Proceedings of the 2009 IEEE International Conference on Systems, Man, and Cybernetics
San Antonio, TX, USA - October 2009

Evaluation the Algorithms of Fuzzy Temporal Databases Join Operations

Liguo DENG[1,2], Z.M. MA[2],Gang ZHANG[3]
([1]College of Software , Shenyang Normal University, Shenyang, 110034, China)
([2]College of Information Science & Engineering, Northeastern University, Shenyang 110004,China)
([3]College of Software , Shenyang University of Technology, Shenyang, 110023, China)
E-mail:liguo_deng@163.com, *mazongmin@ise.neu.edu.cn*, *deepblue848@sina.com*

Abstract

In many applications data values are inherently uncertain. This includes moving-objects, sensors and biological databases here has been recent interest in the development of database management systems that can handle uncertain data. Some proposals for such systems include attribute values that are uncertain. In particular, an attribute value can be modeled as a range of possible values, associated with a fuzzy density function. Joins are arguably the most important relational operators. Poor implementations are tantamount to computing the Cartesian product of that join multiple relations have not been addressed in earlier work despite the significance of joins in fuzzy temporal databases. In this paper we address join over uncertain data. We propose a semantics for the join operation, define fuzzy operators over uncertain data, and propose join algorithms that provide efficient execution of fuzzy joins. The paper focuses on an important class of joins termed inequality predicates prevalent that avoid some of the semantic complexities of dealing with uncertain data. These techniques facilitate pruning with little space and time overhead, and are easily adapted to most join algorithms. We address this need for efficient join evaluation in fuzzy temporal databases. Our purpose is that we first survy previously proposed fuzzy temporal join operators and we then address evaluation algorithms, comparing the applicability of various algorithms to the fuzzy temporal join operators and describing a performance study involving algorithms for important operator.

1. Introduction

While traditionally, databases have required data to be modeled in terms of precise values, there are many applications where uncertainty, or imprecision in values is inherent or desirable [1,2,3,4]. In this model, called the dead-reckoning apporach [5,6], the value of the sensor is correctly modeled as a range around the last reported value. Finally, in a Location-Based Service application, users may wish to provide approximate, imprecise locations in order to preserve their privacy. Given the need for managing uncertain data, several researchers have recently proposed the development of database systems that manage uncertain data [1,2,3,7]. There are two broad types of data uncertainty that is defined in these works: tuple uncertainty and attribute uncertainty. Tuple uncertainty refers to the uncertainty that a given tuple is part of a database [1]. The tuple itself is no different than regular database tuples. Attribute uncertainty refers to the uncertainty in the value of a given attribute [3,7].

In general, uncertainty occurs in any database system that attempts to model and capture the state of the physical world, where entities being monitored such as pressure, temperature, locations of moving objects, are constantly changing. As pointed out in [8], if the received sensor value is directly used to answer queries, erroneous answers may result. In order to alleviate this problem, the idea of incorporating uncertainty information into the sensor data has been proposed recently. Instead of storing single values, each data item is modeled as a range of possible values, associated with a fuzzy temporal density function[8]. Incorporating uncertainty into databases brings about many challenges including issues of query semantics, evaluation, and efficiency. The problem of the semantics of query processing and efficient evaluation of queries for tuple uncertainty have been discussed in earlier work[8,9]. There has also been some work on simple types of queries for databases with attribute uncertainty [10,11]. In this paper, we address the issue of joins over databases with uncertain attributes.

In this paper, we consider data model of fuzzy temporal query processing, the evaluation ad hoc fuzzy temporal join operations, i.e.. Joins are arguably the most important relational operations. This is so because efficient join processing is essential for the overall efficiency of a query processor. Joins occur

978-1-4244-2794-9/09/$25.00 ©2009 IEEE

frequently due to database normalization and are potentially expensive to compute. We aims to present a comprehensive and systematic study of join operations in fuzzy temporal databases, including both semantic and comparing the applicability of various algorithms.

We start this paper with a brief overview of fuzzy temporal databases in Section 2. We then define fuzzy temporal join operators. We show that this calculus, which is similar to a taxonomy of temporal join operators, has extends well-established relational operators to the fuzzy temporal context and classifies all previously defined fuzzy temporal operators. In Section 3 We compare our work with related work and finally conclude with directions for further research of evaluation join algorithms. Finally, conclusion and directions for future work are offered in section 4.

2. Fuzzy temporal join operators

We concentrate on the modeling of two kinds of temporal propositions: events and facts. Facts are used to capture static aspects of the world while events are used to capture dynamic aspects. Events can be instantaneous or have a duration. In the latter case the time over which an event occurs is maximal: an event cannot occur over two intervals one of which contains the other[12].

2.1 Fuzzy temporal DB model

To make statements about when certain events occurred, or when certain facts were/will be true, FT-databases use a calendar and temporal constraints.

Calendar:Our approach is closely related to [10] who considered translating formulas in the first order theory of the structure $(\mathbb{R},<)$. Where $\mathbb{R}$ is isomorphic to the set of time rational numbers.

