# An Efficient Algorithm for Generating Maximal Interval Groups in Interval Databases

Nabil M. Arman

*Palestine Polytechnic University, Hebron, Palestine*

Received 16 February 2003; accepted 24 August 2003

**Abstract:** The development of efficient algorithms to enumerate all intervals that have certain properties has attracted a large amount of research efforts due to the important role of interval-based reasoning in different areas like rule-based systems, including Expert Systems (ESs), Information Distribution Systems (IDSs), and database systems to name a few. These algorithms are very crucial to answer certain queries about these intervals. This paper presents an efficient algorithm to generate all maximal interval groups form a given interval set.

**Key words:** Maximal interval groups; interval database.

## Introduction

Intervals are appropriate and convenient for representing events that span continuous period of time. One may query an interval database to determine what events occur during a given interval. Algorithms to enumerate all intervals that have certain properties have attracted a large amount of research efforts due to the important role of interval-based reasoning in different areas, including rule-based systems and database systems [1-5]. These algorithms have an important role in all these systems. An algorithm that finds an interval in an interval tree, represented as a red-black tree, which overlaps a given interval is presented in [4]. However, the algorithm has the overhead of building and maintaining the interval tree and it can only determine pairs of intervals that overlap. Our algorithm, on the other hand, determines all interval groups that overlap.

In a database of $n$ intervals, there is a need to find all maximal groups, where each group has the intervals that overlap. In a temporal database that stores all courses classes and their times, a query may be asked to generate all groups of classes that meet at a certain time point. In an IDS, it is always needed to check the time validity of rules to determine if they can be chained. This has an important role in controlling the operation of an IDS which is a corner stone of Command, Control, Communication, Computer, and Intelligence (C4I) systems. This paper presents an efficient algorithm to generate all maximal interval groups form a given interval set.

## Interval Grouping Algorithm

Before explaining the grouping algorithm, some concepts that will be used in the algorithm are explained. The algorithm uses the concept of event points and event point schedule [1]. An *event point* is a point on the spatial dimension, where some intervals are leaving a certain interval group and other intervals are entering another interval group. The set of these event points constitutes a *schedule of event points*. In our algorithm, the real schedule is determined dynamically as the algorithm progresses. The algorithm uses intervals where an interval $I = [t_1, t_2]$ is represented as an object with fields *low [I]* $= t_1$ (the low endpoint) and *high [I]* $= t_2$ (the high endpoint). Two intervals overlap if their intersection is not empty. The algorithm also sorts the intervals in *Lexicographic Ordering*. An interval set is sorted in lexicographic ordering if whenever interval *[i,j]* < *[h , k]* then either $i < h$ or $I = h$ and $j < k$. Let *IS* denote an interval set and let $t_1, t_2, ..., t_m$ denote all potential event points. Let $t_{m+1}$ be *high[last_interval],* which is an event point representing a guard condition for the algorithm. Let *LIG* ($t_i$) denote the Low Interval Group of $t_i$, which is the set of intervals *I* whose *high [I]* > = $t_i$ and *low [I]* <$t_i$. Let *UIG*($t_i$) denote the Upper Interval Group of $t_i$, which is the set of intervals *I* whose *low [I]* < $t_{i+1}$ *and low [I]* > = $t_i$, where $t_{i+1}$ is the next event point (determined dynamically). Then for every event point $t_i$ and its next event point $t_{i+1}$,

$IG(t_i) = LIG(t_i) \cup UIG(t_i) = LIG(t_{i+1}) + \Delta$, where

$\Delta = IG(t_i) - LIG(t_{i+1})$, and

$LIG(t_{i+1}) = IG(t_i) - \Delta$.

Therefore, $LIG(t_{i+1}) \subset IG(t_i)$, and

$LIG(t_{i+1}) = \{I \mid I \in IG(t_i) \wedge \text{high}[I] >= t_{i+1}\}$,

and thus it is clear that one can compute part of $IG(t_{i+1})$ while computing $IG(t_i)$.

