Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

A Temporal Join Algorithm Based on Rank Key–to–Address Transformation

C. Y. Chen[1], C. C. Chang[2] and R. C. T. Lee[3]

[1]C. Y. Chen is with the Department of Electronics, Feng Chia University, Taichung, Taiwan 40724, Republic of China
[2]C. C. Chang is with the Institute of Computer Science and Information Engineering, National Chung Cheng University, Chiaji, Taiwan 62107, Republic of China
[3]R. C. T. Lee is with Providence University, Taichung, Taiwan 43301, Republic of China

ABSTRACT

In this paper, we present an efficient nested–loop like algorithm for the temporal join operation in temporal relational databases. The algorithm is based on the concept of rank KAT (key–to–address transformation) which was introduced by Ghosh in 1977. By pre–storing appropriate ranking information of endpoints of time intervals in the joining temporal relations, our algorithm can determine the resulting relation of a temporal join rapidly without requiring any scan on the joining relations. We further show that, for a given tuple in the outer relation satisfying the local predicates, the average case time complexity for the presented algorithm to determine all the inner relation tuples which match the given tuple is $O(\frac{r+3}{4})$, where r is the number of distinct endpoints of time intervals in the joining relations.

Keywords: Temporal database, temporal join, rank KAT (key–to–address transformation)

1. Introduction

Many real database applications intrinsically involve time–varying information. By a temporal relational database, we mean a database consisting of temporal relations. A temporal relation is a relation which involves some time–varying attributes (also called temporal attributes) and some time attributes that indicate the periods of time for which the given values of time–varying attributes are valid.

Temporal join is an important operation for temporal relations which matches tuples from two temporal relations whose time intervals overlap. In other words, a temporal join requires finding events among two temporal relations that happen at the same time. It is also a very expensive operation because temporal relations are usually very large and, unlike the "snapshot" join in traditional relational databases which is only an equijoin, temporal join is a non–equijoin operation in its nature. However, up to now, only a few efficient processing methods for the temporal join operation have been proposed [2,5,7–9,11,14].

Many of the previously proposed temporal join methods are in fact extensions of the existing three major methods for the conventional snapshot join; namely, the nested–loop join, the sort–merge join, and the partition join [5,8,9,11,14]. All these approaches inevitably need to scan or partially scan at least one joining relation and their performances depend on the average scan length through the relations. Leung and Muntz [7], on the other hand, introduced a stream processing approach for various temporal joins. This, however, involves some additional house–keeping cost. Recently, Chen, Chang and Lee [2] proposed a nested–loop like temporal join algorithm which was based upon a line segment intersection searching techniqe suggested by Overmars [10]. In their proposed method, each time interval is associated with a horizontal line segment in the plane and, by preparing an appropriate index for the line segments, the proposed method can find resulting relations without requring any scan on the joining relations.

In this paper, we are also concerned with efficient processing methods for the temporal join operation. Based upon the concept of rank KAT (key–to–address transformation) introduced by Ghosh [4], we present a new nested–loop like temporal join algorithm which doesn't requrie to scan any joining relation either. We further show that, for a given tuple in the outer relation satisfying the local predicates, the average case time complexity for the presented algorithm to

determine all inner relation tuples which match the given tuple is $O(\frac{r+3}{4})$, where r denotes the number of distinct endpoints of time intervals in the joining relations.

The rest of this paper is organized as follows. Section 2 describes the relational representation for temporal data that we adopt in this paper. Section 3 introduces the concept and the definition of the rank KAT. Section 4 first illustrates the idea and the technique of the proposed method while our proposed algorithm is described in the second part of this section. Section 5 consists of time complexity analysis and some discussions. Finally, conclusions and future research problems are given in Section 6.

2. Relational Representation for Temporal Data

The model that we use in this paper to represent temporal data as relations is described as follows.

Firstly, the time dimension is considered as a sequence of discrete, consecutive, equally–distanced time instants. A time interval consisting of time instants t_S, t_S+1, t_S+2,..., t_E is denoted as $<t_S, t_E>$, where t_S is called the starting time and t_E is called the ending time of the interval, respectively.

