future scenario where either the opening price of GOOG will rise or that of MSFT will fall. Based on this insight, the investor can make strategic decisions to buy or sell stocks, aiming for maximum returns.*

Table 1: Trade events in NASDAQ.

	Ticker (stock name)	Open	Volume	Date (timestamp)
e_1	MSFT	329.67	60906	2023/05/26 10:55
e_2	AAPL	175.29	100686	2023/05/26 10:56
e_3	GOOG	125.55	82648	2023/05/26 10:57
e_4	AAPL	175.37	61517	2023/05/26 10:58
e_5	MSFT	330.85	109612	2023/05/26 11:03
e_6	GOOG	125.16	37939	2023/05/26 11:06
e_7	GOOG	125.00	43324	2023/05/26 11:23

```
PATTERN SEQ(MSFT v1, GOOG v2, MSFT v3, GOOG v4)
FROM NASDAQ
USE skip-till-next-match
WHERE 326 <= v1.open <= 334 AND 120 <= v2.open <= 130 AND
      v3.open >= v1.open * 1.003 AND v4.open <= v2.open * 0.997
WITHIN 12 minutes
RETURN COUNT(*)
```

Figure 1: Query statement Q_1.

*In Figure 1, **PATTERN** clauses depict the detected event pattern, and the keyword **SEQ** is a temporal operator that indicates the sequential occurrence relationship; **FROM** clauses denote the queried event set; **USE** clauses specify the selection strategy, i.e., the skip-till-next-match strategy, which allows skipping the irrelevant events until finding the event can match previous results; **WHERE** clauses give a predicate condition set; **WITHIN** clauses specify the maximum distance between the first and last event in the pattern, and **RETURN** clauses give the target output.*

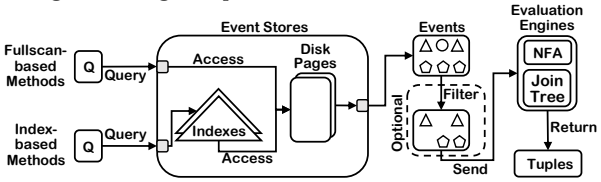


Figure 2: Fullscan-based vs. Index-based methods.

Traditional CER matching mechanisms can be mainly categorized into two types: fullscan-based [3, 14, 23, 52] and index-based [32], as shown in Figure 2: (1) Fullscan-based methods retrieve all events from event stores and feed them into evaluation engines (*e.g.*, Join Tree [36] and Non-deterministic Finite Automata (NFA) [14]) for matching. However, they suffer from high latency due to costly disk I/O overhead and the engines' filtering overhead. Although several works [8, 55] attempt to pre-filtering irrelevant events before sending events to the engines, the issue of costly disk I/O overhead persists, and the query latency still remains at a high level; (2) Index-based methods [32] utilize index techniques to alleviate the I/O overhead in reading events. They construct B+Tree indexes for event attributes (the events stored in the indexes may no longer be chronologically ordered) and greedily choose a part of query conditions with low selectivity to query events. Note that additional sort operations are necessary to ensure ordered events. However, accessing indexes and sorting events incur costly I/O and sorting overhead, especially for queries with high predicate selectivity or long query window. Consequently, prior index-based methods also suffer from high latency and may not be suitable for time-sensitive scenarios [26].

To help readers understand the execution process of the prior index-based methods, we present the following example.

***Example**. Figure 3 illustrates the execution plan of the prior index-based method [32] for query Q_1 (three B+Tree indexes has been constructed for* `ticker`, `open`, *and* `date` *columns). The method greedily selects the condition with minimum selectivity from the type condition and independent predicate condition (IPC) set (here the dependent predicate condition set is processed by the evaluation engines). Then, it uses the indexes to query these conditions. If the current condition can reduce the overall cost, it is selected. Otherwise, it is skipped, and no further conditions are selected. In our example, conditions* `326<=v1.open<=334` *and* `v4.ticker='GOOG'` *are selected (preferring* `v4` *rather than* `v2` *because* `v4` *will generate shorter replay intervals). Then, the method sorts the results, performs a join operation to create the replay intervals that potentially contain matched tuples, and queries the index of* `date` *column to find these events whose timestamps are within these replay intervals. Finally, the method sends these events to the evaluation engines for matching. We provide a detailed matching process of Join Tree evaluation engine for interested readers in Appendix A.*

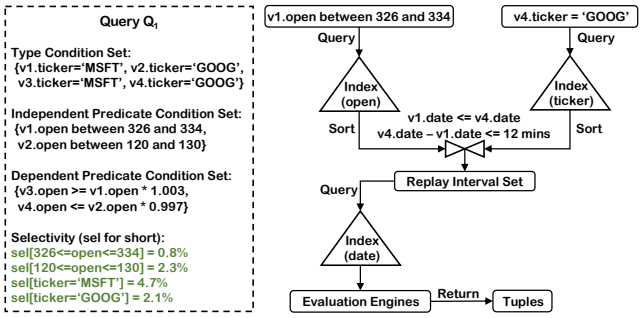
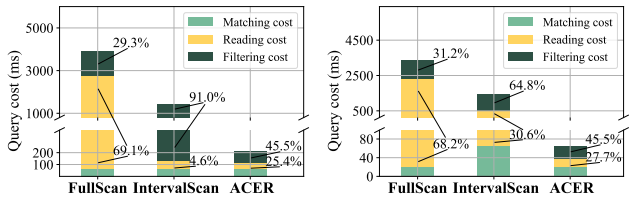


Figure 3: Execution plan of prior index-based methods.

To further give insight into the factors leading to high query latency in fullscan-based and index-based methods, we conduct experiments to estimate the query cost. Typically, the query cost can be mainly split into filtering, reading, and matching cost. Filtering cost refers to the expense of pre-filtering irrelevant events, reading cost mainly involves retrieving events from disk, and matching cost denotes the expense of matching in evaluation engines. Then, we use two queries, Q_1 and Q_2 (Q_2 is shown in Appendix B, which has higher predicate selectivity and a longer query window), to evaluate the cost of these methods on NASDAQ dataset. The results are shown in Figure 4. Here, FullScan and IntervalScan represent the state-of-the-art fullscan-based [55] and index-based methods [32], respectively. ACER represents our solution.



(a) Cost breakdown on query Q_1 **(b) Cost breakdown on query Q_2**

Figure 4: Cost breakdown of Q_1 and Q_2 on NASDAQ dataset.

Although IntervalScan has lower query latency than FullScan, it incurs significant filtering cost due to the extra overhead in index access and result sorting. Besides, it still has a high reading cost on query Q_2 because it greedily selects the conditions with minimal selectivity to generate the best execution plan, which may result in querying partial conditions. Once several conditions are not used to filter, the events obtained from IntervalScan have a high likelihood that they cannot participate in matching. In conclusion, in designing index structures to lower the query latency of CER, we face three primary challenges: (1) **alleviating the I/O overhead of reading indexes**; (2) **ensuring chronologically ordered query results without sorting**; and (3) **avoiding unnecessary disk access in indexes and events**.

We propose ACER, a Range Bitmap-based index, to address the three challenges. To address the first challenge, ACER organizes event attributes by column and groups events of the same type

into a cluster. Then, we only need to store type information once per cluster, which reduces storage redundancy. Due to storage by column, ACER can effectively compress the index size, thus alleviating I/O access overhead in indexes.

To address the second challenge, ACER constructs Range Bitmaps indexing event attributes in batch (block) and ensures that the corresponding events in each cluster are chronologically ordered. Since the output order of the bitmap is aligned with the construction order, ACER generates ordered query results for a specific event type by sequentially merging the query results of each index block, thus avoiding sorting overhead. Notably, directly using original Range Bitmaps [10] cannot ensure ordered query results without sorting when encountering out-of-order insertion and storage caused by network congestion.

Lastly, to address the third challenge, ACER queries all IPCs and type conditions because of its low index query cost. Most importantly, a two-phase filtering algorithm in ACER is proposed to avoid unnecessary disk access in indexes and events based on the window condition. The key intuition of the filtering algorithm is that: (1) if an index block's time range cannot overlap with the replay intervals, it can be skipped accessing as it cannot contain any matched tuples; (2) for the events whose timestamps are within a replay interval, if none satisfy a variable's type condition and IPCs, then these events cannot be combined into matched tuples, thus we can skip accessing them.