The grouping algorithm is implemented by the procedure Determine_Interval_Groups (Fig.1), which can be invoked with any interval set *IS* to be grouped into maximal groups, such that each interval group *IG* has the maximum number of intervals such that for any interval $I_1$, and $I_2$ in *IG*, $I_1 \cap I_2 \neq \phi$. Determine_Interval_GroupsAlgorithm calls the procedure Determine_Potential_Event_Points, which is expalined in Section 2.1 and Determine_Next_Event_Point which is presented in Section 2.2 respectively. It then loops through all intervals adding intervals whose low endpoints are less than $t_i$ (which is the next event point) to the Upper Interval Group (*UIG*) of the current event point. While doing that, it also finds the minimum high endpoint in these intervals and determines all intervals that will belong to the next Lower Interval Group (*LIG*) of the next event point $t_{i+1} = t_j$ (intervals whose high endpoint is greater than or equal to $t_{i+1}$). The procedure then computes the union of the *LIG* and *UIG* to determine the Interval Group (*IG*) of the current event point. For the first group *LIG*

is $\phi$, since no intervals start with a low endpoint less than $t_j$ (the first event point). The procedure then continues and determines the next event point $t_{i+2}$, by calling Determine_Next_Event_Point, and adds all intervals in $LIG(t_{i+2})$ to the next event point lower interval group ($LIG(t_{i+2})$) and finds the minimum high endpoint. At this point, the procedure swaps $t_i$ and $t_j$, to continue with the rest of the event points.

```
Procedure Determine_Interval_Groups(Interval Set: IS)
{
        Sort IS in lexicographic ordering using heap or merge sort;
        Determint_Potential_Event_Points(event_points,tj);
        g=0; // Global Variable: event_points_index
        Determine_Next_Event_Point(ti,min_high_endpoint);
        LIG(tj)={};
        for all unused intervals "I" in IS do {
                GD(tj)=[tj,ti-1]; // GD represents Group Determinants Intervals
                while(low[I] < ti ) {
                        Add interval "I"  to UIG(tj);
                        if(high[I] >= ti) then {
                                Add interval "I" to LIG(ti);
                                if(high[I] < min_high_endpoint) then
                                        min_high_endpoint= high[I];
                        }
                }
                high[event_points[g]]=min_high_endpoint;//To be used in Next_Event_Point
                IG(tj)=LIG(tj) ∪ UIG(tj); // ∪ : represents UNION
                if there are still unused intervals "I" in IS then {
                        Determine_Next_Event_Point(tj, min_high_endpoint);
                        for all intervals "I" in LIG(ti) do {
                                if(high[I] >= tj) then
                                        Add interval "I" to LIG(tj);
                                if(high[I] < min_high_endpoint) then
                                        min_high_endpoint= high[I];
                        }
                        swap(ti,tj);
                }
        }
}
```

**Fig. 1.** Determine_Potential_Event_Groups.

## Determine_Potential_Event_Points Procedure

The Determine_Potential_Event_Points algorithm (Fig.2) is used to determine all potential event points that represent the set of all distinct low endpoints in the interval set. In doing that, the procedure also determines the first real event point ($t_j$), which is the low endpoint in the lexicographic ordering of the interval set. It also adds an event point with a value equal to the high endpoint of the last interval in

the lexicographic ordering to act as a guard condition for the main algorithm. The procedure is invoked once at the beginning of the algorithm execution.

```
Procedure Determine_Potential_Event_Points(event_points,tj);
{
        i=0;
        I=0;
        low[event_points[i]] = low[I];
        min_high_endpoint= low[I];
        tj= low[I]; // first real event point
        for all unused intervals "I" in IS do {
                if(high[I] < min_high_endpoint) then
                        min_high_endpoint= high[I];
                if(low[I] != low[event_points[i]] &&  low[I]  > min_high_endpoint) then {
                        high[event_points[i]] =min_high_endpoint;
                        i=i+1;
                        low[event_points[i]] = low[I];
                        exit;// exit for loop
                }
        }
        // Continue with the rest of potential event points
        for all remaining unused intervals "I" in IS do
                if(low[I] != low[event_points[i]]) then {
                        i=i+1;
                        low[event_points[i]] = low[I];
                }
        low[event_points[++i]] =high[last interval I]; // Guard Condition
}
```

**Fig. 2.** Determine_Potential_Event_Points Algorithm.

## Determine_Next_Event_Point Procedure

The Determine_Next_Event_Point algorithm (Fig.3) is used to determine the next event point during the execution of the Determine_Interval_Groups algorithm, using the minimum high endpoint determined during the generation of the various groups. The minimum high endpoints and the real event points are determined dynamically during the execution of the algorithm (they are not determined beforehand).

```
Procedure Determine_Next_Event_Point(ti,min_high_endpoint)
{       // Advance the pointer to the next event point
        temp_current_end=high[event_points[g]];
        g=g+1;
        while(low[event_points[g]] <= temp_current_end && g< no_event_points)
                g++;
        ti=low[event_points[g]];
min_high_endpoint= low[event_points[g]];  // For Next Event Point
}
```

**Fig. 3.** Determine_Next_Event_Point Algorithm.

*Theorem*

If an interval set $IS=\{I_1, I_2, I_3,...,I_n\}$, where $n$ is the number of intervals, then Determine_Interval_Groups algorithm generates all maximal groups *IGs* of *IS*, where each maximal group *IG* has the maximum number of intervals that overlap.