A temporal relation is a set of temporal tuples. A temporal tuple consists of the surrogate of the tuple, non–time varying attributes, time–varying attributes (also called temporal attributes), and two time attributes T_S and T_E. Interval $<T_S, T_E>$ is called the lifespan of a tuple which indicates the period of time that the given values of temporal attributes are valid. The lifespan of a temporal relation V is defined as the time interval $<L_S, L_E>$, where

$L_S=\min\{v.T_S \mid v\epsilon V\}$ and

$L_E=\max\{v.T_E \mid v\epsilon V\}$. Besides, all temporal relations are assumed to be in first temporal normal form [13], i.e., there are no two intersecting time intervals for a given surrogate instance.

Two time intervals $<t_S, t_E>$ and $<t_S^*, t_E^*>$ are said to intersect each other if and only if $t_S \leq t_E^*$ and $t_E \geq t_S^*$. For the special intersection where $t_S<t_S^*$ and $t_E^*<t_E$, we also say that $<t_S^*, t_E^*>$ is fully contained in $<t_S, t_E>$.

A temporal join over two temporal relations U and V is defined to consists of the concatenation of all tuples $u\epsilon U$ and $v\epsilon V$ such that their time intervals intersect.

Example 2.1

Consider two temporal relations DEP–TRABDG (department–travel budget) and EMP–DEP (employee–department) as shown in Table 2.1.

In DEP–TRABDG, D# is the surrogate, TRABDG is a temporal attribute, and T_S and T_E are time attributes. Similarly, in EMP–DEP, E# is the surrogate, D# is a temporal attribute, and T_S and T_E are time attributes. Note that both relations are in first temporal normal form. The lifespan of the second tuple in DEP–TRABDG is <5, 10> and that of the second tuple in EMP–DEP is <7, 20>. These two time intervals intersect each other. Besides, both relations have the same lifespan <1, 20>.

In order to answer the query Q_1"Find all employees who worked when at least one department had a travel budget greater than 35," we should first do a selection on DEP–TRABDG to get tuples satisfying the local predicate TRABDG>35, then temporally join these tuples with EMP–DEP. The resulting relation is shown in Table 2.2.

Similarly, to answer the query Q_2"Find the budget of each department when employee E_3 worked at department D_1," we should temporally join the sixth tuple of EMP–DEP with all tuples of DEP–TRABDG. The resulting relation is shown in Table 2.3.

Table 2.1 here

Table 2.2 here

Table 2.3 here

3. A Review of Rank KAT

KAT, also called hashing function, is well known as a fast technique for information storage and retrieval, and which has been

widely used in database management, compiler construction, and many other applications

KAT can be formally defined as a transformation which maps a key into an address location for storage or retrieval of the key and its associated information (see Figure 3.1) [4].

Figure 3.1 here

A bijective KAT, also called a minimal perfect hashing function, is a KAT which maps the set of keys one–to–one and onto the address space. Being bijective, it avoids collisions of keys (a collision occurs when two or more keys are mapped into the same address location) and there is no waste of memory locations in storing keys. Further, a bijective KAT allows single probe retrieval for each key. Accordingly, bijective KAT is the most desirable transformation in applying KAT techniques. Unfortunately, up to now, only a few approaches have been proposed for constructing bijective KATs [1,3,4,6,12]. Nevertheless, based upon an interesting bijective KAT introduced by Ghosh [4], called rank KAT, we are capable of designing an efficient processing algorithm for the temporal join operation (see Section 4).