Overall, our key contributions are summarized as follows:

- We propose an index structure named ACER to accelerate CER. It achieves a low storage overhead by eliminating redundant type information and compressing index blocks. Besides, it can obtain chronologically ordered events that hold query constraint conditions without sorting.
- We propose a two-phase filtering algorithm that is aware of the query window condition to filter, effectively avoiding unnecessary disk access in indexes and events.
- We conduct extensive experiments to verify the performance of ACER. The experimental results show that: (1) ACER only needs around 25% storage overhead of prior index-based methods, which alleviates the costly I/O overhead of reading indexes; and (2) ACER reduces the query latency by up to one order of magnitude compared with SOTA techniques.

2 PRELIMINARIES

In this section, we first present the complex event query template we support, and then introduce a data structure, *i.e.*, bitmap used in our solution.

2.1 Complex Event Query Template

Listing 1 shows the complex event query template we support (the template is the SASE-like query style [36]). Next, we define these query components in the template sequentially.

```
PATTERN event_pattern
FROM event_set
USE selection_strategy
WHERE predicate_condition_set
WITHIN query_window
RETURN expected_output
```

Listing 1: Complex event query template.

Definition 1. *An **event pattern** provides the temporal sequential relationship of multiple primitive events. Its representation grammar is as follows:*

```
(i)   S -> PatternExpr
(ii)  PatternExpr -> SEQ(PatternExpr, PatternExpr)
(iii) PatternExpr -> AND(PatternExpr, PatternExpr)
(iv)  PatternExpr -> OR(PatternExpr, PatternExpr)
(v)   PatternExpr -> PatternExpr, PatternExpr
(vi)  PatternExpr -> typeCondition variableName
```

where SEQ, AND, *and* OR *are temporal operators.* SEQ *operator indicates multiple events occurring chronologically.* AND *operator indicates all events must occur, regardless of event occurrence order.* OR *operator indicates at least one event occurring among multiple events.*

Intuitively, for an event pattern with OR operator, we can decompose it to multiple event patterns without OR operator [4, 5, 55]. Additionally, for queries that contain other special operators such as Kleene and negation operator [36], to the best of our knowledge, they can be transformed into queries containing only SEQ and AND operators for filtering [24]. Thus, for convenience, the queries discussed in this paper only contain SEQ and AND operators.

Definition 2. *An **event set** is composed of N primitive events, each sharing a schema defined as $(Type, Attr^1, ..., Attr^d, Ts)$, where $Type$ specifies the type of an event, $Attr^i$ is the i-th attribute of events, and Ts refers to an event occurrence timestamp. Besides, the events in this set are chronologically ordered, e.g., $e_i.Ts \leq e_j.Ts$ for any $i < j$, where $e_i.Ts$ denotes the timestamp of event e_i.*

Definition 3. *A **predicate condition set** is composed of independent and dependent predicate conditions. Here, the independent predicate condition (IPC) [55] is a boolean expression that binds a single variable v_i, specifying the attribute value property of the event. The dependent predicate condition (DPC) [55] is a boolean expression that binds two variables v_i and v_j, specifying the attribute value relationship between the events.*

Definition 4. *A **selection strategy** specifies how evaluation engines select events to match the event pattern. Commonly, three match strategies are available: strict-contiguous, skip-till-next-match, and skip-till-any-match [22, 52]. The first selection strategy requires that the events matching the pattern must occur in direct succession. The second selection strategy allows the engines to skip irrelevant events until they find the event that matches previous results. The third selection strategy allows the engines to skip any events, regardless of whether they can potentially match the previous results.*

Due to the last two selection strategies have fewer matching restrictions and are more robust, they are extensively applied in diverse scenarios [5, 6, 11, 43, 54]. Thus, this paper mainly addresses the queries using the last two selection strategies.

Definition 5. *A **query window** defines the maximum distance between the first and last event in the event pattern.*

There are two types of query windows: count-based and time-based (their detailed introduction can be seen in [6]). This paper only discusses the predominant time-based query windows due to space limitations. Notably, extending our index structures to support count-based query windows is easy because it is a special case of the time-based when the event arrival rate is fixed.

Definition 6. *An **expected output** refers to a specific result format that the user expects the evaluation engines to produce, e.g., the entire matched results or the number of matched results.*

2.2 Bitmap Structures

Here we introduce the basic concepts of bitmaps and then focus on a special type called Range Bitmap adopted in our work.

Essentially, a bitmap is a binary array that can represent an integer set. For example, given two sets $S_1 = \{0, 1, 3, 7\}$ and $S_2 = \{3, 6, 7\}$, the bitmaps of two sets are represented as **10001011** and **11001000**. Using bitwise operations (*e.g.*, AND and OR), we can obtain the intersection result **10001000** and union result **11001011** between two bitmaps. Then, by sequentially reading the bit value from bitmaps, we finally obtain the intersection result $\{3, 7\}$ and union result $\{0, 1, 3, 6, 7\}$ of two sets. Note that the set we obtain from the bitmap always keeps order.

To enable bitmap structures to support efficient range query, Chan and Ioannidis [10] proposed the Base-2 Bit Sliced Range Encoded Bitmap (Range Bitmap for short). Figure 5 shows a constructed Range Bitmap for a given item set of $\{7, 1, 5, 0, 4, 5, 2, 1\}$. This Range Bitmap has three sliced bitmaps, and the values stored in each row of the Range Bitmap are the inverse code of the input values (construction details can be seen in [18]).

To identify locations where the item value is no greater than a query threshold, Range Bitmap first creates an initial bitmap $state$, with all bit values set to 1. Next, it converts the threshold into binary, denoted as t. For the i-th bit value v in t, if v is equal to 1, an OR operation is performed on $state$ and the i-th slice bitmap, and the result is updated to $state$. Otherwise, an AND operation is performed on $state$ and the i-th slice bitmap, and the result is updated to $state$. By performing the OR/AND operation from the lowest bit to the highest bit of t, we obtain a bitmap indicating which locations have item values not greater than the query threshold. An example of finding positions with item values no greater than 4 in the above constructed Range Bitmap is shown in Figure 5. Notably, Range Bitmap can be extended to support other inequality constraint queries by negation or intersection operation.

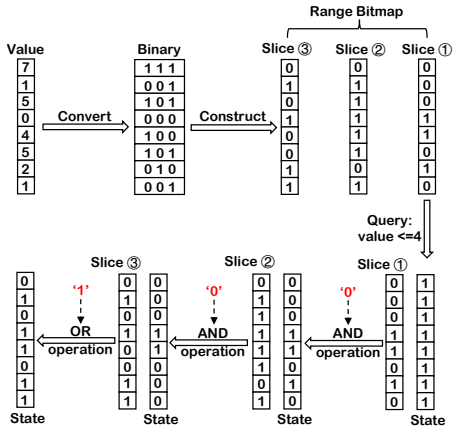


Figure 5: Construction and query of Range Bitmap.

When processing multiple IPCs for a given variable, Range Bitmap can use bitwise operations to return query results quickly. However, even if the IPC selectivity is low, Range Bitmap always needs to access all bits to determine which values satisfy the IPC, which may lead to unnecessary access in indexes. In Section 3.3, we avoid this drawback via a two-phase filtering algorithm.

3 DESIGN OF ACER

In this section, we first provide the ACER structure overview. Then, we introduce the insertion and query processes of ACER. Finally, we propose two optimization techniques to enhance the query performance of ACER.

3.1 ACER Structure Overview

ACER structure is illustrated in Figure 6. It mainly contains four components: Page, Buffer Pool, Index Block, and Synopsis Table. Remarkably, ACER cannot support indexing floating-point (FP) or string values. Fortunately, we can solve this problem by (1) scaling FP numbers to integers by multiplying by 10 to the power of s, where s is the maximum number of decimal places of these FP numbers; (2) using a string dictionary to encode string data into integers. Next, we introduce the components of ACER sequentially.

