Information Retrieval Journal (2023) 26:4
https://doi.org/10.1007/s10791-023-09423-4

Temporal information retrieval using bitwise operators

Prasanna Koirala[1] · Ramazan Aygun[2] · Tathagata Mukherjee[3] · Haeyong Chung[3]

Received: 22 June 2023 / Accepted: 6 September 2023

Abstract
The plethora of available and stored temporal data necessitated the development of effective algorithms for information retrieval. The previous research on temporal information retrieval predominantly focused on the correctness of the retrieval results and supported wider types of temporal operators for retrieval. Many of these algorithmic approaches are based on high-level data structures and libraries supported by high-level programming languages, thus limiting the running time performance of these approaches. In this paper, we develop querying and information retrieval for temporal queries based on Allen's interval algebra that provides a calculus for temporal reasoning by defining thirteen basic relations between two intervals. To increase the retrieval performance, we propose using bitmaps and bitwise operations to identify all of Allen's thirteen relations between any two events across the entirety of the data where events are represented as bitmaps. The indexes in the bitmap represent various time instances in the data, and the values 1 and 0 correspond to the presence and absence of an event. Using bitwise operators such as AND, OR, and bit-shifts, in our compressed representation of the events, we establish expressions for each of Allen's relations. Our experiments show that, for two events with roughly 5×10^6 intervals in each, the bitwise operation-based methods are almost 42 times faster than conventional interval-based linear lookups and almost 21 times faster than conventional pattern-finding parallel techniques inherently available.

Keywords Temporal querying · Allen's relations · Bitwise operations

✉ Ramazan Aygun
raygun@kennesaw.edu

Prasanna Koirala
pkoiralap@gmail.com

Tathagata Mukherjee
tm0130@uah.edu

Haeyong Chung
hc0021@uah.edu

[1] Vanderbilt University, Nashville, TN 37235-0001, USA

[2] Department of Computer Science, Kennesaw State University, Marietta, GA 30060, USA

[3] Department of Computer Science, University of Alabama in Huntsville, Huntsville, AL 35899, USA

Published online: 23 September 2023

1 Introduction

In the era of Big Data, real-time retrieval of relevant information with precision is one of the key technical issues of having a huge volume of data such as images (Duan et al., 2015). Temporal data analysis has various applications such as information retrieval (Shrestha et al., 2019a), querying video data (Bettaiah and Aygun, 2015; Aygun and Bettaiah, 2017), finding temporal patterns (CoVID-19 Rogers et al., 2022a), cheat detection (Rogers et al., 2022b, 2023a), and temporal deduplication (Rogers et al., 2023b). A number of querying techniques, such as keyword-based, pattern-based, and natural language-based techniques, are available for information retrieval (Baeza-Yates and Ribeiro-Neto, 1999). Keyword-based queries are popular as they are intuitive and easy to process (Baeza-Yates and Ribeiro-Neto, 1999; Google, 2021). However, keyword-based queries are not suitable for obtaining temporal relations from the data. For example, the user might remember a particular sequence of events in a movie and might want to use the same information for querying. Hence, a more specific and concise form of querying is required.

Two common ways of processing temporal data include i) mapping the data to a graph and applying temporal operators and ii) representing events as intervals on a timeline. Allen's interval algebra (Allen, 1983) is a calculus for temporal reasoning that was introduced by James F. Allen in 1983, and timeline-based approaches can easily benefit from this algebra. The calculus defines thirteen basic relations between any two intervals (Fig. 1). Here, an interval can be defined as a period of time for which an event occurs. An event can then also be defined as a set of such intervals. Allen's algebra is profoundly used in various fields including but not limited to information retrieval (Shrestha et al., 2019b), pattern detection (Li et al., 2011), spatial reasoning (Renz, 2001), task scheduling (Mudrova and Hawes, 2015), and smart home management (Chuckravanen et al., 2017). While indexing data based on key values enables retrieval of specific instances as quickly as possible, such indexing methods are not typically efficient for range queries or complex queries. It is essential that the indexing structure is versatile enough to support a variety of temporal querying operators. Otherwise, temporal operators must be designed in a way that they can leverage the indexing structure. Moreover, traditional indexing structures may not leverage the underlying hardware configurations if the temporal query operators are not designed according to low level operations. A proper indexing structure that leverages the low level operations by bridging the gap between design patterns and hardware architecture could alleviate temporal querying.

Gap between hardware architectures, software design patterns, and algorithmic approaches. For efficient execution of temporal queries, their implementation should consider how the proposed querying method could leverage underlying hardware architectures. Nevertheless, there is a gap between hardware architectures and software design patterns that can worsen the performance of systems. Hence, it is usually recommended to utilize native libraries and data structures rather than implementing custom data structures unless needed since these native libraries and operations are already optimized for the underlying hardware system. However, this does not reduce the gap between the algorithmic approach between the implementation level and the hardware level.

Bitmap indexing to reduce the gap. To address this problem, we propose developing *bitmap indexing* to improve the performance of temporal querying using a bitmap type of indexing. The indexes in the bitmap represent various time instances in the data, and the

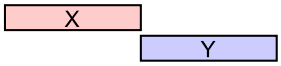
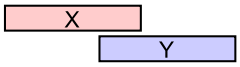
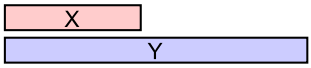
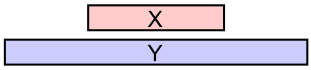
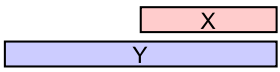
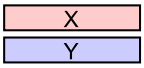
Relation	Visual Representation	Interpretation
X < Y X > Y		X precedes Y Y is preceded by X
X m Y Y mi X		X meets Y Y is met by X
X o Y Y oi X		X overlaps Y Y is overlapped by X
X s Y Y si X		X starts Y Y is started by X
X d Y Y di X		X during Y Y contains X
X f Y Y fi X		X finishes Y Y is finished by X
X = Y		X equals Y

Fig. 1 Allen's interval algebra

values 1 and 0 correspond to the presence and absence of an event respectively. Bitmap indexing has been touted as a promising approach for processing complex ad-hoc queries in read-mostly environments (Chan et al., 1998). In addition, bitmap-based operations can be heavily parallelized (Sinha et al., 2006), resulting in efficient processing of large datasets. Furthermore, bitmaps can also represent temporal data (Shrestha et al., 2019b) for information retrieval purposes (mostly read-only). Hence, leveraging bitmaps for the extraction of events and especially for extracting the temporal relations between these events in temporal data can be immensely beneficial.

In this paper, we propose an unorthodox method for temporal information retrieval using bitwise operations based on all thirteen Allen's temporal relationships using bitmap type of indexing. After comparing the performance of our method with the traditional way of searching these relationships using available libraries, video data is considered as a sample application of temporal querying in this paper with faces in videos as events. The events are pair-wise compressed so that the continuous absence or presence of these events is compressed into a single point. The compression technique enables to look for similar patterns across multiple events. Finally, we apply various bitwise operations to these compressed event pairs to obtain Allen's relations. Using bitwise operations such as AND, OR,

and shift operations in our compressed representation of the events, we establish expressions for each of Allen's relations. Our experiments show that, for two events with roughly 5×10^6 intervals in each, the bitwise operations are almost 42 times faster than conventional interval-based linear lookups and almost 21 times faster than pattern-finding parallel techniques inherently available. This also allows deployment in low-level systems, such as embedded systems, to execute queries at a minimal cost and efficiently.

Our contributions can be summarized as follows:

- Representing all thirteen Allen's interval relationships using bitwise operators and
- Improved performance of temporal querying using bitwise operators.

This paper is organized as follows. The following section discusses the current state of the art regarding Allen's relations and some background on temporal bitmaps, temporal indexing, and event-matrix. Section 3 covers compression and Allen's relations using bitwise operations on the compressed form. Section 4 provides how we analyze the correctness of results. Section 5 explains the experimental results that provide the performance on searching Allen's relations. Finally, the last section concludes our paper.

2 Related work

An event is defined as the presence of an entity or occurrence in data. For instance, detecting people's faces in a video or a person's voice in an audio recording can be considered as sample events. Since events have a duration, they are typically defined as sets of intervals, where an interval i is represented as $(s(i), e(i))$ with $s(i)$ and $e(i)$ being the start and end of the interval, respectively. For example, the event $(10,\ 20)$ starts at time 10 and ends at time 20. Many studies rely on this standard representation of events as the foundation of their research (Allen, 1983; Georgala et al., 2016; Nebel and Bürckert, 1995; Patel et al., 2008; Li et al., 2011). As an event is a set of intervals, multiple interval pairs must be considered from events to identify Allen's relations between any two events. One method to accomplish this is by performing an exhaustive comparison between intervals of the events. For two events e and t, the exhaustive comparison technique compares every interval in event e with every interval in t, resulting in a time complexity of $O(|e| \times |t|)$, where $|e|$ and $|t|$

Fig. 2 Bitmap index table for appearance of actors in movies where 1 indicates the appearance

	Movie1	Movie2	Movie3
Actor1	1	1	0
Actor2	0	0	1
Actor3	0	1	1

represent the number of intervals in events e and t, respectively. Alternatively, it can be represented with a time complexity of $O(n^2)$, where n is the maximum value of $|e|$ and $|t|$.