The concept of the rank KAT is based on computing algebraically the rank of a key among the sorted list of the given keys. Suppose each key is represented as a bit string of the same length $(b_1b_2...b_d)$, where $b_i=0$ or 1 for i=1,2,...,d. Let β_i be the symbolic representation of the bit in the i–th position of the key, i.e., b_i is the value of β_i, for i=1,2,...,d. Further, let the function $h(\beta_{i_1},\beta_{i_2},...,\beta_{i_q};b_{i_1},b_{i_2},...,b_{i_q})$, $1\le i_1<i_2<...<i_q\le d$, denote the number of keys for which $\beta_{i_1}=b_{i_1}$, $\beta_{i_2}=b_{i_2}$,...,and $\beta_{i_q}=b_{i_q}$. Then the rank KAT for the key $(b_1b_2...b_d)$ is defined as follows [4].

$$t(b_1b_2...b_d)=m_0+b_1\cdot h(\beta_1;\overline{b}_1)+b_2\cdot h(\beta_1,\beta_2;b_1,\overline{b}_2)+b_3\cdot h(\beta_1,\beta_2,\beta_3;b_1,b_2,\overline{b}_3)+...+b_d\cdot h(\beta_1,\beta_2,...,\beta_d;b_1,b_2,...,b_{d-1},\overline{b}_d), \quad (3.1)$$

where m_0 is the initial address of the address space and $\overline{b}_i$ denotes the complement of b_i, for i=1,2,...,d.

In order to see why the rank KAT works correctly and illustrate the computation of $t(b_1b_2...b_d)$, let us give an example as follows.

Example 3.1

Consider the set of ten keys as listed in the first column of Table 3.1. The second column of Table 3.1 shows the 5–bit string representations of the keys. Suppose $m_0=1$. Then the address locations computed by (3.1) for all the keys are shown in the last column of Table 3.1. For instance, the address location of k_1 under the rank KAT is computed as

$$t(11101)=1+1\cdot h(\beta_1;0)+1\cdot h(\beta_1,\beta_2;1,0)+1\cdot h(\beta_1,\beta_2,\beta_3;1,1,0)+0\cdot h(\beta_1,\beta_2,\beta_3,\beta_4;1,1,1,1)+1\cdot h(\beta_1,\beta_2,\beta_3,\beta_4,\beta_5;1,1,1,0,0)$$

$$=1+1\times5+1\times2+1\times1+0+1\times0=9.$$

However, the problem is how to evaluate the values of h functions. Let us examine the binary tree corresponding to the set of keys as shown in Figure 3.2. Note that, in Figure 3.2, the value beside each node indicates the number of keys in the subtree of the node which, by the definitions of h functions, is actually identical to the value of $h(\beta_1,\beta_2,...,\beta_q;b_1,b_2,...,b_q)$, where $b_1b_2...b_q$, $1\le q\le d$, denotes the path from the root to the node. Also note that nodes which don't show up in the tree represent the cases where no keys exist in the subtrees of these nodes. For example, the path from the root to node B is 0 and the value beside node B is 5 imply that $h(\beta_1;0)=5$; the path from the root to node F is 10 and the value beside node F is 2 imply that $h(\beta_1,\beta_2;1,0)=2$. Similarly, from Figure 3.2, we obtain $h(\beta_1,\beta_2,\beta_3;1,1,0)$=the value beside node M=1 and $h(\beta_1,\beta_2,\beta_3,\beta_4,\beta_5;1,1,1,0,0)$= the value beside the left child of node V which doesn't appear in the tree=0. Since $b_4=0$ for

the key 11101, there is no need for evaluating $h(\beta_1,\beta_2,\beta_3,\beta_4;1,1,1,1)$.

Careful readers should have found that each of nodes B,F,M, and the left child of node V is the left child of some node on the path from leaf key node k_1 to the root and doesn't belong to the path. Accordingly, the t function for a key can be computed by simply summing up all the values beside the nodes which are left children of nodes on the path from the leaf key node to the root and don't belong to the path. And, consequently, it takes O(d) time, where d is the length of the bit string representing the key. As an example, by traversing the tree from k_1 to root A, we have

$t(11101)$=1+all the values beside the doubly circled nodes
$=1+1+2+5=9$.