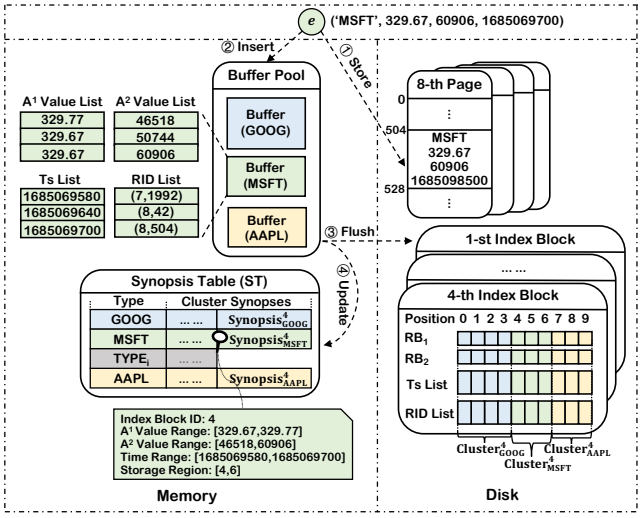


Figure 6: Structure overview and insertion example (on NASDAQ dataset) of ACER.

Page stores the entire event information persistently in row format. When an event arrives, we store it on the disk page and return the physical record identifier (RID) of this event[1].

Buffer Pool temporarily holds events awaiting indexing. It has a limited capacity and houses multiple buffers, each of which stores a part of event information such as timestamps, RIDs, and attribute values. Events within a single buffer have the same event type. When the Buffer Pool is full, each buffer will sort data based on timestamp, transfer data to an Index Block, update the Synopsis Table, and finally clear its content.

Index Block includes several Range Bitmaps [10] (RB for short) for indexing event attributes, a timestamp list (Ts List for short), and an RID list. Here, if i-th position in the RID list in an Index Block points to event e_{s_i}, then i-th position in the Ts list represents $e_{s_i}.Ts$. For j-th Range Bitmap, i-th position in this Range Bitmap

[1]Here the physical record identifier (RID) represents the physical storage location information. RID is a composite of page number and offset position.

stores the inverse code of the corresponding attribute value for the event e_{s_i}. Besides, in each Index Block, events of the same type are grouped in a continuous segment termed a **cluster**.

Synopsis Table (ST for short) logs synopsis information for each cluster across different Index Blocks. The cluster synopsis information includes Index Block ID, attribute's maximum/minimum values (note that range filters [28, 34, 48] can replace maximum/minimum values to obtain better filtering performance), time range, and storage region. The time range records this cluster's minimum and maximum event occurrence timestamp. The storage region records the position range of event storage in this cluster.

3.2 Insertion

When an event arrives, we store it on the Page and obtain its RID. Next, we extract the event's type, timestamp, and indexed attribute values, appending them to the corresponding buffer. Once the insertion process is complete, we check whether the Buffer Pool has reached its capacity. If yes, we flush the data in each buffer to a new cluster of the Index Block. When a buffer finishes the flush operation, we generate a synopsis for the newly generated cluster, add this synopsis information to the Synopsis Table, and clear the buffer. After flushing all buffers, we serialize the Index Block and write it to disk. If the Buffer Pool is not full or the Index Block has been written to disk, we return "true" to indicate a successful insertion. We omit the insertion algorithm to save space.

Example*. Figure 6 shows a running insertion example. When an event* $e = (\text{'MSFT'}, 329.67, 60906, 1685069700)$ *arrives, ACER stores it on the 8-th page and returns the RID, denoted by* $rid = (8, 504)$*. Then, ACER inserts rid, attribute values, and the timestamp of event e into the buffer 'MSFT'. At this time, the Buffer Pool reaches its capacity. Thus, ACER begins flushing the data of the buffer one by one to a new cluster of the Index Block, ensuring that events within a cluster are chronologically ordered. In our example, the buffer 'MSFT' is the second buffer to be flushed into the 4-th Index Block, and the flushed region is denoted by* $Cluster^4_{MSFT}$*. After flushing the buffer 'MSFT', ACER generates a synopsis* $Synopsis^4_{MSFT}$ *for the cluster, inserts it into the Synopsis Table, and clears the buffer. When flushing all buffers, ACER begins to construct two Range Bitmaps, denoted by* RB_1 *and* RB_2*, for the first and second attribute value lists. Finally, ACER serializes the 4-th Index Block to disk and returns "true".*

3.3 Query

Due to the complex event query process of ACER is based on individual variables, we first introduce how ACER queries the events that satisfy a single variable's type condition and IPCs (the DPCs are processed by evaluation engines). Then, we introduce how ACER processes complex event queries.

Before introducing, we give some structure declarations shown in Figure 7. `RidTimePair` is composed of the timestamp and RID of an event, which is used as an intermediate query result of a single variable (the reason why we do not directly use RID as the query result is we need timestamp to filter further). `IPC` defines an independent predicate condition (we have equivalently converted float-point or string values to long values). **`ReplayIntervalSet` (`RIS` for short) denotes a set containing several replay intervals, each defined as a time segment starting and ending at specific timestamps**. Note that `RIS` is ordered, and it includes several functions, *e.g.*, `checkOverlap`, `coveringUpdate`, and `pairFiltering`. We will explain the functionality of these functions later.

```
struct RidTimePair{
   RID rid;
   long timestamp;
};
```

```
struct IPC{
   String attrName;
   long minVal;
   long maxVal;
};
```

```
struct ReplayIntervalSet{
   long[ ] startTsList;
   long[ ] endTsList;
};
```

Figure 7: Structure declarations.

Process of a single variable query. Assume that the event pattern of a given query contains ξ variables $v_1, ..., v_\xi$. We use Θ_i^{IPC} to represent the set of IPC bound to variable v_i. Algorithm 1 illustrates the process of a single variable query.

Algorithm 1: Process of a single variable query

Input: Variable v_i and the IPC set Θ_i^{IPC} bound to it.
Output: RidTimePair set *pairs* that satisfies the type condition and IPCs bound to the variable v_i.

```
1  String type ← the event type bound to the variable v_i;
2  int [] blockIDs ← ST.getAllBlockIds(type, Θ_i^IPC);
3  regions ← ST.getAllStorageRegion(type, Θ_i^IPC);
4  RidTimePair[ ] pairs;
5  for j in range(blockIDs.length) do
6      id = blockIDs[j];
7      IndexBlock ib ← obtain the specified Index Block based on id;
8      regionBitmap ← create a region bitmap based on regions[i];
9      Bitmap ansBitmap = regionBitmap;
10     foreach ipc in Θ^i_IPC do
11         RangeBitmap rb ← obtain the attribute Range Bitmap
             from the Index Block ib based on ipc.attrName;
12         bmp = rb.rangeQuery(ipc.minVal, ipc.maxVal);
13         ansBitmap = ansBitmap & bmp;
14     curRidTimePairs ← find RID and timestamp pairs of
         corresponding positions from Ts list and RID list of Index
         Block ib using bitmap ansBitmap;   ▷ query an Index Block
15     Merge curRidTimePairs to pairs;
16 bufferPairs ← accesss BufferPool[type] to find events that satisfy
     Θ_i^IPC, and return ordered RidTimePair results;   ▷ query buffer
17 Merge bufferPairs to pairs;
18 return pairs.
```

To retrieve the RidTimePair set that satisfies the type condition and IPCs bound to a variable v_i, we first obtain the event type associated with this variable (Line 1). Next, we access the ST, find the Index Blocks that contain this event type and whose attribute value ranges overlap with all IPCs, and return their IDs (Line 2). Similarly, we get the storage region of each Index Block corresponding to the event type from the ST (Line 3). Then, we access the Index Blocks from the disk according to Index Block IDs, identify the events that satisfy IPCs, and merge them into the final result set (Line 5-15). Since the Buffer Pool also holds the events, we still need to find the events that satisfy the IPCs from the corresponding buffer and merge them into the final result set (Line 16-17). Finally, we return the final result set (Line 18). Note that the events of the same type in each Index Block are chronologically ordered. ACER can produce the ordered results by merging instead of sorting, thereby reducing the sorting overhead.