Bitmap indexing is a method that employs bitmaps to represent data efficiently for querying and processing purposes. An example of this can be seen in Fig. 2, which displays a sample bitmap index for actor/actress-movie appearances. However, the sparsity of the bitmap table increases with the size of the data as it uses one bit for each distinct value. Consequently, encoding and compression techniques (Stockinger and Wu, 2007; Chan et al., 1998) are utilized to decrease the size of the bitmaps, as sparse tables require more storage. Temporal bitmap indexing is a specialized form of bitmap indexing, where columns represent different time instances in chronological order and rows represent various events that occur in the data. The final output of temporal bitmap indexing is referred to as an event-time matrix or simply an event-matrix in this paper, as shown in Fig. 3. Recently, Shrestha et al. (2019b, 2019a) proposed representing events using bitmaps or even matrix representations (Zhang and Zhang, 1999), as depicted in Fig. 3. For this paper, we consider this bitmap representation.

Extracting events (such as objects and people) from temporal data (such as videos or audio) presents a significant challenge. Nevertheless, past research studies have attempted to tackle this problem (Nadimi and Bhanu, 2004; Kang et al., 2016; Wang et al., 2014). For instance, Nadimi and Bhanu (2004) detect moving objects in a video by utilizing spatio-temporal albedo tests and dichromatic reflection models. In contrast, Kang et al. (2016) propose a framework to identify objects in video data using Convolution Neural Networks (CNNs). Similarly, Wang et al. (2014) use Support Vector Machines (SVM) and Dynamic Programming (DP) to identify human actions such as walking, running, or performing a pull-up. In this paper, we only consider the faces of individuals in video data as events, and we utilize an open-source Python library called Face Recognition (Geitgey, 2018) to extract faces from the videos. We should note that the expressions provided to realize Allen's temporal relations are generalizable and not restricted to faces in videos.

Georgala et al. (2016) demonstrated that Allen's temporal relations can be represented using only eight atomic relations. They proposed an algorithm calculating Allen's relations between any two events in $O(nlogn)$ time complexity. Still, the algorithm is designed to work with the conventional representation of events. Papadias et al. (2001) proposed using binary encoding of a temporal object with respect to an interval that could be split for

Fig. 3 Event matrix for events occurring at specific time instances

	T1	T2	T3
E1	1	0	0
E2	1	0	0
E3	1	0	1
E4	0	1	1

further intervals to increase resolution leading to additional relationships. An interval splits time into 5 segments including the start and end point of the interval. Their method does not leverage bitwise binary encoding, but they rather focus on the window of bits to process. Wattamwar and Ghosh (2008) extend this logic for fuzzy relationships. Zhang and Zhang (1999) took a different approach by representing an interval with five values and all possible relationships between two intervals as a 5×5 matrix. However, since multiple intervals represent an event, calculating all relations between any two events across the entire dataset is still computationally intensive. Shrestha et al. (2019b, 2019a) introduced bitwise operations-based video querying and identified four basic relationships between events: co-appearance, next-appearance, prior appearance, and eventual appearance, which they used to define Allen's thirteen relations. However, those four relations do not fully capture all thirteen relations, and the approach does not account for cases when the same event appears multiple times in the temporal data. For example, the same event can co-exist with an event and precede another type of the same event at a later time in real-world temporal data. Naik et al. (2008, 2012) developed a semantic sequence state graph to respond queries based on the sequential ordering of events. On the other hand, Jain and Aygun (2008, 2009) use SQL pattern matching to retrieve video clips for temporal ordering queries.

Various techniques of bitmap compression have been discussed in the literature (Chen et al. (2015)). However, these techniques do not take into consideration the possible temporal relationship between other bitmap values. For our implementation, we use a compression technique that performs pairwise compression of events.

3 Method

Initially, the temporal data is analyzed to identify events, which is then utilized to create an event matrix. This event matrix serves the purpose of tracking whether a particular event is transpiring at a given moment or not. Next, the event matrix is subjected to pairwise compression of events and saved in the cache to be used later. Finally, the compressed representation of the events is subjected to bitwise operations to derive Allen's interval relationships. Assume that video V is represented as a sequence of n frames, $V =< f_1, f_2, \ldots, f_n >$. An event e occurring in a frame is represented as $e(f)$, with a value of 1 denoting its presence and a value of 0 indicating its absence. Let F_i represent the event occurrence vector for all events in f_i, where F_i^j denotes the occurrence of event e_j in f_i. Similarly, E_j denotes the vector (array) for the presence or absence of event e_j in each frame. In the event matrix, E corresponds to rows, whereas F represents columns. Moreover, our method utilizes aligned patterns in two different event sequences E_i and E_j, and uses p_i and p_j to denote patterns for events e_i and e_j respectively.

3.1 Compression algorithm

Our focus is on monitoring the occurrence of events, which entails detecting changes in the bit values of the event arrays (i.e., rows of the event matrix). Specifically, we are interested in the transition from presence to absence or vice versa of any given event. Our algorithm

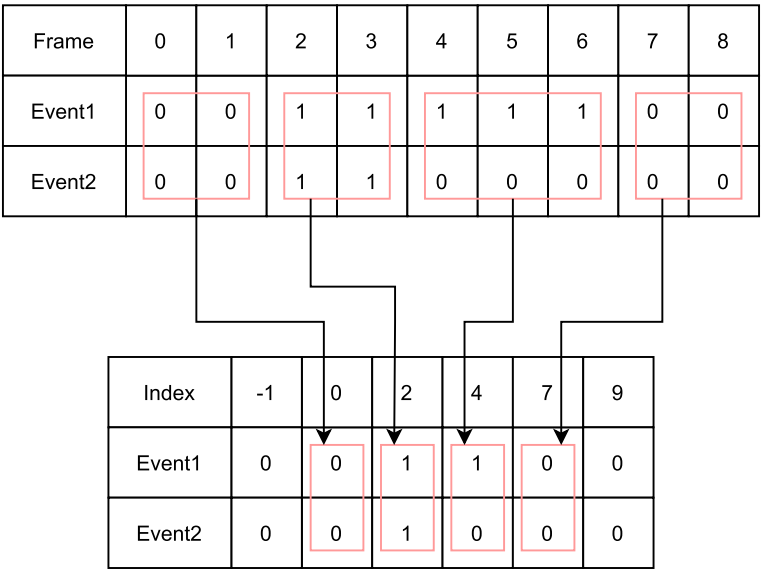


Fig. 4 An example of compression with *Index* generation

then compresses the continuous presence or absence of both events into a singular point (i.e., column). Figure 4 provides a visual representation of the compression algorithm. As depicted in the figure, since $F_0 = F_1$, these event vectors are compressed into a single vector. The same holds for $F_2 = F_3$, $F_4 = F_5 = F_6$, and $F_7 = F_8$.

Without compression, computing certain Allen's relations such as *equals, overlaps,* or *during* in Allen's relations can be challenging to perform in a single step using bitwise operations. This challenge arises from the difficulty in distinguishing between event arrays that contain the long continuous presence of events. However, when the bit array is represented in a compressed form, all continuous presences and absences are collected into a single point. This simplifies the process of applying bitwise operations to these arrays. For instance, in Fig. 4, e_2 starts (with) e_1 at f_2. In the uncompressed representation, determining up to which frame both events are present requires some calculations. However, in the compressed representation, we can easily look for the aligned pattern $p_1 = [0;1;1]$ and $p_2 = [0;1;0]$ since the first 1 value in the compressed representation compresses the continuous presence of 1 s in both event arrays.

If only the start time of bit changes is recorded, it may not be possible to maintain the end index when the bit arrays end with a continuous sequence of 1 s. To handle boundary cases when calculating Allen's relations using bitwise operations, we introduce a 0 value at both the beginning and end of compressed events. This can be observed in Fig. 4. The first added 0 corresponds to an index of -1, and the last added 0 corresponds to the end of the bitstreams.

Algorithm 1 Compression Algorithm

e_1, e_2: event arrays to be compressed
R: compressed list for each input
I: index array corresponding to original bit arrays

```
1:  procedure COMPRESS(e1, e2)
2:      R = [ [ 0 ], [ 0 ] ]                    ▷ List of two lists each initialized with 0
3:      I = [ -1 ]                                ▷ Initializing the I
4:      i = 0
5:      while i < len(e1) do
6:          I.add(i)
7:          while (i < len(e1) - 1 and (e1[i] = e1[i+1]) and (e2[i] = e2[i+1])) do
8:              i += 1
9:          end while
10:         R[0].add(e1[i])
11:         R[1].add(e2[i])
12:         i += 1
13:     end while
14:     R[0].add(0)
15:     R[1].add(0)
16:     I.add(i)
17:     return R[0], R[1], I
18: end procedure
```

The compression algorithm is presented in Algorithm 1. The output variables, R (compressed vector) and I (index vector), are initialized with 0 s. To determine continuous presence or absence, the condition specified in line 7 of the algorithm is used. The algorithm continues to loop through the bit arrays until a change in the previous state is detected. At each change, the algorithm adds the starting index and the values at the point of change to I and R, respectively. Once all the events have been looped through, the algorithm appends zero values to R and the last index to I. This is necessary to handle the boundary case and make the compressed representation consistent.