*Proof*

Any algorithm that produces the groups has to consider all points on the time calendar/dimension. To prove that the Determine_Interval_Groups algorithm works, we have to prove that it is considering all points on the calendar. We show that it is enough to consider only event points, which are interval low end points. Since we are considering the maximal groups, it is sufficient to consider event points where intervals are inserted into the groups (A group that is a subset of another group is not useful). A real event point is a point where some intervals are to be inserted into a group and other intervals are leaving as a result of inserting these new intervals (a new group is formed). Deleting an interval will only reduce the size of the group. Therefore, we need only consider the low end points of the intervals. The algorithm assumes that the intervals are indexed/sorted in ascending order using lexicographic ordering. Suppose that the intervals are $I_1, I_2, I_3,...,$ and $I_n$. Also suppose that for any interval *I* low end point is denoted by *low [I]* and the high end point is denoted by *high [I]*. Assume that the potential event points are $t_1, t_2, ..., t_m$, and $t_{m+1}$=*high [In]* (serves as a guard condition). Let the Low Interval Group ($LIG(t_i)$) denote the set of intervals *I* whose *high [I]* $> = t_i$ and *low[I]* $< t_i$ and the Upper Interval Group ($UIG(t_i)$) denote the set of intervals *I* whose *low[I]* $>= t_i$ and *low [I]* $< t_{i+1}$, where $t_{i+1}$ is determined dynamically. The first potential event point is $t_i$, where *i=1*. This is a real event point because there is no point before it (the intervals are sorted). The algorithm then determines the next event point $t_{i+1}$, which represents the point on the time dimension when intervals will start leaving the first group, since their high endpoints are greater than $t_{i+1}$. Intervals will start leaving if their *low [I]* > *min{high[I] | I belongs to the first group}*. Thus, next event point is:

$t_{i+1}$=*min{low[I] | low[I]*$> t_i \wedge$ *low [I] > min{high[I] | I belongs to the first group }}*

Thus, the group will be determined by $[t_i, t_{i+1}]$.

Two intervals *I* and *J* overlap, if $I \cap J \neq \phi$, which means that *low[I] <= high[J]* and *low[J]<=high[I]*. Therefore, intervals *I* to be in the group should satisfy the conditions:

*low[I] < = high[* $[t_i, t_{i+1}$ *[ ]*, or *low[I]* $< t_{i+1}$ and *low[* $[t_i, t_{i+1}$ *[ ]<=high[I]),* or $t_i$ *< = high[I],* or *high[I] >=* $t_i$*,* and thus

$IG(t_i)=LIG(t_i) \cup UIG(t_i)$, where

$LIG(t_i)=\phi$, since no intervals start with a point less than $t_1$.

$UIG(t_i)$ = *{I| low[I]* $< t_{i+1}$ *and high[I] >=* $t_i$*},* and therefore,

$IG(t_i)$= *{I| low[I]* $< t_{i+1}$ *and high[I] >=* $t_i$*}.*

$LIG(t_{i+1}) = \{I \mid I \in IG(t_i) \wedge high[I] >= t_{i+1}\}$ (to be used in the next group).
The first group that is determined by the algorithm is just this group.

Any point $< t_1$ is not relevant, since no interval has an interval low endpoint$< t_1$. Assume that there are some points $p_1, p_2, ..., p_r$, such that $t_i < p_1, p_2, ..., p_r < t_{i+1}$. These points do not affect the grouping, since no interval will leave any group at these points (otherwise, one of these points is the minimum high endpoint).

For any event point $t_j$ other than the first event point, the algorithm proceeds the same way, except that the condition that determines when the intervals will start leaving the groups is different. Intervals will start leaving if their *low[I] > min {high[I] | I* $\in LIG(t_j)\}$, thus

$t_{j+1} = min\{low[I] \mid low[I] > t_j \wedge low[I] > min\{high[I] \mid I \in LIG(t_j)\}\}$.