Process of complex event queries. We first give a pruning observation in matching, and then we propose a window-aware filtering via two-phase filtering algorithm to process a given query.

Empirically, for an event e_j that satisfies the type condition and IPCs of variable v_i, if it involves a final matched tuple, then other events in this matched tuple must occur in the following replay interval (denoted by δ):

$$\delta = \begin{cases} [e_j.Ts,\ e_j.Ts+w], & v_i \text{ is the head variable,} \\ [e_j.Ts-w,\ e_j.Ts], & v_i \text{ is the tail variable,} \\ [e_j.Ts-w,\ e_j.Ts+w], & \text{otherwise,} \end{cases} \quad (1)$$

where w denotes the query window, the head variable denotes the event corresponding to this variable always is the first event in matched tuples, and the tail variable denotes the event corresponding to this variable always is the last event in matched tuples. Then, we have the following pruning observation:

Observation 1. *Given a query without the* OR *operator, for the events whose timestamps are within a replay interval, if none of them can satisfy a variable's type condition and IPCs (events cannot be reused), then this replay interval does not contain any matched tuples for this query.*

Inspired by Observation 1, we propose a two-phase filtering algorithm to avoid unnecessary disk access in indexes and events. Intuitively, (1) after obtaining the RIDTimePair results of the first queried variable, we can generate a replay interval set *RIS* based on the results. Then, if an Index Block's time range cannot overlap with *RIS*, it can be skipped accessing as it cannot contain matched tuples; and (2) after obtaining the query results of all variables, we can utilize Observation 1 to remove irrelevant replay intervals from *RIS*. Then, before querying related events from a disk, we filter these events whose timestamps are not within *RIS*, which effectively avoids accessing irrelevant events from the disk. Thus, ACER achieves window-aware filtering because it utilizes the window condition to avoid unnecessary access in indexes and events.

Algorithm 2 illustrates the query procedure of ACER. Before starting the first-phase filtering (Line 4-14), we first estimate the overall selectivity of each variable (Line 1) and sort them according to selectivity (Line 2). Subsequently, we query the RidTimePair results of the variable with minimum overall selectivity and store the results in a map structure (Line 4-6). Then, we generate the replay interval set *RIS* (Line 7) based on Equation 1. Next, we query the RidTimePair results of other variables that satisfy the corresponding conditions (Line 8-14). To avoid unnecessary index access, we verify whether an Index Block's time region overlaps with *RIS* (Line 10-11). If yes, the Index Block may contain the events participating in matching, and other variables need to query the Index Block. Otherwise, other variables can skip accessing the Index Block. The `checkOverlap` function achieves the overlap check and returns a boolean list to mark these Index Blocks that cannot be skipped. Finally, we access corresponding Index Blocks according to the boolean list to obtain the query results and put the results into the map structure (Line 12-14).

In the second-phase filtering (Line 15-22), we first verify whether each replay interval in *RIS* may contain the matched tuples according to Observation 1. If a replay interval does not contain any matched tuples, we can remove it from *RIS*. The removal process

Algorithm 2: Process of complex event query in ACER

Input: A complex event query without `OR` operator.
Output: Matched tuples.

1 Estimate the overall selectivity of each variable in the query;
2 $v_{s_1}, ..., v_{s_\xi} \leftarrow$ sort all variables based on their overall selectivity;
3 $\Theta^{IPC}_{s_1} \leftarrow$ obtain the IPC set of v_{s_1};
4 $minPairs \leftarrow$ put the parameters v_{s_i} and $\Theta^{IPC}_{s_i}$ into Algorithm 1 to obtain the RidTimePair results; ▷ start the first-phase filtering
5 *Map<String, RidTimePair[]> varPairsMap*;
6 $varPairsMap.put(v_{s_i},\ minPairs)$;
7 $RIS \leftarrow$ generate the replay interval set based on $minPairs$;
8 **foreach** *variable* v_i *in* $\{v_{s_2}, ..., v_{s_\xi}\}$ **do**
9 String $type \leftarrow$ the event type bound to the variable v_i;
10 $timeRanges \leftarrow$ *ST.getAllBlockTimeRange(type,* $\Theta^{IPC}_{s_i}$*)*;
11 bool [] $overlaps \leftarrow RIS.checkOverlap(timeRanges)$;
12 int [] $blockIDs \leftarrow$ *ST.getAllBlockIds(type,* $\Theta^{IPC}_{s_i}$*, overlaps)*;
13 $curPairs \leftarrow$ query the RidTimePair results that satisfy the type condition and IPCs of variable v_i from related Index Blocks and BufferPool; ▷ same as Line 5-17 in Algorithm 1
14 $varPairsMap.put(v_i, curPairs)$;
15 $Event[]\ events$; ▷ start the second-phase filtering
16 **foreach** *variable* v_k $(v_k \neq v_{s_1})$ *in query* **do**
17 $pairs_k = varPairsMap.get(v_k)$;
18 $RIS.coveringUpdate(pairs_k)$; ▷ update replay interval set
19 **foreach** *variable* v_k *in* $\{v_1, ..., v_\xi\}$ **do**
20 $pairs_k = varPairsMap.get(v_k)$;
21 $filteredPairs_k = RIS.pairFiltering(pairs_k)$; ▷ remove the events whose timestamps are not within *RIS*
22 $varPairsMap.update(v_k, filteredPairs_k)$;
23 $events \leftarrow$ retrieve events based on all RIDs in varPairsMap;
24 **return** *EvaluationEngine.match(events).*

is achieved by `coveringUpdate` function (Line 18). Next, we remove the RidTimePair results whose timestamps are not within *RIS* using `pairFiltering` function (Line 19-22). Finally, we retrieve events from the disk using RIDs (Line 23), merge these events based on timestamp, put the events into the evaluation engine to obtain matched tuples, and then return these tuples (Line 24).

Theorem 1. *Algorithm 2 has a linear time and space complexity.*

Proof. (Sketch) Algorithm 2 obtains query results based on individual variables, and obtaining the query results of each variable has a linear time and space complexity. Then, Algorithm 2 has a linear time and space complexity. □

Example*. Suppose an Index Block only stores six events. Following the first example, when processing the query* Q_1 *on the event set in Table 1 (note that we extra add five unrelated events* e_8*-*e_{12} *to this event set), ACER first calculates the overall selectivity of each variable and identifies that variable* v1 *has the minimum selectivity. Then, ACER uses Algorithm 1 to obtain query results* $\{(rid_1, 10:55), (rid_5, 11:03)\}$ *of variable* v1*. Next, based on Equation 1, ACER generates the replay interval set* $RIS = \{[10:55, 11:07], [11:03, 11:15]\}$ *as shown in Figure 8. Then, ACER continues to query the results of other variables. Importantly, before querying an Index Block, ACER verifies whether the Index Block's time range overlaps with RIS.*

Clearly, the second Index Block cannot overlap with RIS. Thus, other variables only access the first Index Block to obtain query results. The query results of variable v1, v2, v3, *and* v4 *are* $\{e_1, e_5\}$, $\{e_3, e_6\}$, $\{e_1, e_5\}$, *and* $\{e_3, e_6\}$, *respectively. After gathering the query results of all variables, ACER starts removing replay intervals that do not contain matched tuples based on Observation 1. In this example, although each variable has events whose timestamps are within the two replay intervals, the second replay interval reuses events* e_5 *and* e_6. *Then, the second replay interval can still be removed. Finally, we only need to query the events whose timestamps are within the first replay interval, i.e.,* $\{e_1, e_3, e_5, e_6\}$, *and we send these events to the evaluation engines for matching.*

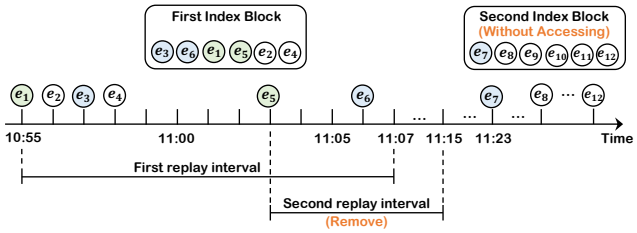


Figure 8: Query running example.