The compression algorithm has a linear time complexity, and recalculation is avoided by storing the compressed results. This allows for immediate access to the compressed forms of event pairs without the need to repeatedly recalculate them. However, since compression requires two event arrays at once, the system needs to perform compression for every possible event pair in the data.

3.2 Bitwise operations for Allen's relations

The bitmap representation of events enables the performance of bitwise operations on them. In this section, we will present expressions based on bit operations that are involved in determining the various Allen's relations. Table 1 defines the symbols used throughout this section.

Table 1 Symbols used for bitwise operations

Symbol	Meaning
X, Y	Compressed event arrays
x, y	Patterns in the event arrays
$\overleftarrow{Z}$	Left shift operation on Z
$\vec{Z}$	Right shift operation Z
&	Bitwise AND
|	Bitwise OR

3.2.1 Equals ($X = Y$)

In the case of an "equals" relationship between two events, the compressed aligned pattern vectors should be $p_i = p_j = [0\ 1\ 0]$. To obtain the final result, we need to identify all occurrences of this pattern in the event vectors. This reduces the problem to a pattern-finding task in the bitstreams. For this specific pattern, we need to identify all positions where both events have a non-occurrence, i.e., a 0 value. To achieve this, we use an intermediate array Z obtained by applying the $(not(X \mid Y))$ operation. Next, we need to identify places in the event vectors where there is an occurrence or a 1 value that lies between two non-occurrences on either side ([0 1 0] pattern). We achieve this by performing a bitwise AND operation on the arrays X, Y, $\vec{Z}$, and $\overleftarrow{Z}$, where $\vec{Z}$ and $\overleftarrow{Z}$ are obtained by shifting the intermediate Z vector to the right and to the left, respectively. Figure 5 provides an example of how the "equals" operation works. The expression for finding "equals" is as follows:

$$Equals(X, Y) = X\ \&\ Y\ \&\ \vec{Z}\ \&\ \overleftarrow{Z} \tag{1}$$

where Z is $not\ (X|Y)$

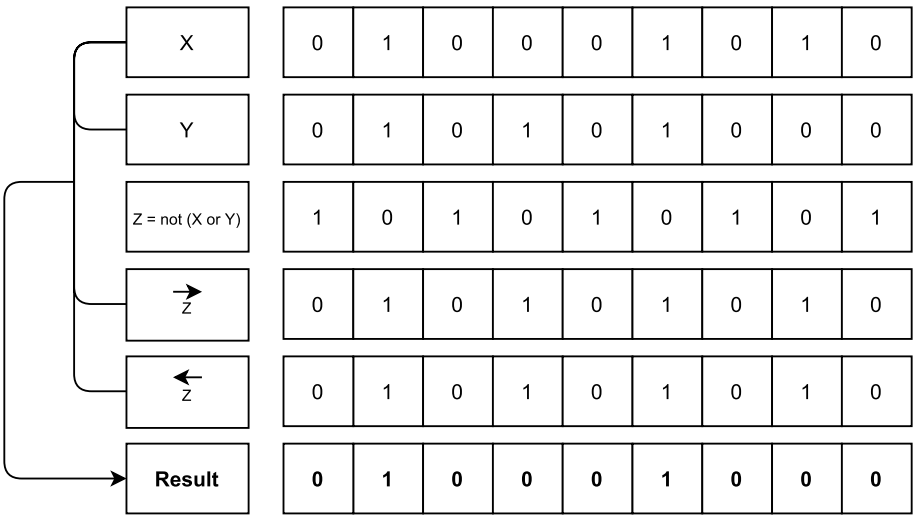


Fig. 5 Example for Equals operation

3.2.2 Before ($X < Y$)

While determining 'before' relations, it is important to note that there could be many cases of 'before' relations. For instance, let us consider two event vectors X and Y:

$X \leftarrow 0\ 1\ 0\ 1\ 0\ 0$

$Y \leftarrow 0\ 0\ 0\ 0\ 0\ 1$

In this example, event in X appears before event in Y at positions 1 and 3. Rather than focusing on every possible before-appearance, we focus on immediate before-appearances. For this, we need to search for two patterns in the event arrays.

The first pattern pair is $p_x = [1\ 0\ \_]$ and $p_y = [\_\ 0\ 1]$ where "_" represents a wildcard. Therefore, our objective is to find the instance of event X happening, then followed by the absence of both events and finally followed by the occurrence of event Y. This technique is effective because consecutive disappearances of both events are compressed into a single "0" value in the middle. The subsequent expression is used to identify this pattern (P1).

$$P1(X, Y) = X \ \& \ \overleftarrow{\overleftarrow{Y}} \ \& \ (not(\overleftarrow{X} \mid \overleftarrow{Y})) \tag{2}$$

The second pattern to look for is $x = [1\ 0]$ and $y = [0\ 1]$. This looks for locations where event X's occurrence is followed immediately by event Y's occurrence. Since this query does not involve wildcards, we find locations where "X & not Y" is followed by "Y & not X". The expression used to find this pattern(P2) is as follows:

$$P2(X, Y) = (X \ \& \ not\ Y) \ \& \ \overleftarrow{(Y \ \& \ not\ X)} \tag{3}$$

In this paper, we only examine before-appearances in the event vectors if the numerical representation of event vector X is greater than the numerical representation of event vector Y. This makes sure that X precedes Y. Otherwise, there could exist patterns having both Y precedes X, and X precedes Y. This is how we interpret 'before' relation in this paper. It is important to note that the original Allen's interval relations are established on pairwise intervals.

Lastly, we employ the disjunction (OR) operation on the values P1 and P2 acquired from Eqs. 2 and 3, respectively, to obtain the ultimate expression for Allen's "before" relation. The expression for the 'before' relation is presented next, followed by Fig. 6, which provides a visual explanation of the relationship.

$$Before(X,\ Y) = If\ X > Y\ then\ P1(X,\ Y) \mid P2(X,\ Y) \tag{4}$$

3.2.3 Overlaps ($X\ o\ Y$)

Overlaps relation is equivalent to finding the aligned pattern $p_x = [1\ 1\ 0]$ and $p_y = [0\ 1\ 1]$ (i.e., X and not Y) followed by (X and Y) and then by (not X and Y). Although the same expressions are used by the *meets* relation as well, there is a key difference between these relations. The main difference is that the overlapping segment should be an interval (not a single instant). Since the compressed representation maintains intervals, if results obtained from the conjunction (AND) of the mentioned expressions yield a compressed entry (an interval) instead of a single entry in the bit arrays, then it can be considered as overlaps.

To identify whether a point in the compressed representation of an event is an interval or a single entry, we define a new function called *CON*. This function takes in a sorted array of

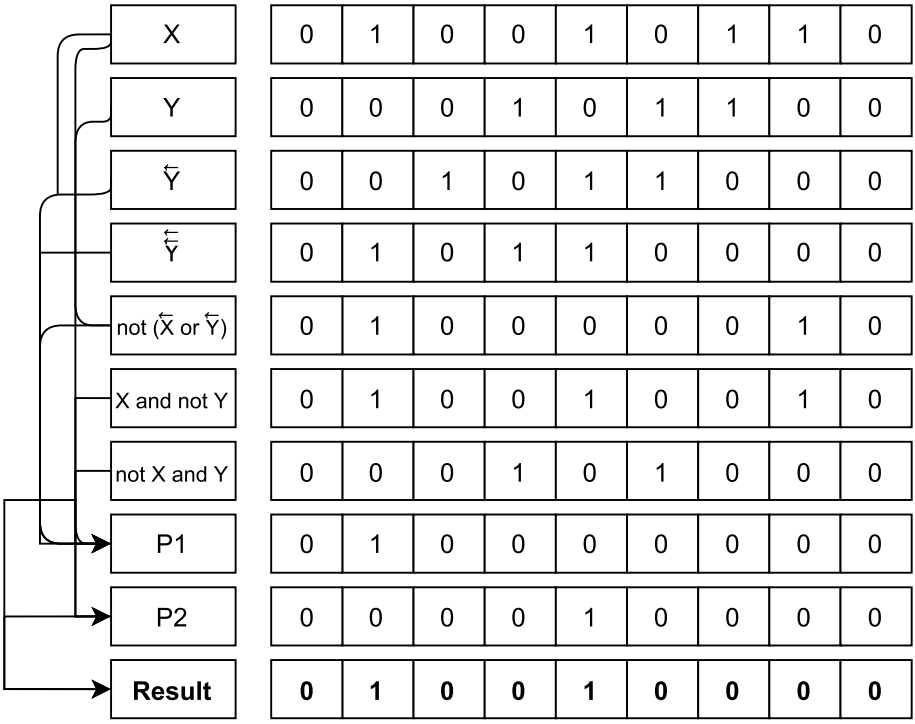


Fig. 6 Example for before operation

integers as input and returns a bit array of length equal to that of the input. The bit array will have a 1 value for consecutive values in the input array and a 0 value otherwise. This function operates on a sorted array of integers as input and produces a bit array of equal length. The resulting bit array will contain a 1 value for every pair of consecutive values in the input array, and a 0 value for every non-consecutive pair. For instance, consider the following example of the *CON* function:

$X \leftarrow [0, 1, 2, 5, 10, 11, 12]$

$CON(X) \rightarrow [1, 1, 0, 0, 1, 1, 0]$

With this function, determining whether an entry is a compressed interval or singular instant is straightforward. By examining the index array generated by our compression algorithm and comparing consecutive values, we can easily identify a singular instant by observing that its corresponding index array value in *CON(Index)* is equal to 1. In other words, for a single instant, the corresponding entry in *CON(Index)* will be a 1 value. Here, *Index* is the index array.