Table 3.1 here

Figure 3.2 here

From the above example, we realize that rank KAT does guarantee to give the rank of each key among the sorted list of the given keys and the time taken to compute the rank KAT value $t(b_1b_2...b_d)$ for key $(b_1,b_2,...,b_d)$ is O(d). However, one disadvantage for rank KAT is that the values of h functions for computing rank KAT are data dependent. That is, if some keys are added or deleted, the values of rank KAT will also be changed. Consequently, rank KAT is good only for static sets of keys for which keys are not added or deleted too often.

4. A New Temporal Join Algorithm Based upon Rank KAT

In this section, we present a new temporal join processing method which is based on the rank KAT that we have reviewed in the last section. The idea and the technique are introduced as follows.

4.1 The Idea and the Technique

Let there be two temporal relations U and V. Suppose we want to temporally join a given tuple u* of U with the full relation V. We should first find out all intervals in V which intersect the interval $<u^*\cdot T_S, u^*\cdot T_E>$. Then concatenate the corresponding tuples of V with u*. Thus the real issue is knowing how to determine these intersecting intervals efficiently and rapidly.

Note, by definition, that an interval $<v\cdot T_S, v\cdot T_E>$ in V intersects $<u^*\cdot T_S, u^*\cdot T_E>$ if and only if $u^*\cdot T_S \le v\cdot T_E$ and $v\cdot T_S \le u^*\cdot T_E$. Further note that $u^*\cdot T_S \le v\cdot T_E$ and $v\cdot T_S \le u^*\cdot T_E$ are equivalent to (1) $u^*\cdot T_S \le (v\cdot T_S$ or $v\cdot T_E) \le u^*\cdot T_E$, or (2) $v\cdot T_S < u^*\cdot T_S$ and $u^*\cdot T_E < v\cdot T_E$. Now, by the rank preserving property of rank KAT, we see that $t(k_1)<t(k_2)$ if and only if $k_1<k_2$, for any two given keys k_1 and k_2. Thus, by applying rank KAT to the set of keys $\{u\cdot T_S, u\cdot T_E, v\cdot T_S, v\cdot T_E \mid u\in U$ and $v\in V\}$, it is easy to see that $<v\cdot T_S, v\cdot T_E>$ intersects $<u^*\cdot T_S, u^*\cdot T_E>$ if and only if (1) $t(u^*\cdot T_S) \le (t(v\cdot T_S)$ or $t(v\cdot T_E)) \le t(u^*\cdot T_E)$, or (2) $t(v\cdot T_S)<t(u^*\cdot T_S)$ and $t(u^*\cdot T_E)<t(v\cdot T_E)$. This reveals that we can find the required intersecting intervals through the technique of rank KAT if appropriate ranking information of endpoints of time intervals in the joining relations are available.

4.2 The Algorithm

The above discussed idea and technique can be formally implemented as the following algorithm.

Algorithm RKATTJ(Rank KAT Temporal Join)

Input: Two temporal relations U and V, where $|U|=m$ and $|V|=n$.

Output: The resulting relation for temporal join of U and V.

Preprocessings:

Apply rank KAT on the set of all endpoints of time intervals in U or V to construct a sequence of information sets $INFO_1, INFO_2, ..., INFO_r$ to indicate the ranks of endpoints and the relationships of full containments for all time intervals in U and V. Here r denotes the number of distinct

endpoints of time intervals in U and V. The constructing steps are as follows.