Temporal Constraint:A temporal constraint C over calendar H is defined inductively in the following way. If C_1 and C_2 are temporal constraints over calendar $(\mathbb{R},<)$, then $(C_1 \wedge C_2)$,$(C_1 \wedge C_2)$, and $(\neg C_1)$ are temporal constraints over $(\mathbb{R},<)$.

Fuzzy Degree Function:Consider a simple statement saying that data tuple t is in relation r at some time point in a set with fuzzy degree a value. Suppose we are now asked "what the fuzzy value that t is in r at some point?" Let ts be a temporal constraint over calendar $(\mathbb{R},<)$ such that $|SL(ts)| \geq 1$. Then a fuzzy degree function (FDF) over calendar $\mathbb{R}$, denoted FDF(ts,t_p), is a function which takes ts and a time point $t_p \in ST$ as input, and returns as output a degree δ_p which satisfies each $t_p \in ST$ and $0 \leq \delta_p \leq 1$. For all $t_p \in ST$ where $t_p \notin SL(ts)$, $\delta_p = 0$ and $\sum_{tp \in ST}(\delta_p) \leq 1$. If the sum $\sum_{tp \in ST}(\delta_p)$ is strictly less than one, the function is called a partial FDF; if the sum is exactly equal to one, the function is called a complete FDF. A FDF is determinate if $\sum_{tp \in ST}(\delta_p)$ is computable in constant time.

FT-*tuple*(*ftt*):Intuitively, a fuzzy temporal relation is a multiset of fuzzy temporal tuples(FT-tuples). Each FT-tuple consists of a "data" part and a fuzzy temporal part.Let ts be a temporal constraint over $\mathbb{R}$ where $|SL(ts)| \geq 1$. Furthermore, let $\eta,\xi \in [0,1]$ be fuzzy degree where $\eta \leq \xi$, and let μ be a fuzzy degree function over $\mathbb{R}$. Then the quadruple $\langle ts, \eta, \xi, \mu \rangle$ is called a fuzzy value tuples or *FP-tuple*.

FT-*relation*(*ftr*): A FT-relation statement over calendar $\mathbb{R}$, denoted λ, is an expression of the form $\langle C_i, ts_i, \eta_i, \xi_i, \mu_i \rangle$ where $i \geq 1$, C_i and ts_i are a temporal constraints over $\mathbb{R}$, η_i and ξ_i are fuzzy value degrees, μ_i is a distribution function over $\mathbb{R}$.

Fuzzy Value Degree Strategies:Given the fuzzy value degrees d_1 and d_2 of events E_1 and E_2, how do we compute the degree d of compound event $(E_1 \wedge E_2)$? The answer depends on the relationship between E_1 and E_2. For instance if E_1 and E_2 are mutually exclusive, d should be zero; if E_1 and E_2 are independent of each other, d should be $(d_1 \bullet d_2)$. A similar situation arises when computing the probability of $(E_1 \vee E_2)$.

Semantic and Consistency of fuzzy Temporal Relations:FT-tuples FTP=(D, λ) and FTP´=(D', λ') are data-identical iff $(D=D')$. Note that FTP and FTP´ may come from different FT-relations as long as both FT-relations have the same schema.

2.2 Example

For instance, consider the table1 FT-tuple.

Table1 FT-tuple

Item	Origin	Dest	C	ts		
001	Beijing	Tokyo		4/2/1999	0.4	1
002	Beijing	Tokyo		2/5/1999	0.5	1

It is easy to see that the upper bounds of the FT-cases above can be tightened to 0.5 and 0.4 respectively. Hence, there upper bounds are "loose" and need to be tightened if a definition similar to definition15 is to hold. Let $\lambda = \{\lambda_1, \ldots, \lambda_n\}$ be a FT-cases statement where each FT-cases $\lambda_i = \langle C_i, ts_i, \eta_i, \xi_i, \mu_i \rangle$. A tightening of λ returns a FT-cases statement $\lambda'' = \{\lambda_1'', \ldots, \lambda_n''\}$ where each $\lambda_n'' = \langle C_i'', ts_i'', \eta_i'', \xi_i'', \mu_i'' \rangle$ and each $\xi_i'' \leq \xi_i$ for all $1 \leq i \leq n$. Let λ consist of one FT-tuple which contains two FT-cases as shown table2.

table2 FT-tuple which contains two FT-cases

Item	Origin	Dest	C	ts	η	ξ	μ

001	Beijing	Tokyo	day≤2∧month =4∧year=1999	0.4	0.7	φ_1
002	Beijing	Tokyo	day≤5∧month =2∧year=1999	0.5	0.8	φ_2

2.3 Equijoin

The fuzzy temporal equijoin operator enforces equality matching among specified subsets of the explicit attributes of the input relations. Suppose $R=(A_1,\ldots,A_k)$, $R'=(A'_1,\ldots,A'_k)$, r is a FT-relation over R, $\mathbb{R}$, r', is a FT-relation over R', $\mathbb{R}$, α is a fuzzy conjunction strategy, CPF is a Cartesian product function, and *ftm* is a FT-map function. Then the join of r and r' under α using of CPF, denoted $r \bowtie_{\alpha}^{CPF,ftm}{}_{r[A]=r'[A']} r'$, produces FT-relation r'' over R'', where R''=(Fd), $r''=\pi Fd(\sigma_c^{ftm}(r\times_{\alpha}^{CPF} r'))$.