Since the expressions utilized for both *meets* and *overlaps* are identical, we can create a unified expression to ascertain whether either relationship exists between events. This function, which we shall refer to as *OverlapsOrMeets*, accepts the event vectors as input and returns 1 at the locations where intervals either overlap or meet. Its expression is presented as follows:

$$OverlapsOrMeets(X,\ Y) = (X\ \&\ Y)\ \&\ \overrightarrow{(X\ \&\ not\ Y)}\ \&\ \overleftarrow{(not\ X\ \&\ Y)} \tag{5}$$

Using the function *CON* and *OverlapsOrMeets* defined above, we then present the expression for the overlaps relation next with an example in Fig. 7.

$$Overlaps(X,\ Y) = OverlapsOrMeets(X,\ Y)\ \&\ not\ CON(Index) \tag{6}$$

where *Index* is the index array from the compression algorithm.

3.2.4 Meets (*X m Y*)

The meets relation is very similar to overlaps. The main difference is that the overlapping segment should be an instant (a single point or entry). Using the definition of functions *CON* and *OverlapsOrMeets* discussed in the previous section, we define the meets relation next with an example in Fig. 8.

$$Meets(X,\ Y)\ \leftarrow\ OverlapsOrMeets(X,\ Y)\ \&\ CON(Index) \tag{7}$$

where *Index* is the index array from the compression algorithm.

3.2.5 Starts (*X s Y*)

Starts relation requires a pattern search for $p_x = [0\ 1\ 0]$ and $p_y = [0\ 1\ 1]$. Hence, a search of (not X and not Y) followed by (X and Y) and then by (not X and Y) yields a

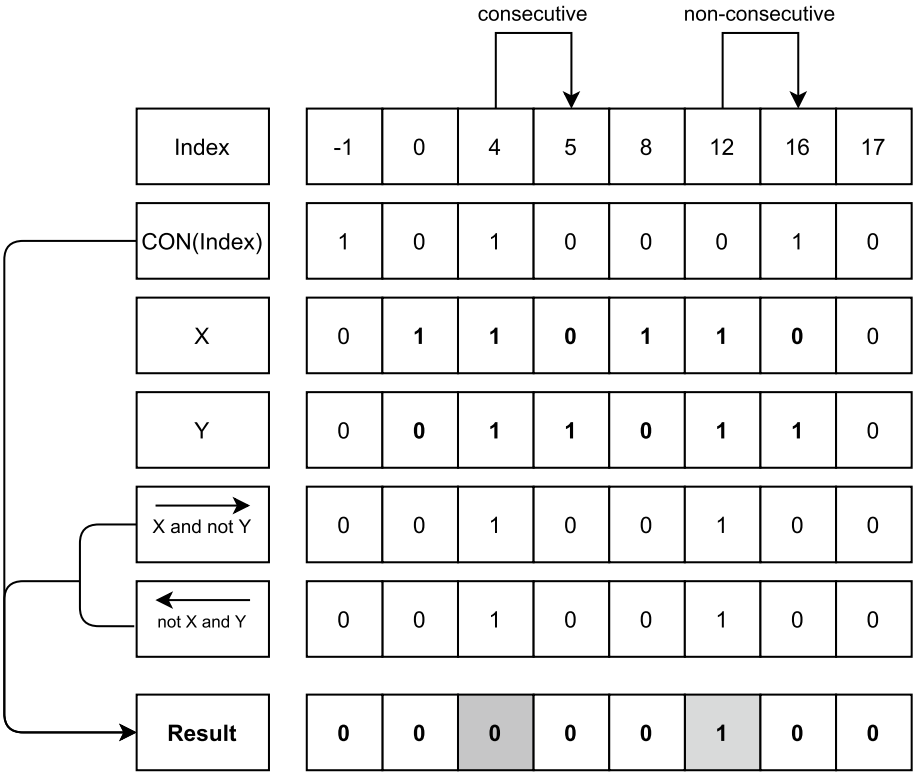


Fig. 7 Example for overlaps operation

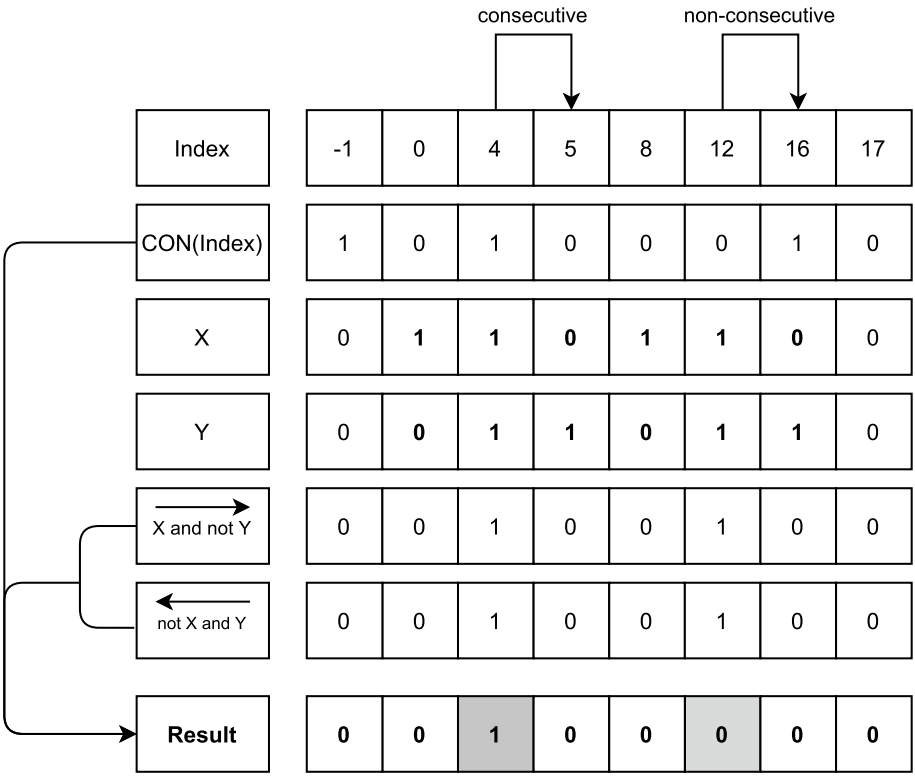


Fig. 8 Example for meets operation

starts relation. Figure 9 presents an example for the starts relation. The expression for this relation is as follows.

$$Result = (X\ and\ Y)\ \&\ \overrightarrow{(not\ X\ \&\ not\ Y)}\ \&\ \overleftarrow{(not\ X\ \&\ Y)} \tag{8}$$

3.2.6 During ($X\ d\ Y$)

During relation is a pattern search for $p_x = [0\ 1\ 0]$ and $p_y = [1\ 1\ 1]$. Hence, the resulting expression looks for (not X and Y) on both sides of (X and Y). Figure 10 presents an example of applying bitwise operations for determining during relation. The expression is as follows.

$$Result = X\ \&\ Y\ \&\ \overleftarrow{Z}\ \&\ \overrightarrow{Z} \tag{9}$$

where Z = not X & Y.