3.4 Optimization

In this subsection, we present two optimization techniques, abbreviated as **O1** and **O2**, to improve the query performance of ACER further as follows:

(**O1**) Index Block compression. ACER places data with the same category (*e.g.*, timestamps, attribute values, and RIDs) in a continuous area, which enables us to reduce Index Block size through compression effectively. Although reducing index block size can alleviate the I/O access overhead, it incurs additional computational overhead when decoding. In this paper, delta compression[2] is selected to compress the content of Index Blocks as it has a very low encoding/decoding overhead. Besides, Roaring encoding [33] is chosen to compress Range Bitmap[3].

(**O2**) Query optimization. The query process in Algorithm 2 accesses the corresponding Index Block based on individual variables. Assume that a variable v_i with minimum overall selectivity needs to access an Index Block IB, and other variables also need to access IB. Then, before accessing the Index Block IB, other variables must wait until the variable v_i queries all related Index Blocks. In the waiting period, the Index Block IB may have been removed from the cache, leading to ACER repeatedly retrieving the same Index Blocks from the disk. To avoid repeated retrieval, we allow other variables to immediately query the Index Block IB once the variable v_i completes its query in the Index Block IB. However, this optimization, O2, may not always improve query performance. The reason is that if an Index Block does not contain matched tuples, the two-phase filtering algorithm will remove the Index Block to prevent other variables from accessing it. At this time, allowing other variables to access the Index Block immediately incurs additional query overhead. Thus, we recommend enabling O2 only when the selectivity of variable v_i is greater than 0.01%.

In Section 4.2, we will verify the two optimizations **O1** and **O2**.

[2]The compressed value is equal to the real value minus the minimum value.
[3]Note that when each Index Block stores the same number of events, Roaring encoding may cause the size of Range Bitmap in each index block to be different.

4 EXPERIMENTAL EVALUATION

In this section, we first describe our experimental settings. Then, we evaluate the performance of ACER against prior index-based and fullscan-based methods on three real-world and three synthetic datasets. The source code is publicly available [2].

4.1 Experimental Settings

Datasets. We use three real-world datasets [6, 32, 43, 54]: Cluster [49], NASDAQ [1], and Crimes [42]. Cluster, NASDAQ, and Crimes datasets record job lifecycle, stock trading price and volume, and crime reports, respectively. The space overhead of an event on Cluster, NASDAQ, Crimes, and Synthetic datasets is 24, 48, 40, and 36 bytes. Table 2 shows the details of each dataset we use. Particularly, the multiple underlines mark the attributes that the query involves, which means we need to construct corresponding indexes for these attributes.

Table 2: Real-world dataset details.

Dataset	Event number	Columns
Cluster	2M	type, jobID, schedulingClass, timestamp
NASDAQ	8.7M	ticker, open, high, low, close, vol, date
Crimes	7.7M	primaryType, ID, beat, district, lat, lon, date

Besides, we generate three synthetic datasets (denoted by SD10M, SD100M, and SD1G) containing 10M, 100M, and 1G events, respectively. Each synthetic dataset follows the schema: *(string type, int a1, int a2, float a3, float a4, long timestamp)*, where each float-point value has at most two decimal places. The event arrival rate is one event per millisecond, and the event type obeys the Zipf distribution (skew=1.3, n=50). The value range of each attribute is between 0 and 1000, and they obey uniform distribution.

Baselines. We choose the following baselines: (1) **FullScan** [55] scans all events from disk and pre-filters irrelevant events before sending them to the evaluation engine; (2) **NaiveIndex** [32] constructs the B+Tree indexes for the event type and corresponding attributes, and queries all type conditions and IPCs through the indexes to obtain filtered events; and (3) **IntervalScan** [32] additionally constructs an index for the timestamp column compared with NaiveIndex. It combines the window condition and IPCs to obtain relevant events (the detailed query plan can be seen in Section 1). More implementation details can be seen in Appendix C.

Queries. On Cluster dataset, we identify patterns that jobs (with the same scheduling class) are submitted, scheduled, and evicted/finished/killed within a short period (10ms-1s) [54]. On the NASDAQ dataset, we search for price anomalies of two highly correlated stocks, where the opening price of one stock rises first, and then the opening price of the other stock falls within 15 minutes. On Crimes dataset, we search a sequence of ROBBERY, BATTERY and MOTOR VEHICLE THEFT within 30 minutes, and the places where these events occurred are relatively near [32, 55]. These criminal activities are likely to be committed by the same criminal. All real-world queries choose *skip-till-next-match* as the selection strategy.

On the three synthetic datasets, we test 5 event patterns listed in Table 3, where capital letters (A-E) represent an event type $type_i$. The event type of each pattern obeys the Zipf distribution (skew = 1.3) and $type_i \neq type_j$ when $i \neq j$. Besides, we randomly add 1-3 IPCs for each variable (the selectivity of each IPC is randomly set to 0.01-0.2) and randomly add 1-3 DPCs for each query. We

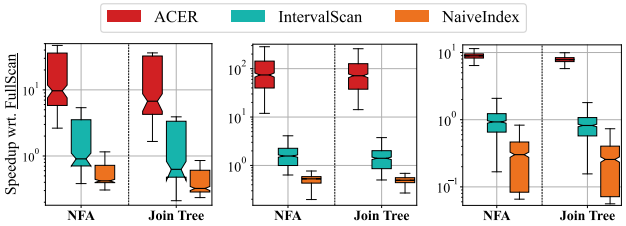
Table 3: Queried event patterns on synthetic datasets.

P1	SEQ(A a, B b, C c)
P2	SEQ(A a, B b, C c, D d, E e)
P3	SEQ(SEQ(A a, AND(B b, C c)), D d)
P4	SEQ(A a, AND(B b , C c), D d)
P5	SEQ(AND(A a, B b), AND(C c, D d))

set the query time window to 1000ms. For patterns P1 and P2, we randomly choose *skip-till-next-match* or *skip-till-any-match* as the selection strategy. However, for patterns P3, P4, and P5, we set the selection strategy to *skip-till-next-match* to reduce the number of matched tuples and avoid exceeding memory space.

4.2 Results for Real-world Datasets

Speedup performance comparison. We first generate 500 queries for each real-world dataset. Then, we evaluate their query latency under different methods, calculate the speedup of index-based methods compared with FullScan, and show the results in Figure 9.



(a) Cluster (b) NASDAQ (c) Crimes

Figure 9: Speedup of index-based methods when using NFA or Join Tree as evaluation engines on real-world datasets.

Regardless of the evaluation engines chosen, ACER consistently outperforms FullScan across the three real-world datasets, with median speedup ranging from 6.5× to 70×. Notably, ACER's query speedup exhibits significant fluctuations on Cluster and NASDAQ datasets due to heavy changes in variables' overall selectivity or the query window. In contrast, ACER shows stable query speedup on Crimes dataset, where the query window remains fixed, and the overall selectivity of each variable remains stable. Compared with FullScan, IntervalScan can reduce query latency on Cluster and NASDAQ datasets because the queries have a low predicate selectivity and a short query window. Once the query has high predicate selectivity or a long query window (e.g., on Crimes dataset), IntervalScan struggles to accelerate CER. NaiveIndex needs to query all IPCs and type conditions using B+Tree, which incurs significant filtering overhead, leading to its inability to accelerate CER.

Insertion latency comparison. We evaluate the average insertion latency of different methods on the three real-world datasets and show the results in Figure 10. Compared with FullScan, ACER increases the insertion latency by 47.5%, 103.7%, and 121.5% on Cluster, NASDAQ, and Crimes datasets (because the number of attributes need to be indexed increases), respectively. IntervalScan increases the insertion latency by 15.8×, 47.2×, and 63.6× on the same datasets. NaiveIndex has lower insertion latency than IntervalScan because NaiveIndex does not construct a B+Tree index on the timestamp column. IntervlScan and NaiveIndex have a high insertion latency because they need to insert attribute values into multiple tree-like indexes, and insertion operation may cause frequent node splitting, slowing insertion performance.