Therefore, the group will be determined by $[t_j, t_{j+1}[$. *low[I] <= high[* $[t_j, t_{j+1}[$, *or low[I]* $< t_{j+1}$ and *low[* $[t_j, t_{j+1}[$ *<=high[I]), or* $t_j$ *<=high[I], or high[I] >=* $t_j$, and thus

$UIG(t_j) = \{I \mid low[I] < t_{j+1}$ *and high[I] >=* $t_j\}$.

$IG(t_j) = LIG(t_j) \cup UIG(t_j)$, and thus

$IG(t_j) = \{I \mid I \in IG(t_j) \wedge high[I] >= t_{j+1}\} \cup \{I \mid low[I] < t_{j+1}$ *and high[I] >=* $t_j\}$

Again, this is equivalent to what the algorithm is producing.

Finally, for the last event point $t_m$, the algorithm determines $t_{m+1}$ = *high[In]* (guard condition for the algorithm). Thus, the group is determined by $[t_m$*,high[In]*$[$, and it proceeds in the same way. $UIG(t_m) = \{I \mid low[I] < high[In]$ *and high[I] >=* $t_m\}$, and this means that all intervals that are remaining will be included in $UIG(t_j)$ since *low[I] <high[In]* is not placing any constraint on the intervals. Notice that *high[In] >=* $t_m$, since intervals are sorted and a low endpoint is always smaller or equal to the high endpoint. Therefore, the algorithm is producing the $IG(t_m)$ for this case too, and thus the algorithm is considering all points on the time dimension, and generating all maximal groups and is correct.

## Complexity of the Determine_Interval_Groups Algorithm

1. The step that sorts the interval set has a complexity of *O (nlogn),* using merge sort or heap sort.
2. The procedure Determine_Potential_Event_Points that determines all potential event points, the first minimum high endpoint, and the first real event point in the interval set has a complexity of *O (n).*
3. The procedure Determine_Next_Event_Point that determines the next event point has a complexity of *O(m),* where *m* is the number of potential event points (the set of all distinct low endpoints of the intervals). Notice that *O(m)=O(n),* in the worst case, and therefore it is *O(n).* This part will be executed only *O(n)* times for the whole procedure, and not once per loop of the procedure.
4. The main *for* loop has a complexity of *O(n).*

5. The step that determines the output from the backward step in the intervals takes $k_1$ for all iterations (i.e., $k_1 = \sum_{\forall t_i} LIG(t_i)$.)
6. The step that determines the output from the forward step in the intervals takes $k_2$ for all iterations (i.e., $k_2 = \sum_{\forall t_i} UIG(t_i)$).
7. Steps (5) and (6) takes a total of $k=k_1+k_2 = \sum_{\forall t_i} LIG(t_i) + \sum_{\forall t_i} UIG(t_i)$, steps for all real event points $t_i$. Note that $k$ is the size of the output and thus has a complexity of $O(k)$.
8. Therefore, the whole procedure takes $O(n\log n)+O(n)+O(n)+O(k) = O(n\log n+k)$.

It is clear that the complexity of the procedure is bounded by the complexity of the sorting step, which is assumed to be $O(n\log n)$, and the size of the output $k$. If one can assume that the interval low endpoints and high endpoints are integers, then radix/bucket sorting can be used, and the complexity of the sorting step is $O(n)$, and in that case the complexity of the algorithm becomes $O(n)+O(n)+O(n)+O(k) = O(n+k)$. The algorithm is using the ideas of dynamic programming, where a subset of the output of one group is used as part of the output of the second group. As mentioned before, an algorithm that finds an interval in an interval tree, represented as a red-black tree, which overlaps a given interval is presented in [4]. The algorithm maintains an interval tree as a red-black tree, and thus has the overhead of building and maintaining the interval tree. In addition, the algorithm can only determine pairs of intervals that overlap. Our algorithm, on the other hand, determines all interval groups that overlap and during that it determines all group determinants, to be used if new intervals are added dynamically to the interval database, to determine to which group new intervals are to be added.

*Example*

Consider the interval set: {[0,1],[0,3],[0,5],[0,7],[0,9],[0,11],[2,13],[4,13]}
The algorithm sorts the interval set if it is not sorted.
The algorithm then determines all potential event points: 0,2,4, and 13 (Guard Condition)
The algorithm then proceeds as explained below.