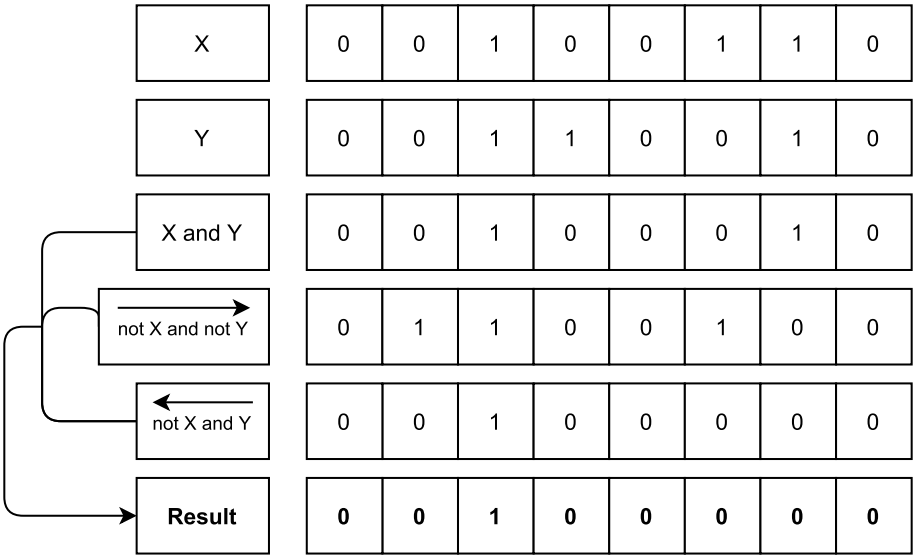


Fig. 9 Example for starts operation

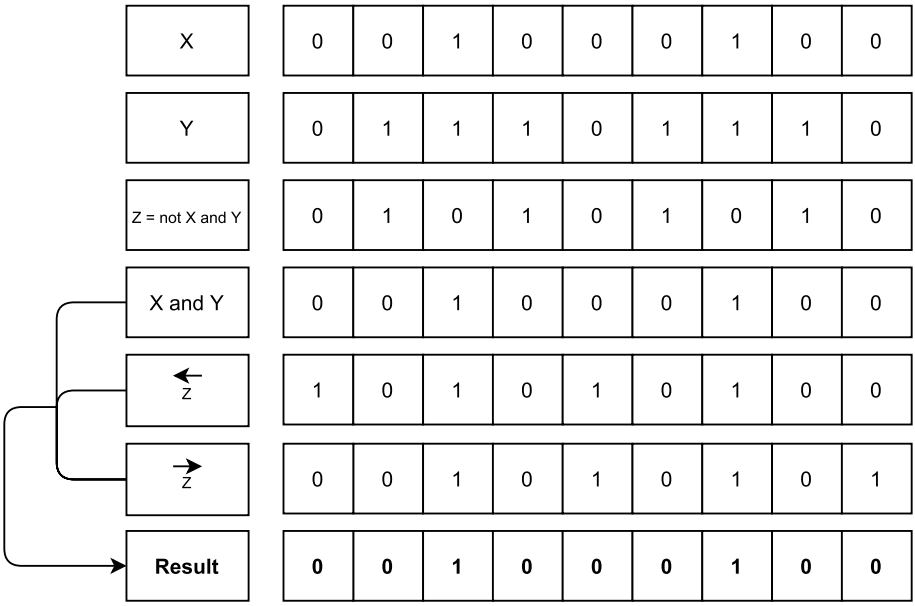


Fig. 10 Example for during operation

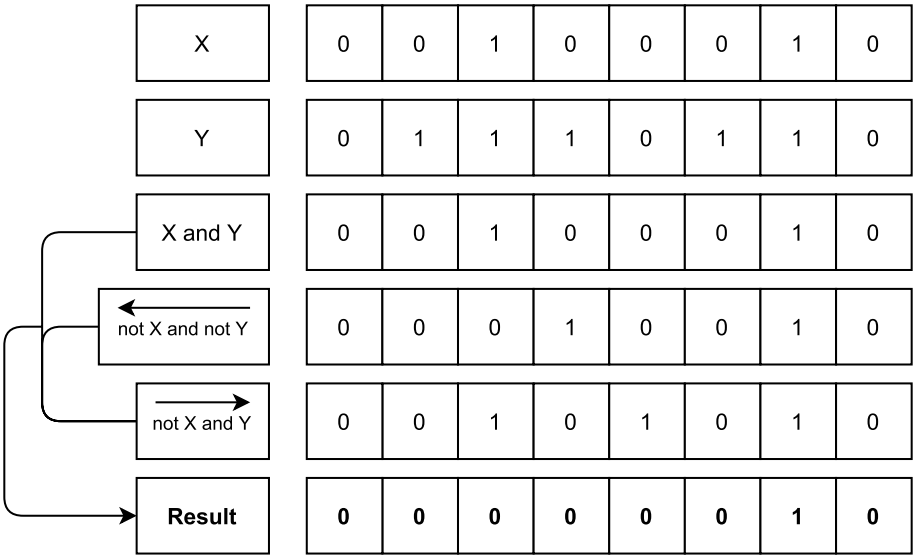


Fig. 11 Example for finishes operation

3.2.7 Finishes ($X\ f\ Y$)

X finishes Y is similar to the *starts* relation. Here, the resulting expression looks for pattern p_x = [0 1 0] and p_y = [1 1 0]. Hence the resulting expression searches for (not X and Y), (X and Y), and (not X and not Y) in both the event arrays. Figure 11 presents an example of applying bitwise operations for determining finishes relation.

$$Result = (X\ and\ Y)\ \&\ \overleftarrow{(not\ X\ \&\ not\ Y)}\ \&\ \overrightarrow{(not\ X\ \&\ Y)} \tag{10}$$

4 Validation of expressions

Rather than following a formal proof, we will use a brute-force technique to validate the expressions in the previous section. We generate ground truth from interval-based representations by identifying Allen's relations. Then we compare relations determined by our method with the ground truth. To accomplish this, a function called *ValidateAllens* is developed to compare the results obtained from bitwise operations with the traditional definitions of Allen's relations based on intervals. The pseudocode of Function *CheckAllAllens* is presented in Algorithm 2.

Algorithm 2 Algorithm for validating bitwise expressions for Allen's relations

INPUT: $\boldsymbol{S} \rightarrow$ Possible bit arrays of length n where every element in **S** represents an independent event. For example, for $n = 2$, $S = [\,[0,\ 0],\ [0,\ 1],\ [1,\ 0],\ [1,\ 1]\,]$.

OUTPUT: validation result as boolean

```
1:  function ValidateAllens(S)
2:      for A in S do
3:          for B in S do
4:              A_i ← toIntervals(A) // convert bit representations to intervals
5:              B_i ← toIntervals(B)
6:              // Find Allen's relations from intervals (ground-truth)
7:              AllenR_i ← FindAllenRels(A_i, B_i) // Find Allen's relations from intervals
8:              A_c, B_c, I_c ← compress(A, B) // compress bit representations
9:              // Find Allen's relations from compressed form
10:             AllenR_c ← FindAllenRelsCompressed(A_c, B_c)
11:             AllenR_ci ← [ ] // map Allen's relations from bit form to intervals
12:             for r_c in AllenR_c do:
13:                 r_ci ← toIntervalFromCompressed(r_c, I_c)
14:                 AllenR_ci.add(r_ci)
15:             end for
16:             if AllenR_i != AllenR_ci then // validate against the ground-truth
17:                 return False
18:             end if
19:         end for
20:     end for
21:     return True
22: end function
```

In Algorithm 2, the input **S** comprises all possible bit arrays of length n, where each element in **S** denotes an event vector. For instance, when $n = 2$, $S = [\,[0,\ 0],\ [0,\ 1],\ [1,\ 0],\ [1,\ 1]\,]$. The algorithm iterates through every feasible pair of events from the collection **S** by using nested loops to go through all possible pairs of event vectors. The variables A and B represent potential pairs of event arrays.

After obtaining the pair of events, the algorithm converts them to the conventional interval-based representation of events. This is done by using the function *toIntervals*, which accepts a bit array event as input and returns its conventional representation in the form of intervals. This conversion is necessary because the algorithm needs to establish a ground truth for validation.

Next, the algorithm produces Allen's relations using event arrays represented conventionally. This is accomplished by the *FindAllenRels* function, which is shown in line 7 of Algorithm 2. Given a pair of events represented conventionally, this function generates

all Allen's relations that exist between them. Now that the algorithm has established a ground truth for comparison, it can proceed to apply the bitwise operations described in this paper to validate our method. Since our technique relies on compressed events, the validation algorithm compresses the two-bit arrays A and B using the *compress* function (Line 10), which returns the compressed representation of the event arrays, along with the index array. The index array creates a mapping between the values in the compressed events and the actual indices of those same values in the uncompressed events.

Once compressed events are obtained, the algorithm then generates all Allen's relations present between the events using the function *FindAllenRelsCompressed*. It takes two compressed event arrays as input and provides every Allen's relation. The values obtained from this function can then be compared with the ground truth obtained from line 7. However, the conventional representation based output $AllenR_i$ and and the compressed output $AllenR_c$ are not directly comparable. Hence the algorithm converts the compressed output to conventional representation. This is done using the function *toIntervalFromCompressed* as seen in Lines 11 - 15 of the algorithm.

ValidatelAllens function only returns *True* if all pairs of event arrays yield the same result for both conventional definitions and the presented bitwise expressions based on definitions. *CheckAllAllens* function is further explained using an example as follows. Assume that the value of n is 8. Hence, there exist 256 uncompressed event arrays in $\mathbf{S}$. Firstly, consider the following uncompressed event arrays A and B as an example.

```
A = [1, 1, 0, 1, 0, 0, 1, 0]
B = [1, 1, 0, 1, 0, 1, 1, 1]
```

Then, the conventional representations (intervals) for event arrays A and B are generated as follows:

```
A_i = [ [0, 1], [3, 3], [6, 6] ]
B_i = [ [0, 1], [3, 3], [5, 7] ]
```

The Allen's relations based on the conventional presentation are computed as follows:

```
rels_conv = [
  equals =     [ [0, 1], [3, 3] ]
  before =     [ ]
  overlaps =   [ [6, 6] ]
  meets =      [ ]
  starts =     [ ]
  during =     [ ]
  finishes =   [ ]
]
```

The compressed representation and indices of event arrays a and b are calculated as:

```
A_c = [0  1  0  1  0  0  1  0  0]
B_c = [0  1  0  1  0  1  1  1  0]
I_c = [-1 0  2  3  4  5  6  7  8]
```

The bitwise operations determine Allen's relations and their positions as follows:

```
AllenR_c = [
  equals =    [0  1  0  1  0  0  0  0  0]
  before =    [ ]
  overlaps =  [0  0  0  0  0  0  1  0  0]
  meets =     [ ]
  starts =    [ ]
  during =    [ ]
  finishes =  [ ]
]
```

Finally, when our method is applied to compressed data by mapping results to the conventional form, the following Allen's relations are obtained:

```
AllenR_{ci} = [
  equals =    [ [0, 1], [3, 3] ]
  before =    [ ]
  overlaps =  [ [6, 6] ]
  meets =     [ ]
  starts =    [ ]
  during =    [ ]
  finishes =  [ ]

  ]
```

In the sample run shown above, it can be seen that $AllenR_i = AllenR_{ci}$. We ran the function *ValidateAllens* for $\mathbf{S}$ with different n values where $1 \leq n \leq 16$ and found that the function returns a *True* value for every case. Hence, we conclude that our expressions for identifying Allen's relations are correct.