1. For i=1 to m do
   compute $t(u_i \cdot T_S)$ and $t(u_i \cdot T_E)$
   add (U,i,S) to $INFO_{t(u_i \cdot T_S)}$ // store the rank information of $u_i \cdot T_S$ in // // $INFO_{t(u_i \cdot T_S)}$//
   add (U,i,E) to $INFO_{t(u_i \cdot T_E)}$
   end for
2. For j=1 to n do
   compute $t(v_j \cdot T_S)$ and $t(v_j \cdot T_E)$
   add (V,j,S) to $INFO_{t(v_j \cdot T_S)}$
   add (V,j,E) to $INFO_{t(v_j \cdot T_E)}$
   end for
3. For i=1 to m do
   compute $t(u_i \cdot T_S)$ and $t(u_i \cdot T_E)$
   if $t(u_i \cdot T_S) \le r - t(u_i \cdot T_E)$
   then do
   for each $(V,x,S) \in INFO_1 \cup INFO_2 \cup \ldots \cup INFO_{t(u_i \cdot T_S)-1}$
   if $t(v_x \cdot T_E) > t(u_i \cdot T_E)$
   then add (V,x,I) to $INFO_{t(u_i \cdot T_S)}$ // $<u_i \cdot T_S, u_i \cdot T_E>$ is fully // // contained in $<v_x \cdot T_S, v_x \cdot T_E>$ //
   end if
   end for
   else do
   for each $(V,x,E) \in INFO_{t(u_i \cdot T_E)+1} \cup INFO_{t(u_i \cdot T_E)+2} \cup \ldots \cup INFO_r$
   if $t(v_x \cdot T_S) < t(u_i \cdot T_S)$
   then add (V,x,I) to $INFO_{t(u_i \cdot T_E)}$ // $<u_i \cdot T_S, u_i \cdot T_E>$ is fully // // contained in $<v_x \cdot T_S, v_x \cdot T_E>$ //
   end if
   end for
   else
   for each $(V,x,S) \in INFO_{t(u_i \cdot T_S)+1} \cup INFO_{t(u_i \cdot T_S)+2} \cup \ldots \cup INFO_{t(u_i \cdot T_E)-1}$
   if $t(v_x \cdot T_E) < t(u_i \cdot T_E)$
   then add (U,i,I) to $INFO_{t(v_x \cdot T_E)}$ // $<u_i \cdot T_S, u_i \cdot T_E>$ fully // // contains $<v_x \cdot T_S, v_x \cdot T_E>$ //
   end if
   end for
   end for
4. Store $INFO_i$, $1 \le i \le r$, in the i–th address location determined by the rank KAT.

Steps:

1. Input the next tuple u of U
2. Compute $t(u \cdot T_S)$ and $t(u \cdot T_E)$ // the matching V–tuples for u can be found in// // $INFO_{t(u_i \cdot T_S)} \cup INFO_{t(u_i \cdot T_S)+1} \cup \ldots \cup INFO_{t(u_i \cdot T_E)}$ //
3. For $k=t(u \cdot T_S)$ to $t(u \cdot T_E)$ do
   for each tuple v in V which appears in $INFO_k$ do
   input v
   output the resulting temporal join tuple of u and v
   end for
   end for
4. Go to Step 1.

Example 4.1

Consider the temporal relations DEP–TRABDG and EMP–DEP, and queries Q_1 and Q_2 that we have met in Example 2.1. For simplicity, let U denote DEP–TRABDG and V denote EMP–DEP. Then, according to Algorithm RKATTJ, we have, in preprocessing stage, a sequence of information sets $INFO_1, INFO_2, \ldots, INFO_{14}$ as shown in Table 4.1.

Table 4.1 here

Now, to answer query Q_1 "Find all employees who worked when at least one department had a travel budget greater than

35," we first select u_2 and u_4 from U which satisfy the local predicate TRABDG>35. Since $t(u_2 \cdot T_S)=t(5)=3$ and $t(u_2 \cdot T_E)=t(10)=7$, those tuples of V appeared in $INFO_3 \cup INFO_4 \cup ... \cup INFO_7$, i.e.,$v_1$,$v_2$,$v_3$,$v_5$ and v_6, are the matching V–tuples for u_2. Similarly, since $t(u_4 \cdot T_S)=t(1)=1$ and $t(u_4 \cdot T_E)=t(15)=11$, those tuples of V appeared in $INFO_1 \cup INFO_2 \cup ... \cup INFO_{11}$, i.e.,$v_1$,$v_2$,$v_3$,$v_5$,$v_6$ and v_7, are the matching V–tuples for u_4. Finally, temporally join these matching tuples of u_2 and u_4 with u_2 and u_4, respectively. We have the resulting relation as shown in Table 2.2, which answers Q_1.