(1) $t_j = 0$, $t_i = 2$, $LIG(t_j = 0) = \{\}$

$UIG(t_j = 0) = \{[0,1],[0,3],[0,5],[0,7],[0,9],[0,11]\}$

$LIG(t_i = 2) = \{[0,3],[0,5],[0,7],[0,9],[0,11]\}$

First group $IG(t_j = 0) = LIG(t_j = 0) \cup UIG(t_j = 0)$

$= \{[0,1],[0,3],[0,5],[0,7],[0,9],[0,11]\}$

$GD(t_j = 0) = [0,1]$

(2) $t_j = 4$, $t_i = 2$

$LIG(t_j = 4) = \{[0,5],[0,7],[0,9],[0,11]\}$

Swap($t_j$, $t_i$), thus $t_j = 2$, $t_i = 4$

$UIG(t_j = 2) = \{[2,13]\}$

$LIG(t_i = 4)$ = {[0,5],[0,7],[0,9],[0,11],[2,13]}

Second group $IG(t_j = 2) = LIG(t_j = 2)$

$UIG(t_j = 2)$={[0,5],[0,7],[0,9],[0,11],[2,13]}

$GD(t_j = 2)$ = [2,3]

(3) $t_j = 6$, $t_i = 4$

$LIG(t_j = 6)$ = {[0,7],[0,9],[0,11],[2,13]}

Swap($t_j$ , $t_i$), thus $t_j = 4$, $t_i = 6$

$UIG(t_j = 4)$ = {[4,13]}

$LIG(t_i = 6)$ = {[0,7],[0,9],[0,11],[2,13],[4,13]}

Third group

$IG(t_j = 4)$=$LIG(t_j = 4) \cup UIG$(tj = 4) = {[0,5],[0,7],[0,9],[0,11],[2,13],[4,13]}

$GD(t_j = 4)$ = [4,5]

Since all intervals have been used, the algorithm terminates with three groups. The algorithm also determines the interval groups determinants, which represent the smallest interval with which all intervals overlap. If any new interval is to be added to the set, it is only sufficient to check the group determinants to determine to which interval group the new interval is to be added.

## Conclusion

An efficient algorithm for generating the maximal interval groups has been presented. The algorithm is very crucial to answer certain queries about the intervals in an interval set. The algorithm can be used to generate the maximal interval groups needed in many Systems. The complexity of the algorithm has also been presented. The efficiency of the algorithm is clear from the given theoretical analysis, since its complexity is bounded by the complexity of sorting and the output size. In addition, the ideas of dynamic programming improved the performance of the algorithm since a subset of the output of one group is used as part of the output of the next group. Finally, the algorithm generates group determinants to determine to which interval group new intervals are to be added.

## References

1. Aiken A, Hellerstein J and Widom J (1995) Static Analysis Techniques for Predicting the Behavior of Active Database Rules. ACM Transactions on Database Systems, **20** (1): 3-41.
2. Chamberlain S (1994) Automated Information Distribution in Bandwidth-Constrained Environments (1994) *IEEE MILCOM Conference Record*, Vol. 2, October.
3. Chinn S and Madey G (1997) A Framework for Developing and Evaluating Expert Systems for Temporal Business Applications, *Expert Systems With Applications*.

4. Cormen T, Leiserson C, Rivest R and Stein C (2001) *Introduction to Algorithms*, McGraw-Hill Book Company.
5. Harrison J (1993) Active Rules in Deductive Databases, *ACM CIKM*, Washington DC, USA, November.

## خوارزمية فعالة لإيجاد مجموعات الفترات الكبرى في قواعد بيانات الفترات

نبيل محمد عرمان

*جامعة بوليتكنيك فلسطين، الخليل، فلسطين*

**ملخص**: إن تطوير خوارزميات فعالة لإيجاد جميع الفترات التي لها خصائص معينة قد استحوذ على قدر كبير من البحث وذلك للأهمية الكبيرة للدور الذي يقوم به التفسير/التبرير المعتمد على الفترات في مجالات عديدة مثل الأنظمة المعتمدة على القواعد/القوانين كالأنظمة الخبيرة ونظم توزيع المعلومات ونظم قواعد البيانات. إن هذه الخوارزميات مهمة جدا في الإجابة على بعض الاستفسارات المتعلقة بهذه الفترات. تقدم هذه الورقة خوارزمية فعالة لإيجاد المجموعات القصوى من الفترات المتطابقة من مجموعة فترات معينة.