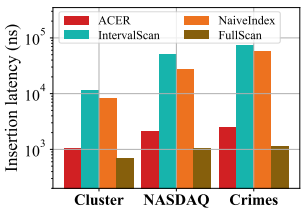


Figure 10: Insertion latency on the real-world datasets.

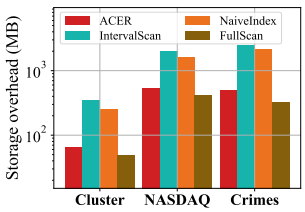


Figure 11: Storage overhead on the real-world datasets.

Storage overhead comparison. We report the total disk storage overhead of ACER and baseline methods. The results are shown in Figure 11. Compared with FullScan, ACER only increases storage space by around 35%, 28%, and 57% on the three real-world datasets, while IntervalScan increases it by around 5.8×, 3.8×, and 6.7× on the same datasets. ACER has the lowest storage overhead for two reasons. Firstly, ACER only stores the type information once per cluster. Secondly, Range Bitmap indexes, the timestamp list, and the RID list in each Index Block store delta values rather than real values (see Section 3.4), thus reducing the size of ACER. In contrast, IntervalScan needs to construct additional indexes for the timestamp and type columns compared with ACER. Even if NaiveIndex does not construct an index for the timestamp column, it still has an expensive storage overhead. The reason is that the tree-like indexes have numerous non-full nodes, and they repeatedly store the RID and timestamp of the event in the tree-like indexes, causing severe storage redundancy.

Query latency comparison with real systems. On NASDAQ dataset, we evaluate the query latency of three queries (Q_1, Q_2, and Q_3) in Flink (we set the parallelism to 1) and a commercial database (CDB for short). In addition, we employ ACER and Apache Pinot [40] (the inverted index and Range Bitmap indexes have been constructed for the type and query attribute columns) as index-filtering plugins in Flink to pre-filter events. Each query is tested 10 times across these systems, and their latency results are shown in Figure 12. Here, Filnk w. ACER denotes Flink using ACER for pre-filtering, while Filnk w. Pinot denotes Flink using Pinot for pre-filtering. ACER filters out more than 99% of events for these queries, reducing the query latency of Flink by 10×. In contrast, Pinot only filters out around 90% of events for these queries, reducing the query latency of Flink by 3× to 5×. This discrepancy arises because Pinot solely leverages type conditions for filtering (see Appendix B for details). Notably, the query latency in CDB is always over half an hour because it has a more complex selection strategy than *skip-till-next-match* (the relevant SQL query statement can be seen in Appendix B), which leads to high matching cost for CDB. Even if we rewrite the query based on [55] to obtain an improved execution plan, CDB still cannot return query results within 20 minutes.

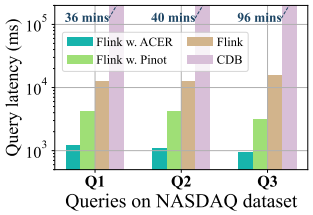


Figure 12: Query latency in the real systems.

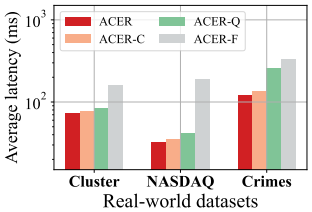
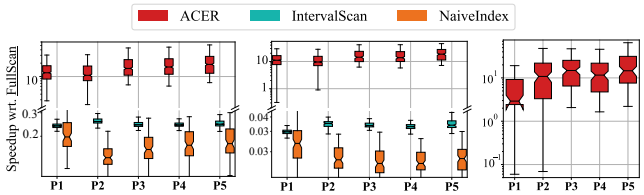


Figure 13: Ablation and optimization studies.

Ablation studies. To validate the effectiveness of optimizations O1, O2, and the two-phase filtering, we conduct ablation studies on the three real-world datasets[4]. Specifically, ACER-C denotes the disabling of optimization O1, ACER-Q denotes the disabling of optimization O2, and ACER-F denotes replacing the two-phase filtering algorithm with a bucketized filtering algorithm [55]. Then, we measure the average query latency of 500 queries on each dataset and show the results in Figure 13. Clearly, (1) without O1, the average latency on the three real-world datasets will increase by around 10% due to higher I/O access overhead in indexes; (2) without O2, the average latency on the three real-world datasets will increase by 13%, 29%, and 113%. The reason is that obtaining query results based on individual variables leads to repeated retrieval of Index Blocks; and (3) without the two-phase filtering, the query latency on the three real-world datasets will increase by 1.2×, 4.8×, and 1.8×. This is because the bucketized filtering algorithm cannot avoid unnecessary disk access in indexes and events.

4.3 Results for Synthetic Datasets

To further evaluate the scalability of ACER, we generate three synthetic datasets with more events. We generate 100 queries for each query pattern and calculate the query speedup of the index-based methods compared with FullScan on the three synthetic datasets. The results are shown in Figure 14.



(a) SD10M (b) SD100M (c) SD1G

Figure 14: Speedup of index-based methods compared with FullScan on the three synthetic datasets for the five patterns.

Compared with FullScan, ACER achieves a median speedup ranging from 2× to 10× across the three synthetic datasets. Among the index-based methods, ACER shows the best speedup performance. Notably, the 0-th percentile of ACER is lower for patterns P1 and P2 because half of the queries on P1 and P2 contain *skip-till-any-match* selection strategies, resulting in numerous partial matches. Numerous partial matches lead to high matching overhead and high copying overhead. Thus, when the selection strategy is set to *skip-till-any-match*, the matching cost accounts for a more significant proportion of the overall query cost, resulting in a low speedup performance of ACER. Neither IntervalScan nor NaiveIndex can improve query performance. The reason is that NaiveIndex filters out irrelevant events by querying all IPCs and type conditions using tree-like indexes, which have an expensive index query overhead. Although IntervalScan only queries partial conditions, the access overhead of tree-like indexes is still high when processing predicates with high selectivity. Besides, as the dataset size increases, the performance of NaiveIndex and IntervalScan deteriorates due to the increased cost of sorting operations.

[4]Unless otherwise stated, NFA will be selected as the default evaluation engine in the following experiments.

5 RELATED WORKS

Complex event recognition (CER) in event stores. CER aims to *identify specific patterns from historical data*. Typically, event stores collect events from different sources and respond to queries by retrieving relevant patterns from the stored events. This process is described as *human-active database-passive* [13]. Some implementations of event stores include SASE [23], SASE+ [16], and CORE [7]. Yet, these systems often suffer from high query latency due to costly disk I/O overhead and the filtering of the evaluation engine. To reduce high query latency, (1) Zhu *et al.* [55] introduced a bucketized pre-filtering approach that effectively reduces matching costs, but it cannot alleviate the costly I/O overhead; (2) Korber *et al.* [32] proposed an index filtering approach that reduce I/O access in events. However, this method incurs extra and expensive index accessing and sorting overhead. As a result, it still struggles with high latency, particularly for queries with high predicate selectivity or long query windows. In contrast, our proposed ACER structure efficiently alleviates costly disk I/O overhead and automatically produces ordered results without sorting, thus reducing the overall query latency.

Complex event processing (CEP) in data-stream systems. CEP focuses on *real-time monitoring and processing* of predefined patterns from *streaming data*. Concretely, the user first specifies the predetermined detection patterns to data-stream systems, and then the systems consume events to generate matched tuples and trigger planned actions. Thus, the interaction in CEP is described as *human-passive database-active* [13]. A practical data-stream system for CEP is Flink [19]. Commonly, the data-stream systems use NFA as the evaluation engine. However, NFA often encounters low processing throughput. To address this issue, various techniques have been proposed, including parallel-based techniques [31, 51, 53], join-based techniques [29, 30, 36], sub-pattern sharing techniques [11, 35, 41, 44], and filter-based techniques [6] (a comprehensive survey can be found in [20]). Notably, CEP runs in an online scenario where every incoming event is examined and stored in main memory [32]. In contrast, we aim to reduce the I/O overhead when loading the events to the main memory. Thus, the optimization techniques in CEP are orthogonal to our solution.