Next, consider the query Q_2"Find the budget of each department when employee E_3 worked at department D_1." The only tuple which satisfies the local predicate is v_6. Since $t(v_6 \cdot T_S)=t(8)=6$ and $t(v_6 \cdot T_E)=t(12)=9$, the U–tuples which match v_6 are those which appear in $INFO_6 \cup INFO_7 \cup INFO_8 \cup INFO_9$, i.e.,$u_2$, u_3,u_4 and u_6. Temporally join these tuples with v_6. The resulting relation is shown in Table 2.3, which answers Q_2.

5. Complexity Analysis and Some Discussions

It is meaningless to speak of worst case time complexity for temporal join operation since, in worst case, each tuple in one relation matches all tuples in the other joining relation. However, average case time complexity is often difficult to analyze and there has been no average case time complexity being given for all previously proposed temporal join algorithms. Nevertheless, the average case time complexity for Algorithm RKATTJ can be easily analyzed as follows.

The complexity of Algorithm RKATTJ is mainly dominated by that of Step 3 which, however, is dominated by the value $L_u=|t(u \cdot T_E)-t(u \cdot T_S)|$. Note that since $u \cdot T_S \le u \cdot T_E$, we have $1 \le t(u \cdot T_S) \le t(u \cdot T_E) \le r$. Suppose that the probability of $t(u \cdot T_S)$ being a, $1 \le a \le r$, is $\frac{1}{r}$. Then the expected value of L_u is $\frac{r+3}{4}$ which can be evaluated as follows.

$$E(L_u)=E(E(L_u \mid t(u \cdot T_S)))$$
$$=\frac{1}{r}(E(L_u \mid 1)+E(L_u \mid 2)+...+E(L_u \mid r))$$
$$=\frac{1}{r}[\frac{1}{r}(1+2+...+r)+\frac{1}{r-1}(1+2+...+r-1)$$
$$+...+\frac{1}{r-a+1}(1+2+...+r-a+1)+...+1]$$
$$=\frac{r+3}{4}.$$

Accordingly, for a given tuple u of U, it takes $O(\frac{r+3}{4})$ time in average to determine and temporally join the matching V–tuples. Therefore, the average case time complexity of Algorithm RKATTJ is $O(m \cdot \frac{r+3}{4})$. Recall that r is the number of distinct endpoints of time intervals in the joining relations U and V.

The preprocessing time of the presented algorithm can also be given as follows. Steps 1 and 2 compute rank KAT values of endpoints of all intervals in U and V which need O((m+n)d) time, where d is the number of bits used to represent the endpoints as bit strings. Step 3 takes advantage of information sets $INFO_i$, $1 \le i \le r$, partially constructed in Steps 1 and 2, to determine all intervals $<u \cdot T_S, u \cdot T_E>$ and $<v \cdot T_S, v \cdot T_E>$ such that $<u \cdot T_S, u \cdot T_E>$ fully contains $<v \cdot T_S, v \cdot T_E>$ or vice versa. The time to be taken is at most O(mn).

It has been pointed out in [9] that the nested–loop method is a reasonably good choice for the temporal join operation. Our approach also behaves like a nested–loop algorithm. However, unlike most previously suggested nested–loop temporal join algorithms which need to scan or partially scan the inner and/or outer relations, our approach doesn't require any scan on the joining relations. The line segment intersection based temporal join method proposed recently by the authors themselves [2] doesn't require any scan either. However,

the rank KAT based method still has a lot of merits over the line segment intersection based method. This includes (1) simpler data structure, (2) faster computation, and (3) easier implementation. In addition, our approach also provides an efficient direct access to the matching tuples when the tuples in each joining relations are clusterly stored according to the ranks of starting points of the associated time intervals.

It has been pointed out, in Section 3, that the main disadvantage of rank KAT is that it is a static KAT. Accordingly, the main disadvantage of our approach is that it is good only for static temporal databases for which data are not added or deleted too often.