5 Experiments and results

In this section, we explain the experiments that were conducted to show the performance of our approach. We explain the compression factors, the comparison of the performance with respect to an interval-based approach, and the application to a real scenario. We have run our experiments on a MacBook Pro with a 2.6 GHz 6-Core Intel Core i7 CPU, 16 GB 2400 MHz DDR4 RAM, and Intel UHD Graphics 630 1536 MB. To measure the performance of our method, we first evaluated our approach on simulated data with varying

probabilities of events. Later we tested our method on a real dataset. For simulated data, we generated a random bitstream of 1 s and 0 s of varying lengths with a variable probability of their appearance. This was done because it is easier to show the performance gain in our method as compared to other methods.

5.1 Compression analysis

To begin with the experiments, we generated random bit arrays with varying lengths, where the probability of a 0 appearing in the array is 0.95. This sparsity of the event matrix stimulates a real-world scenario. The algorithms were evaluated on a range of values of n, where $0 \leqslant n \leqslant N$, with $N = 10^8$. The results are shown in Fig. 12, which depicts the compression time as the array size increases. The algorithm operates in linear time, with a complexity of O(n), and can be optimized using ahead-of-time computations. Notably, this operation only needs to be performed once, and the results can be cached for future use.

As explained in Sect. 3.1, compression needs to be performed for every pair of events in the event matrix. When there are 'n' events in the matrix, this requires $C(n, 2)$ compression. Figure 13 illustrates the size comparison between event arrays and pairwise compressed arrays as the number of events increases. The figure displays several plots for different probabilities. It is evident from the figure that memory usage grows quadratically ($O(n^2)$). However, it is worth noting that when the event probability increases significantly, compression size will eventually decrease as there will be more common 1 s in the bit arrays. This can be seen in Fig. 14.

5.2 Comparison of methods on Allen's 'Equals' Relation

A question that may emerge is whether our technique can enhance the retrieval performance, or if there exist alternative approaches for processing bit array representations. In order to illustrate the comparative effectiveness of our method, we will evaluate it against

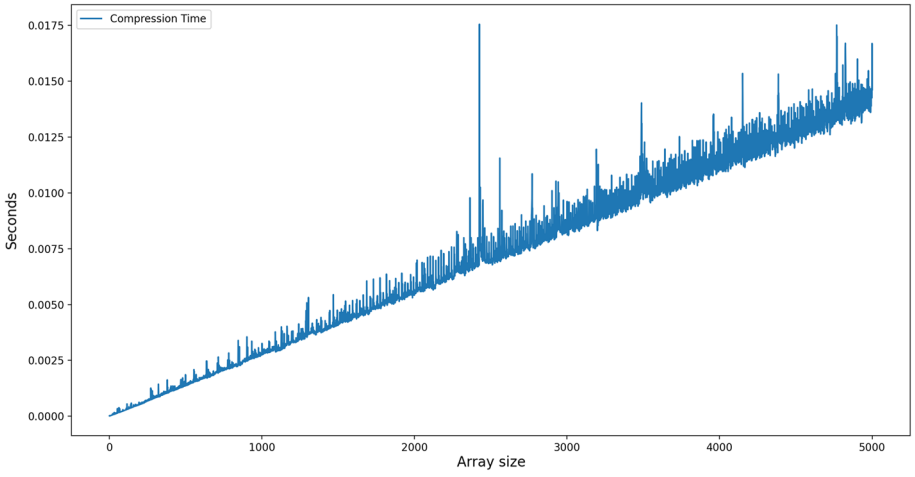


Fig. 12 Compression time vs array length

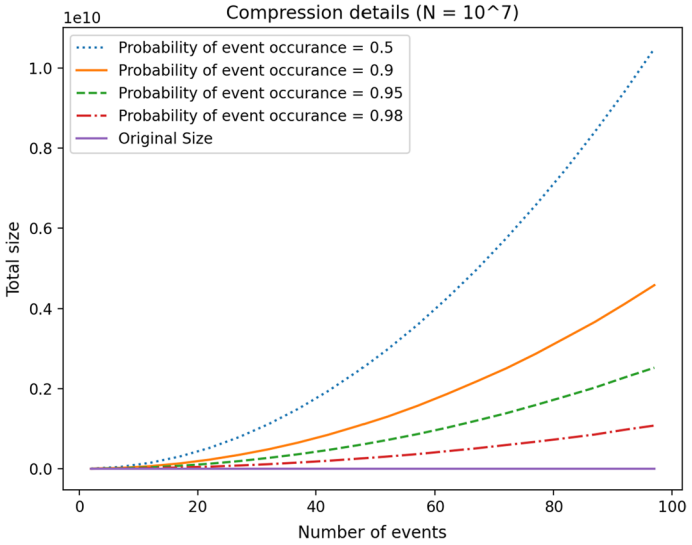


Fig. 13 Total array size for pair-wise storage of compressed arrays for an increasing number of events in the event matrix

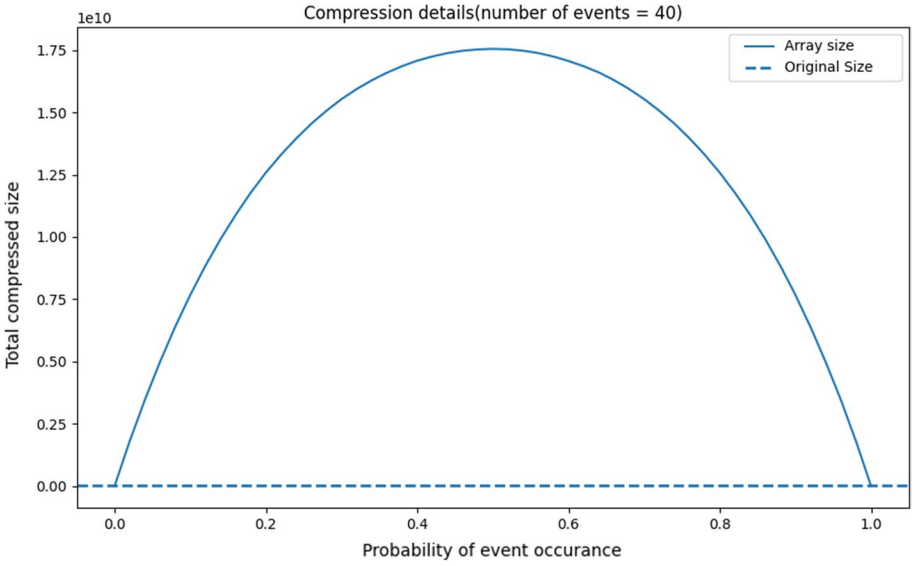


Fig. 14 Total memory consumption for pair-wise storage of compressed arrays for the varying probability of occurrence

three alternative approaches for the 'Equals' relation: (i) a conventional representation based on intervals, (ii) linear lookup using pattern matching, and (iii) vectorization with NumPy in Python. It should be noted that we have opted to use Python for the purposes of our experimentation.

5.2.1 Brief description of alternate approaches

Conventional Representation. The conventional representation just needs to check the equality of the intervals. Hence, a linear search between intervals can be conducted and further optimized if needed when the intervals are sorted based on their start times.

Linear Lookup using Pattern Match. "Equals" relation can be found by matching a certain pattern in the event bitmaps. Basically, we need to check for the patterns $p_x = [0\ 1\ 0]$ and $p_y = [0\ 1\ 0]$ as we linearly scan through the compressed event arrays.

Lookup using Vectorization. We have also conducted an analysis of the performance of pattern lookup using NumPy. By leveraging NumPy's vectorization capabilities, we could improve the retrieval performance. Additionally, this analysis serves to demonstrate that the performance gain we achieve with our proposed bitwise operations is not solely due to NumPy's vectorization. The method, *searchPatternNumpy*, presented in Algorithm 3, initially creates lists with a length equal to that of the *pattern* by applying a windowing technique to the arrays. For example, given $E_1 = [1, 0, 0, 1, 0]$ and $pattern = [0, 1, 0]$, all subarrays of length 3 are created from the E_1 array, i.e. $[1,\ 0,\ 0]$, $[0,\ 0,\ 1]$, and $[0,\ 1,\ 0]$, with the variable inds at line 4 holding the indexes used to create these arrays. The algorithm then proceeds to compare each of these subarrays to the *pattern*, returning only those that completely match the sequence. This process is applied to both event arrays, with a bitwise AND operation subsequently being performed to determine the locations where both event arrays contain the sequence. To ensure the resultant array maintains its length, an excess of $(pattern.size - 1)$ 0 s are added to the bitwise AND result, which is then right-shifted once to return true at the location of the 1's.