6 CONCLUSION

This paper proposes ACER, a Range Bitmap-based index for accelerating CER. The main ideas of ACER are reducing the size of the index block to alleviate the I/O access overhead in indexes, constructing Range Bitmaps for query attributes to produce ordered results without sorting, and utilizing a two-phase filtering algorithm to avoid unnecessary disk access. The experimental results show that ACER reduces the query latency by up to one order of magnitude on six real-world and synthetic datasets over SOTA techniques.

ACKNOWLEDGMENTS

We thank all anonymous reviewers for their valuable comments and suggestions. This work was supported in part by the National Key R&D Program of China under Grant No. 2023YFB4502400, in part by the National Natural Science Foundation of China under Grant 62272223 and 62072230, and in part by the Postgraduate Practice & Research Innovation Program of Jiangsu Province.

REFERENCES

[1] [n. d.]. Eoddata. https://www.eoddata.com.
[2] ACER. 2024. https://github.com/Josehokec/ACER4CER.
[3] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. 2008. Efficient Pattern Matching Over Event Streams. In *International Conference on Management of Data*. 147–160.
[4] Samira Akili, Steven Purtzel, and Matthias Weidlich. 2023. INEv: In-Network Evaluation for Event Stream Processing. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.
[5] Samira Akili and Matthias Weidlich. 2021. MuSE Graphs for Flexible Distribution of Event Stream Processing in Networks. In *International Conference on Management of Data*. 10–22.
[6] Adar Amir, Ilya Kolchinsky, and Assaf Schuster. 2022. DLACEP: A Deep-Learning Based Framework for Approximate Complex Event Processing. In *International Conference on Management of Data*. 340–354.
[7] Marco Bucchi, Alejandro Grez, Andrés Quintana, Cristian Riveros, and Stijn Vansummeren. 2022. CORE: a COmplex event Recognition Engine. *Proceedings of the VLDB Endowment* 15, 9 (2022), 1951–1964.
[8] Bruno Cadonna, Johann Gamper, and Michael H. Böhlen. 2012. Efficient Event Pattern Matching with Match Windows. In *Proceedings of ACM SIGKDD conference on Knowledge Discovery and Data Mining*. 471–479.
[9] Marco Caselli, Emmanuele Zambon, and Frank Kargl. 2015. Sequence-aware Intrusion Detection in Industrial Control Systems. In *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security*. 13–24.
[10] Chee Yong Chan and Yannis E. Ioannidis. 1998. Bitmap Index Design and Evaluation. In *International Conference on Management of Data*. 355–366.
[11] 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.
[12] ClickHouse. 2016. https://clickhouse.com.
[13] Gianpaolo Cugola and Alessandro Margara. 2012. Processing Flows of Information: From Data Stream to Complex Event Processing. *Comput. Surveys* 44, 3 (2012), 1–62.
[14] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. 2007. Cayuga: A General Purpose Event Monitoring System. In *International Conference on Innovative Data Systems Research*. 412–422.
[15] Sarang Dharmapurikar and John W Lockwood. 2006. Fast and Scalable Pattern Matching for Network Intrusion Detection Systems. *IEEE Journal on Selected Areas in Communications* 24, 10 (2006), 1781–1792.
[16] Yanlei Diao, Neil Immerman, and Daniel Gyllstrom. 2007. *Sase+: An Agile Language for Kleene Closure over Event Streams*. Technical Report. University of Massachusetts.
[17] Daniel Lemire, et al. 2013. Roaring Bitmap. https://github.com/RoaringBitmap/RoaringBitmap.
[18] FeatureBase. 2022. Using Bitmaps to Perform Range Queries. https://www.featurebase.com/blog/range-encoded-bitmaps.
[19] Apache Flink. [n. d.]. https://flink.apache.org/.
[20] Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, and Minos N. Garofalakis. 2020. Complex Event Recognition in the Big Data Era: A Survey. *The VLDB Journal* 29, 1 (2020), 313–352.
[21] Koosha Golmohammadi and Osmar R. Zaïane. 2012. Data Mining Applications for Fraud Detection in Securities Market. In *European Intelligence and Security Informatics Conference*. 107–114.
[22] Alejandro Grez, Cristian Riveros, Martín Ugarte, and Stijn Vansummeren. 2021. A Formal Framework for Complex Event Recognition. *ACM Transactions on Database Systems* 46, 4 (2021), 16:1–16:49.
[23] Daniel Gyllstrom, Eugene Wu, Hee-Jin Chae, Yanlei Diao, Patrick Stahlberg, and Gordon Anderson. 2007. SASE: Complex Event Processing over Streams (Demo). In *International Conference on Innovative Data Systems Research*. 407–411.
[24] 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), 130:1–130:26.
[25] ISO. 2021. ISO/IEC 19075-5:2021 Information technology — Guidance for the use of database language SQL — Part 5: Row pattern recognition. https://www.iso.org/standard/78936.html.
[26] Alejandro Jaimes and Joel Tetreault. 2021. Real-time Event Detection for Emergency Response Tutorial. In *Proceedings of ACM SIGKDD conference on Knowledge Discovery and Data Mining*. 4042–4043.
[27] Sarah Kleest-Meißner, Rebecca Sattler, Markus L. Schmid, Nicole Schweikardt, and Matthias Weidlich. 2022. Discovering Event Queries from Traces: Laying Foundations for Subsequence-Queries with Wildcards and Gap-Size Constraints. In *International Conference on Database Theory*. 18:1–18:21.
[28] Eric R. Knorr, Baptiste Lemaire, Andrew Lim, Siqiang Luo, Huanchen Zhang, Stratos Idreos, and Michael Mitzenmacher. 2022. Proteus: A Self-Designing Range Filter. In *International Conference on Management of Data*. 1670–1684.
[29] Ilya Kolchinsky and Assaf Schuster. 2018. Efficient Adaptive Detection of Complex Event Patterns. *Proceedings of the VLDB Endowment* 11 (2018), 1346–1359.
[30] Ilya Kolchinsky and Assaf Schuster. 2018. Join Query Optimization Techniques for Complex Event Processing Applications. *Proceedings of the VLDB Endowment* 11 (2018), 1332–1345.
[31] Ilya Kolchinsky and Assaf Schuster. 2019. Real-time Multi-Pattern Detection over Event Streams. In *International Conference on Management of Data*. 589–606.
[32] Michael Körber, Nikolaus Glombiewski, and Bernhard Seeger. 2021. Index-Accelerated Pattern Matching in Event Stores. In *International Conference on Management of Data*. 1023–1036.
[33] Daniel Lemire, Owen Kaser, Nathan Kurz, Luca Deri, Chris O'Hara, François Saint-Jacques, and Gregory Ssi Yan Kai. 2018. Roaring Bitmaps: Implementation of An Optimized Software Library. *Software: Practice and Experience* 48, 4 (2018), 867–895.
[34] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores. In *International Conference on Management of Data*. 2071–2086.
[35] Lei Ma, Chuan Lei, Olga Poppe, and Elke A. Rundensteiner. 2022. Gloria: Graph-based Sharing Optimizer for Event Trend Aggregation. In *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 *International Conference on Management of Data*. 193–206.
[37] Dave Moten. 2019. bplustree. https://github.com/davidmoten/bplustree.
[38] Oracle. 2017. SQL for Pattern Matching. https://docs.oracle.com/database/121/DWHSG/pattern.htm.
[39] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.
[40] Apache Pinot. 2019. https://pinot.apache.org/.
[41] 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 *International Conference on Management of Data*. 1452–1464.
[42] Chicago Data Portal. [n. d.]. Crimes - 2001 to present. https://data.cityofchicago.org/stories/s/5cd6-ry5g.
[43] Steven Purtzel, Samira Akili, and Matthias Weidlich. 2022. Predicate-based Push-Pull Communication for Distributed CEP. In *ACM International Conference on Distributed and Event-based Systems*. 31–42.
[44] Medhabi Ray, Chuan Lei, and Elke A Rundensteiner. 2016. Scalable Pattern Sharing on Event Streams. In *International Conference on Management of Data*. 495–510.
[45] Marc Seidemann, Nikolaus Glombiewski, Michael Körber, and Bernhard Seeger. 2019. ChronicleDB: A High-Performance Event Store. *ACM Transactions on Database Systems* 44, 4 (2019), 13:1–13:45.
[46] Snowflake. 2020. Identifying Sequences of Rows That Match a Pattern. https://docs.snowflake.com/en/user-guide/match-recognize-introduction.
[47] Shaoxu Song, Ruihong Huang, and Yu Gao andJianmin Wang. 2021. Why Not Match: On Explanations of Event Pattern Queries. In *International Conference on Management of Data*. 1705–1717.
[48] Ziwei Wang, Zheng Zhong, Jiarui Guo, Yuhan Wu, Haoyu Li, Tong Yang, Yaofeng Tu, Huanchen Zhang, and Bin Cui. 2023. REncoder: A Space-Time Efficient Range Filter with Local Encoder. In *IEEE International Conference on Data Engineering*. 2036–2049.
[49] John Wilkes. 2020. Yet more Google compute cluster trace data. Google research blog. Posted at https://ai.googleblog.com/2020/04/yet-more-google-compute-cluster-trace.html.
[50] Lih-Chyau Wuu, Chi-Hsiang Hung, and Sout-Fong Chen. 2007. Building Intrusion Pattern Miner for Snort Network Intrusion Detection System. *Journal of Systems and Software* 80, 10 (2007), 1699–1715.
[51] Maor Yankovitch, Ilya Kolchinsky, and Assaf Schuster. 2022. HYPERSONIC: A Hybrid Parallelization Approach for Scalable Complex Event Processing. In *International Conference on Management of Data*. 1093–1107.
[52] Haopeng Zhang, Yanlei Diao, and Neil Immerman. 2014. On Complexity and Optimization of Expensive Queries in Complex Event Processing. In *International Conference on Management of Data*. 217–228.
[53] Shuhao Zhang, Hoang Tam Vo, Daniel Dahlmeier, and Bingsheng He. 2017. Multi-Query Optimization for Complex Event Processing in SAP ESP. In *IEEE International Conference on Data Engineering*. 1213–1224.
[54] Bo Zhao, Han van der Aa, Thanh Tam Nguyen, Quoc Viet Hung Nguyen, and Matthias Weidlich. 2021. EIRES: Efficient Integration of Remote Data in Event Stream Processing. In *International Conference on Management of Data*. 2128–2141.
[55] 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.