6. Conclusions

In this paper, based upon the concept of rank KAT, we have presented a new algorithm for the temporal join operation. An efficient direct access to matching tuples is also provided. Since it is meaningless to speak of worst case time complexity for a temporal join algorithm, we have analyzed the average case time complexity of the presented algorithm.

Comparing with other previously suggested methods, it can be seen that our approach has three advantages. They are (1) the data structure used is simple, (2) the computation needed is little, and (3) the implementation is easy. However, since rank KAT is a static KAT, it has also been pointed out that the main disadvantage of our approach is that it is not suitable for "dynamic" temporal databases for which tuples are frequently added or deleted. Therefore, we are now working toward the design of a dynamic rank KAT to overcome this problem.

REFERENCES

[1] C.C. Chang: "The Study of an Ordered Minimal Perfect Hashing Scheme," Commun. ACM, Vol. 27, No. 4, pp. 384–387, 1984.

[2] C.Y. Chen, C.C. Chang and R.C.T. Lee: "A Line Segment Intersection Based Temporal Join," to appear in ADTI'94, Nara, Japen, Oct. 1994.

[3] R.J. Cichelli: "Minimal Perfect Hash Function Made Simple," Commun.ACM, Vol. 23, No. 1, pp. 17–19, 1980.

[4] S.P. Ghosh: "Data Base Organization for Data Management," Academic Press, New York, N.Y., pp. 147–151, 1977.

[5] H. Gunadhi and A. Segev: "Query Processing Algorithms for Temporal Intersection Join," Proceedings of the Seventh International Conference on Data Engineering, pp. 336–344, Kobe, Japan, April 1991.

[6] G. Jaeschke: "Reciprocal Hashing: A Method for Generating Minimal Perfect Hashing Function," Commun. ACM, Vol. 24, No.12, pp. 829–833, 1981.

[7] T. Leung and R. Muntz: "Query Processing for Temporal Databases," Proceedings of the Sixth International Conference on Data Engineering, pp.200–208, Los Angeles, CA, April 1990.

[8] T. Leung and R. Muntz: "Temporal Query Processing and Optimization in Multiprocessor Database Machines," Proceedings of the 18th International Conference on Very Large Data Bases, pp. 383–394, Vancouver, Canada, August 1992.

[9] H. Lu, B. Ooi and K. Tan: "On Spatially Partitioned Temporal Join," to appear in Proceedings of 1994 International Conference on Very Large Data Bases, Chile, August 1994.

[10] M. H. Overmars: "Efficient Data Structures for Range Searching on a Grid," Journal of Algorithms, Vol. 9, No. 2, pp. 254–275, 1988.

[11] S. Rana and F. Fotouhi: "Efficient Processing of Time–join in Temporal Data Bases," Proceedings of the 3rd International Symposium on Database Systems for Advanced Applications, pp. 427–432, Taejon, Korea, April 1993.

[12] T.J. Sager: "A Polynomial Time Generator for Minimal Perfect Hashing Functions,"Commun. ACM, Vol. 28, No.5, pp.523–532, 1985.

[13] A. Segev and A. Shoshani: "The Representation of a Temporal Data Model in the Relational Environment," Lecture Notes in Computer Science, Vol. 339, M. Rafanelli, J.C. Klensin, and P. Svensson (eds.), Springer–Verlag, pp. 39–61, 1988.

[14] M. Soo, R. Snodgrass, and G. Jenson: "Efficient Evaluation of the Valid–time Natural Join," Proceedings of the Tenth International Conference on Data Engineering, April 1994.