Algorithm 3 Allen's Equals relation using NumPy Based Pattern Detection

COMMENT: *Searches for a given pattern in the event arrays*
INPUT: E_1, E_2: event arrays
pattern: the pattern to look for

```
1:  procedure searchPatternNumpy(E1, E2, patttern)
2:      Na, Nseq = E1.size, pattern.size
3:      rSeq = np.arange(Nseq)
4:      inds = np.arange(Na - Nseq+1)[:, None] + rSeq
5:      M = (E1[inds] == pattern).all(1)
6:      N = (E2[inds] == pattern).all(1)
7:      newVal = np.logical_and(M, N)
8:      if newVal.any() > 0 then
9:          padded = np.concatenate((newVal, np.full(Nseq - 1, 0)))
10:         result = shift(padded, 1)
11:         return result
12:     end if
13:     return [ ]
14: end procedure
```

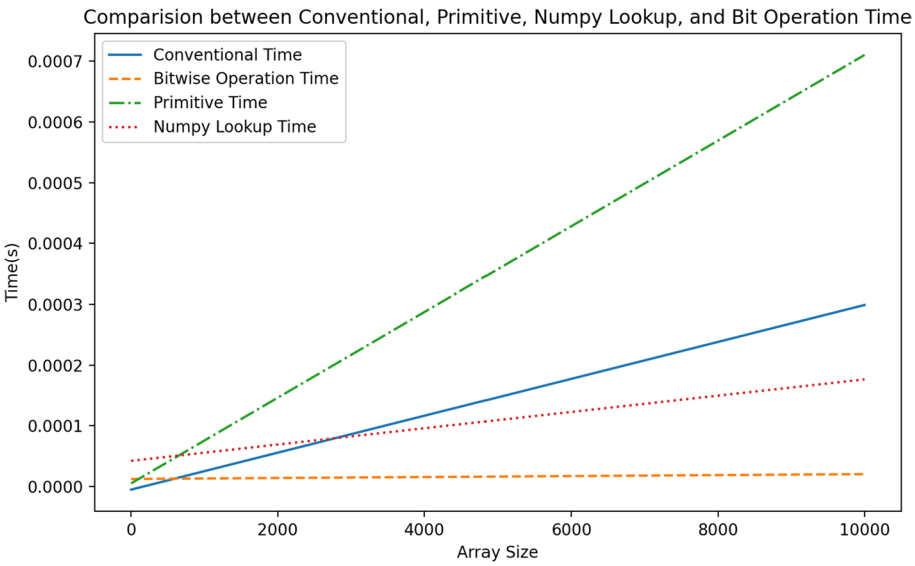


Fig. 15 Comparison between Conventional, Linear Lookup, NumPy Based Lookup, and Bitwise methods over growing value of array size

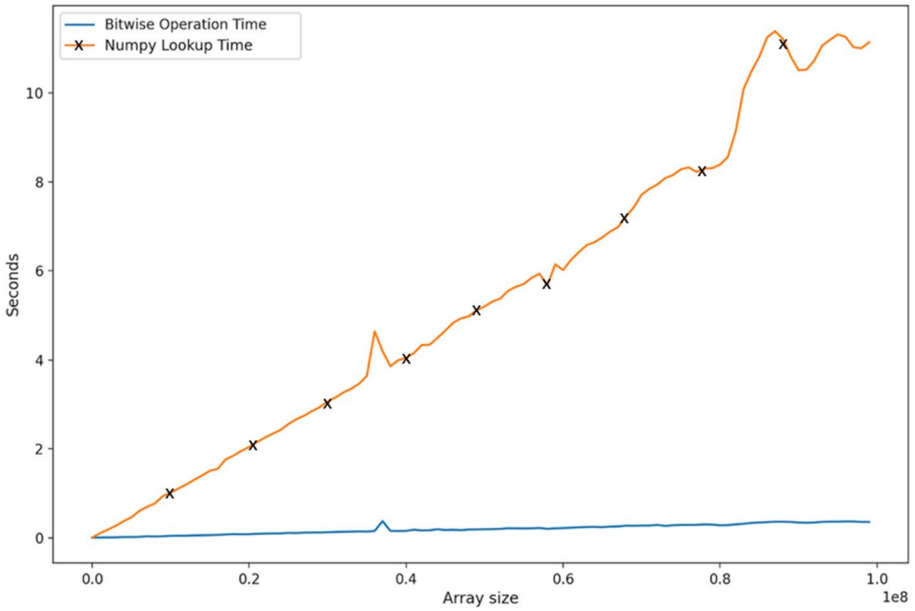


Fig. 16 Comparison between the proposed Bitwise operations (blue) and vectorized lookup (orange) methods over growing value of array size (Color figure online)

Table 2 Comparison of Allen's Equals relation using Conventional Representation, Linear Pattern Lookup, Vectorization Lookup, and the proposed Bitwise Operations

Method	Time(s)
Conventional	3.06
Pattern lookup	6.375
Vectorization	1.552
Proposed bitwise	0.073

5.2.2 Comparison of approaches

Figure 15 shows the comparison between the four mentioned methods including ours. When the array size is very small, all methods outperform the vectorized (NumPy) method, as there are few events to search for, as expected. Nonetheless, as the bit array size grows, the vectorization and bitwise operations methods become more efficient.

To confirm that the difference in speed is not solely due to NumPy's vectorization, we expanded the array size and evaluated the performance of both vectorized and bitwise operations-based methods. The comparison of the methods is presented in Fig. 16. The results demonstrate that our bitwise operations method surpasses the vectorized lookup approach by a significant margin, indicating that our technique's speed is not solely attributable to vectorization.

To present the actual runtime of our methods, we conducted an experiment using two uncompressed event arrays, each with a length of 10^8. The probability of an event occurring

was set as 0.95, resulting in approximately 5^6 intervals in each event. We computed the Allen's equals relation using all these methods mentioned earlier and recorded their respective timing information in Table 2. The results indicate that the proposed bitwise operations method outperforms the conventional representation method by almost 42 times and vectorized lookup by nearly 21 times.

5.3 Sample evaluation for a real world scenario

In this paper, we consider videos as temporal data and identify faces present in the videos as events in the data. To detect faces, we utilized an open-source Python library called Face Recognition (Geitgey, 2018), which, in turn, utilizes one of the pre-trained network libraries called Dlib (King, 2009) for face extraction and comparison. The accuracy of our entire system relies heavily on the quality of event extraction, which is the detection of faces in this case. To obtain the clustered face embeddings and generate the event matrix, the video is scanned.

We have used multiple video clips from the famous sitcom comedy series named Friends. Each video was three minutes in length, and we identified the appearance of six individuals as events: Monica, Ross, Chandler, Phoebe, Joey, and Rachel. The following operations were performed on each three-minute video clip with six identified individuals as events:

1. The generation of an event matrix using faces as events took an average of 1 min and 57 s per video.

Table 3 Timing information of Allen's Relations

Allen's Relation	Time Taken (μs)
Equals	14.35
Before	28.47
Meets	9.54
Overlaps	8.46
Starts	8.24
During	13.06
Finishes	7.74

Fig. 17 Frames showing the location where Monica (on the left) starts Chandler (on the right)

Fig. 18 Frames showing the location where Monica (on the left) appears during Phoebe (on the right)

2. Pairwise compression of every pair of events required an average of 33 milliseconds per video.
3. The calculation of all thirteen relations for each pair of events required an average of 0.83 milliseconds per video. The average time taken for each operation is summarized in Table 3.

Next, we provide two sample queries and their results.

Query 1: *Find all locations in the video where Monica starts Chandler.*

The query had a total runtime of 11 µs, with 9 µs spent on query time and 2 µs spent on indexing time. A sample result from the query is illustrated in Fig. 17, which shows Monica (on the left) and Chandler (on the right) appearing simultaneously. By the fourth frame, Monica is already unidentifiable, and as a result, the algorithm returns the starting time of these frames when Monica is first seen with Chandler.

Query 2: *Find all locations in the video where Monica appears during Phoebe.*

The query had a total runtime of 14 µs, with 12 µs spent on query time and 2 µs spent on indexing time. A sample result from the query is presented in Fig. 18. However, we can see that the algorithm produces an incorrect result in the figure. This is because Monica's face is seen from the side and is not recognized by the face recognition algorithm. Thus, the algorithm is only effective when events are extracted accurately from the video.

5.4 Discussion

If there are n events in the temporal data, then there will be $C(n, 2)$ event pairs that need to be stored for the video. This is because the compression algorithm presented only works when two events at once are taken into consideration. Since the compression algorithm alters the size of the event arrays, running the same query on multiple such pairs is time-consuming.