TECHNICAL APPENDIX

A MATCHING PROCESS OF JOIN TREE

Example. *Figure 15 shows the built Join Tree for query Q_1. This tree has four leaf nodes and three internal nodes. Each leaf node in the query tree is associated with a variable, a matching buffer, a type condition, and the IPCs, while each internal node is associated with a temporal operator, a matching buffer, and join conditions. Leaf nodes' matching buffers store primitive events, while internal nodes' matching buffers store partial matches. Note that the root node's matching buffer stores the matched tuples.*

When receiving the events in Table 1, each leaf node checks if an event satisfies the type condition and IPCs bound to this node. If yes, the event is stored in the corresponding leaf node's matching buffer. In this example, node ① has the conditions `v1.ticker='MSFT'` *and* `326<=v1.open<=334`*. As a result, the matching buffer of node ① stores the primitive event sets $\{e_1, e_5\}$. Similarly, the matching buffers of leaf nodes ②, ③, and ④ stores the events sets $\{e_3, e_6, e_7\}$, $\{e_1, e_5\}$, and $\{e_3, e_6, e_7\}$, respectively. Once a leaf node buffers a new event, it triggers the join operation of its father node. The join operation combines the child nodes' matching buffers to generate new matches based on temporal operator semantics, query window condition, DPCs, and selection strategy. The join conditions of node ⑤ are* `v1.date<=v2.date, v2.date-v1.date<=12 minutes`*, and skip-till-next-match strategy. Thus, its join results are $\{(e_1, e_3), (e_5, e_6)\}$ (note that the result (e_1, e_6) is removed by skip-till-next-match strategy). Similarly, the join results of root node ⑦ are $\{(e_1, e_3, e_5, e_6)\}$, representing final matched tuples.*

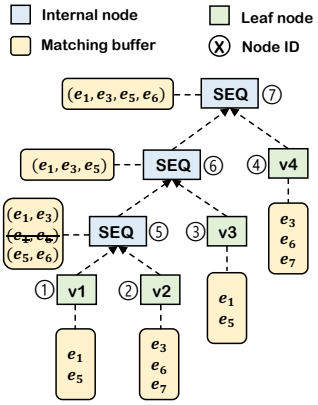


Figure 15: Join Tree for query Q_1.

B EXAMPLE QUERIES ON NASDAQ

Listing 2 and Listing 3 show two complex event queries Q_2 and Q_3 used in our experimental evaluation.

```
PATTERN SEQ(NVDA v1, INTC v2, NVDA v3, INTC v4)
FROM NASDAQ
USE skip-till-next-match
WHERE 403<=v1.open<=423 AND 23<=v2.open<=43
  AND v3.open>=v1.open*1.007
  AND v4.open<=v2.open*0.993
WITHIN 15 minutes
RETURN COUNT(*)
```

Listing 2: Query statement Q_2.

```
PATTERN SEQ(AMZN v1, BABA v2, AMZN v3, BABA v4)
FROM NASDAQ
USE skip-till-next-match
WHERE 100<=v1.open<=120 AND 90<=v2.open<=110
  AND v3.open>=v1.open*1.007
  AND v4.open<=v2.open*0.993
WITHIN 15 minutes
RETURN COUNT(*)
```

Listing 3: Query statement Q_3.

Listing 4 shows a SQL statement Q_4 corresponding to the complex event query Q_1 (note that the two queries are not equivalent due to inconsistent selection strategies). Since the type conditions of variables v1 and v3 are the same, Q_4 needs to union their predicate conditions to retrieve related events, which leads to it only filtering the events based on type conditions. In contrast, the query process in ACER is based on individual variables, and it utilizes the window condition and IPCs to filter events, which can avoid unnecessary disk access and achieve a better filtering performance. Thus, even though advanced database systems, such as Apache Pinot [40] and ClickHouse [12], have implemented Range Bitmap indexes to handle range queries, their filtering performance is poor when processing complex event queries due to the lack of window-aware filtering.

```
WITH events AS(
  SELECT ticker,open,ts FROM NASDAQ
  WHERE ticker='MSFT' OR ticker='GOOG'
  ORDER BY ts
)SELECT COUNT(*) FROM events MATCH_RECOGNIZE(
  MEASURES V1.ts AS D1,V2.ts AS D2,V3.ts AS D2,V4.
      ts AS D4
  ONE ROW PER MATCH
  AFTER MATCH SKIP TO NEXT ROW
  PATTERN (V1 Z* V2 Z* V3 Z* V4)
  DEFINE
    V1 AS V1.ticker='MSFT'
      AND V1.open BETWEEN 326 AND 334,
    V2 AS V2.ticker='GOOG'
      AND V2.open BETWEEN 120 AND 130
      AND V2.ts-V1.ts<=INTERVAL '12' MINUTE,
    V3 AS V3.ticker='MSFT'
      AND V3.open>=V1.open * 1.003
      AND V3.ts-V1.ts<=INTERVAL '12' MINUTE,
    V4 AS V4.ticker='GOOG'
      AND V4.open<=V2.open * 0.997
      AND V4.ts-V1.ts<=INTERVAL '12' MINUTE);
```

Listing 4: SQL statement Q_4.

C IMPLEMENTATION DETAILS

Environment. We conduct all experiments on a Windows 11 PC with Intel(R) Core(TM) i9-10900K CPU @ 3.70GHz and 32GB memory at 3200-MHz. We implement all methods in Java and set the `Xmx` parameter in JVM to 4GB (`Xmx` specifies the maximum memory allocation pool for a JVM).

Implementation. We use the open-source B+Tree [37] and Roaring Bitmap [17] codes as the tree-like index and bitmap module and adopt their default parameter settings. Note that, for IntervalScan method, we use B+Tree rather than LSM-Tree [39] to index the timestamp column because B+Tree has better query performance. The disk page size in all methods is set to 8KB, and the Buffer Pool capacity of ACER is set to 42K.