DEP_TRABDG

D#	TRABDG	T_S	T_E
D_1	30	1	4
D_1	40	5	10
D_1	20	11	20
D_2	45	1	15
D_2	10	16	20
D_3	20	1	12
D_3	15	13	20

EMP_DEP

E#	D#	T_S	T_E
E_1	D_3	1	6
E_1	D_2	7	20
E_2	D_1	1	16
E_2	D_2	17	20
E_3	D_3	1	7
E_3	D_1	8	12
E_3	D_2	13	20

Table 2.1 Example of two temporal relations

E#	D#	D#	TRABDG	T_S	T_E
E_1	D_3	D_1	40	5	6
E_1	D_2	D_1	40	7	10
E_2	D_1	D_1	40	5	10
E_3	D_1	D_1	40	5	7
E_3	D_1	D_1	40	8	10
E_1	D_3	D_2	45	1	6
E_1	D_2	D_2	45	7	15
E_2	D_1	D_2	45	1	15
E_3	D_1	D_2	45	1	7
E_3	D_1	D_2	45	8	12
E_3	D_2	D_2	45	13	15

Table 2.2 Result of a temporal join for answering query Q_1

D#	TRABDG	E#	D#	T_S	T_E
D_1	40	E_3	D_1	8	10
D_1	20	E_3	D_1	11	12
D_2	45	E_3	D_1	8	12
D_3	20	E_3	D_1	8	12

Table 2.3 Result of a temporal join for answering query Q_2

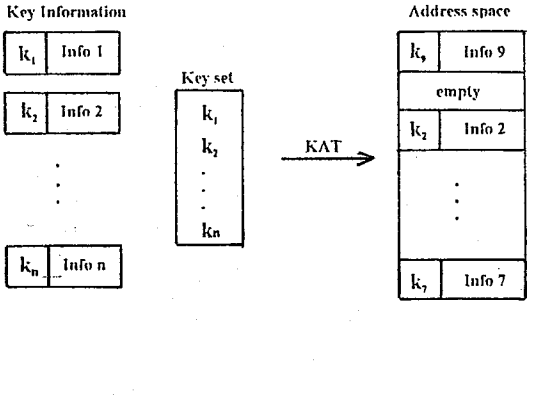

Figure 3.1 The KAT (key to address transformation)

Key k_i	Bit-string (b_1 b_2 ... b_5)	Address t (b_1 b_2 ... b_5)
29	11101	9
10	01010	2
17	10001	6
12	01100	4
11	01011	3
23	10111	7
24	11000	8
14	01110	5
6	00110	1
31	11111	10

Table 3.1 Address location computed by rank KAT

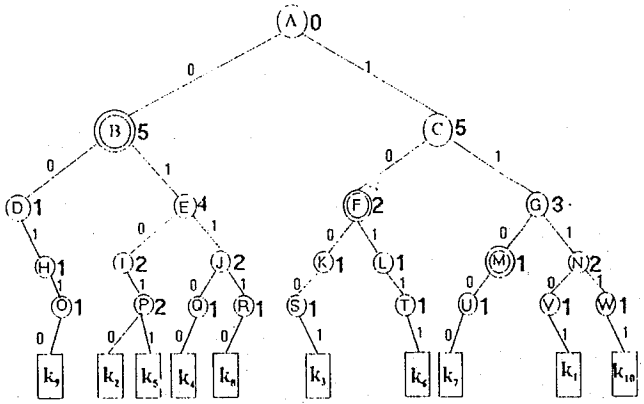

Figure 3.2 Binary tree corresponding to the keys in Example 3.1

Distinct endpoints	Ranks	Associated information sets INFO
1	1	{ (U,1,S), (U,4,S), (U,6,S), (V,1,S), (V,3,S), (V,5,S) }
4	2	{ (U,1,E) }
5	3	{ (U,2,S), (V,3,I) }
6	4	{ (V,1,E) }
7	5	{ (V,2,S), (V,5,E) }
8	6	{ (V,6,S) }
10	7	{ (U,2,E) }
11	8	{ (U,3,S) }
12	9	{ (U,6,E), (V,6,E), (U,4,I) }
13	10	{ (U,7,S), (V,7,S) }
15	11	{ (U,4,E) }
16	12	{ (U,5,S), (V,3,E) }
17	13	{ (V,4,S) }
20	14	{ (U,3,E), (U,5,E), (U,7,E), (V,2,E), (V,4,E), (V,7,E) }

Table 4.1 Distinct values of endpoints in Example 4.1 and their associated rank KAT values and information sets