We have also assessed the time and space requirements of our proposed system. We observed a quadratic relationship between the space complexity of the system and the number of events in the system. However, we observed a significant reduction in the space requirement for sparse data, as the compression factor was found to be higher for sparse data.

6 Conclusion

This paper introduces a novel method for querying temporal data by utilizing bitwise operators for Allen's temporal relations. Our approach involves creating an event matrix and storing it in compressed arrays, which enables the use of bitwise operators for obtaining the Allen's temporal relations. The paper also discusses the challenges and complexities associated with compression and bitwise operations. Additionally, the study compares the proposed method with alternate methods and demonstrates its superior performance. Overall, the approach offers a promising solution for efficient querying of temporal data.

For future work, we plan to apply our method for ternary relations. If there are n events, our compression algorithm generates compressed bit representations for every possible pairwise array leading to generating $C(n, 2)$ compressed bit arrays. One possible way of extension to ternary relations is by compressing the arrays while considering all three of the arrays at once. However, that will decrease the compression ratio, and increase the memory consumption as $C(n, 3)$ events-triplets will have to be calculated, and the compressed event arrays might not even work for other Allen's relations mentioned in this paper. Alternatively, we also plan to apply bitwise operations on uncompressed event arrays. This way we can in parallel compute complex queries involving many events. Furthermore, there are various interpretations of the "before" relation, and it would be beneficial to examine different representations of this relation for future research.

Author contributions PK worked on this project as his thesis under the guidance of firstly by RA. Dr. Aygun has provided the research problem and the methodology, and all coding and experiments were conducted by PK. TM made sure that the work has been thorough and he was PK's advisor when he finished his defense. HC provided feedback especially on the usability aspects of the project. All authors reviewed manuscript.

Data availability The validation of expressions has been conducted on the fly. The query results are provided as sample results based on face detection and recognition in the paper. No further datasets were generated or analysed during the current study.

Declarations

Conflict of interest The authors have no competing interests as defined by Springer, or other interests that might be perceived to influence the results and/or discussion reported in this paper.

Ethical approval This research does not involve human participants and/or animals. Sample results include clip snaphots from a pular TV series.

References

Allen, J. F. (1983). Maintaining knowledge about temporal intervals. *Communications of the ACM, 26*(11), 832–843.

Aygun, R. S., & Bettaiah, V. (2017). In N. Lee (Ed.), *Query-by-Gaming* (pp. 1–10). Cham: Springer. https://doi.org/10.1007/978-3-319-08234-9_100-1

Baeza-Yates, R., Ribeiro-Neto, B., et al. (1999). *Modern Information Retrieval* (Vol. 463). New York: ACM press.

Bettaiah, V., & Aygun, R. S. (2015). Query-by-gaming: Interactive spatio-temporal querying and retrieval using gaming controller. *Journal of Visual Languages & Computing, 29*, 63–76.

Chan, C.-Y., & Ioannidis, Y. E. (1998). Bitmap index design and evaluation. In *ACM SIGMOD Record* (Vol. 27, pp. 355–366). ACM.

Chen, Z., Wen, Y., Cao, J., Zheng, W., Chang, J., Wu, Y., Ma, G., Hakmaoui, M., & Peng, G. (2015). A survey of bitmap index compression algorithms for big data. *Tsinghua Science and Technology, 20*(1), 100–115. https://doi.org/10.1109/TST.2015.7040519

Chuckravanen, D., Daykin, J., Hunsdale, K., & Seeam, A. (2017). Temporal patterns: Smart-type reasoning and applications.

Duan, H., Peng, Y., Min, G., Xiang, X., Zhan, W., & Zou, H. (2015). Distributed in-memory vocabulary tree for real-time retrieval of big data images. *Ad Hoc Networks, 35*, 137–148. https://doi.org/10.1016/j.adhoc.2015.05.006. Special Issue on Big Data Inspired Data Sensing, Processing and Networking Technologies.

Geitgey, A. (2018). Face Recognition. GitHub.

Georgala, K., Sherif, M. A., & Ngomo, A.-C. N. (2016). An efficient approach for the generation of allen relations. In *Proceedings of the Twenty-Second European Conference on Artificial Intelligence*. ECAI'16 (pp. 948–956). IOS Press, NLD. https://doi.org/10.3233/978-1-61499-672-9-948.

Google: Year in Search 2020. https://trends.google.com (2021)

Jain, V., & Aygun, R. (2008). Smart: A grammar-based semantic video modeling and representation. *In IEEE SoutheastCon, 2008*, 247–251. https://doi.org/10.1109/SECON.2008.4494294

Jain, V., & Aygün, R. S. (2009). Spatio-temporal querying of video content using sql for quantizable video databases. *Journal of Multimedia, 4*, 215–227.

Kang, K., Ouyang, W., Li, H., & Wang, X. (2016). Object detection from video tubelets with convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 817–825).

King, D. E. (2009). Dlib-ml: A machine learning toolkit. *Journal of Machine Learning Research, 10*(Jul), 1755–1758.

Li, M., Mani, M., Rundensteiner, E.A., & Lin, T. (2011). Complex event pattern detection over streams with interval-based temporal semantics. In *Proceedings of the 5th ACM International Conference on Distributed Event-Based System*. DEBS '11, pp. 291–302. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2002259.2002297

Mudrova, L., & Hawes, N. (2015). Task scheduling for mobile robots using interval algebra. In *2015 IEEE International Conference on Robotics and Automation (ICRA)* (pp. 383–388). https://doi.org/10.1109/ICRA.2015.7139027

Nadimi, S., & Bhanu, B. (2004). Physical models for moving shadow and object detection in video. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 26*(8), 1079–1087.

Naik, M., Jain, V., & Aygun, R. S. (2008). S3g: A semantic sequence state graph for indexing spatio-temporal data - a tennis video database application. In *2008 IEEE International Conference on Semantic Computing* (pp. 66–73). https://doi.org/10.1109/ICSC.2008.77.

Naik, M. M., Sigdel, M., & Aygun, R. S. (2012). Spatio-temporal querying recurrent multimedia databases using a semantic sequence state graph. *Multimedia Systems, 18*(3), 263–281. https://doi.org/10.1007/s00530-011-0255-8

Nebel, B., & Bürckert, H.-J. (1995). Reasoning about temporal relations: A maximal tractable subclass of allen's interval algebra. *Journal of the ACM, 42*(1), 43–66. https://doi.org/10.1145/200836.200848

Papadias, D., Mamoulis, N., & Delis, V. (2001). Approximate spatio-temporal retrieval. *ACM Transactions of Information System, 19*(1), 53–96. https://doi.org/10.1145/366836.366874

Patel, D., Hsu, W., & Lee, M. L. (2008). Mining relationships among interval-based events for classification. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (pp. 393–404).

Renz, J. (2001). A spatial odyssey of the interval algebra: 1. directed intervals. In *IJCAI* (pp. 51–56). Citeseer.

Rogers, J., Aygun, R., Etzkorn, L. (2022). Identifying variability in us covid-19 response through temporal partial ordering detection. In *2022 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)* (pp. 2266–2273). IEEE.

Rogers, J., Aygun, R., Etzkorn, L.: Cheat detection through temporal inference of constrained orders for subsequences. In *2022 IEEE Fifth International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)* (pp. 45–52). IEEE (2022)

Rogers, J., Etzkorn, L., & Aygun, R. (2023). Confidence based cheat detection through constrained order inference of temporal sequences. *International Journal of Semantic Computing*.

Rogers, J., Etzkorn, L., & Aygun, R. (2023). Temporaldedup: Domain-independent deduplication of redundant and errant temporal data. *International Journal of Semantic Computing*.

Shrestha, B., Chung, H., & Aygun, R. (2019). Temporal querying of faces in videos using bitmap index. In *2019 IEEE Conference on Multimedia Information Processing and Retrieval (MIPR)*, pp. 36–41. https://doi.org/10.1109/MIPR.2019.00015

Shrestha, B., Chung, H., & Aygün, R. S. (2019). Facetimemap: Multi-level bitmap index for temporal querying of faces in videos. *International Journal of Multimedia Data Engineering and Management (IJMDEM), 10*(2), 37–59.

Sinha, R.R., Mitra, S., & Winslett, M. (2006). Bitmap indexes for large scientific data sets: a case study. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium* (p. 10). https://doi.org/10.1109/IPDPS.2006.1639304

Stockinger, K., & Wu, K. (2007). Bitmap indices for data warehouses. In *Data Warehouses and OLAP: Concepts, Architectures and Solutions* (pp. 157–178). IGI Global.

Wang, L., Qiao, Y., & Tang, X. (2014).. Video action detection with relational dynamic-poselets. In *European Conference on Computer Vision* (pp. 565–580). Springer.

Wattamwar, S. S., & Ghosh, H. (2008). Spatio-temporal query for multimedia databases. In *Proceedings of the 2nd ACM Workshop on Multimedia Semantics*. MS '08 (pp. 48–55). Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1460676.1460686.

Zhang, S., & Zhang, C. (1999). Imc: A method for interval calculus in matrix. *Knowledge and Information Systems, 1*(2), 257–268.