2.4 Natural join

The fuzzy temporal natural join and the temporal equijoin bear the same relationship to one another as their counterparts. That is, the temporal natural join is simply a fuzzy temporal equijoin on identically named explicit attributes followed by a subsequent projection operation. A *FT*-relation statement over calendar $\mathbb{R}$, denoted λ, is an expression of the form $\langle C_i, te_i, \eta_i, \xi_i, \mu_i\rangle$ where $i \geq 1$, C_i and te_i are a temporal constraints over $\mathbb{R}$, η_i and ξ_i are fuzzy value degrees, μ_i is a distribution function over $\mathbb{R}$, and the following conditions are satisfied for postulates:

▪$(0 \leq \eta_i \leq \xi_i \leq 1)$.

▪ $SL(C_i) \subseteq SL(te_i)$. This ensures that $\mu_i(te_i, te_i)$ is de -fined for each time point $te_i \in SL(C_i)$.

▪ $SL(C_i) \geq 1$. In other words, C_i and te_i each have at least one solution in *ST*.

▪For all $0 \leq j \leq 1$, $i \neq j \Rightarrow SL(C_i) \cap SL(C_j) = \varnothing$.

For each $1 \leq i \leq n$, $\lambda_i=\langle C_i, te_i, \eta_i, \xi_i, \mu_i\rangle$ is called a FT-relation statement of λ. On occasion, we may want to assign fuzzy value degrees to every time point in *ST*.

▪Fd is the attribute list $A_1,\ldots,A_k$, $R(a_{n+1},\ldots,a_{k'})$ where $\{a_{n+1},\ldots,a_{k'}\}=\{a\in R' | a\notin R\}$ and $a_{n+1},\ldots,a_{k'}$ is a sublist of $A'_1,\ldots,A'_k$.

▪*C* is the selection condition $(a_1=R(a_1)\wedge\ldots\wedge a_n=R(a_n))$ where $\{a_1,\ldots,a_n\}=\{a\in R' | a\in R\}$.

$r \bowtie_{\alpha}^{CPF,ftm} r'=\{\langle z, (t),\eta,\xi,\rangle | t\in SL(\lambda'.C\wedge\lambda.C)\wedge[\eta,\xi]= ftr(\lambda,t)\otimes ftr(\lambda',t)\wedge(\exists x\in r,\exists y\in r'(x[A]=y[A']))\}$.

Since $r \bowtie_{\alpha}^{CPF,ftm} r' \equiv r \bowtie_{\alpha}^{CPF,ftm1} r'$ must hold, we let $r \bowtie_{\alpha} r'$ denote $r \bowtie_{\alpha}^{CPF,ftm} r'$ with any choice for CPF and *ftm*. Now, let *ftr*(*r*) denote $\sum_{(D,Tc)\in r}(|T_c|)$, i.e., the total number of FT-tuples in *r*. It is easy to see that $ftr(r \bowtie_{\alpha} r')$ can be huge.

We showed in earlier work that the fuzzy temporal natural join plays the same important role in reconstructing normalized fuzzy temporal relations. Most previous work in fuzzy temporal join evaluation has addressed, either implicitly or explicitly, the implementation of the fuzzy temporal natural join or the closely related fuzzy temporal natural join or the closely related fuzzy temporal equijoin.

2.5 Outerjoins

Like the temporal outerjoin, fuzzy temporal outerjoins and Cartesian products retain that tuples do not participate in the join. However, in a fuzzy temporal database, a tuple may dangle over a portion of its fuzzy temporal interval and be covered over others; this situation must be accounted for in a fuzzy temporal outerjoin or Cartesian product. We may define the fuzzy temporal outerjoin as the union of two subjoins, like the temporal outerjoin. The two subjoins are the fuzzy temporal left outerjoin and the fuzzy temporal right outerjoin. As the left and right outerjoin are symmetric, we define only the left outerjoin.

We need two auxiliary functions: A FT-compression function *ftc*(*r*) is a function that maps FT-relation *r* over *R*, $\mathbb{R}$, to a FT-relation *r* over *R*, $\mathbb{R}$, where (i) $ftr(r')\leq ftr(r)$ and (ii) there exists a maps each $(D,t,\eta_t,\xi_t)\in ftt(r)$ to a matching $(D,t,\eta_t,\xi_t)\in ftt(r')$. Let $|r|$ denote the number of FT-tuples in *r*. Then it must be the case that $ftr(r)\geq|r|$. Note that If μ is a pair of FDF_{fts}, then $ftr(ftc(r))=|ftc(r)|=|r|$ for every FT-relation *r*.

Suppose $M=\{[\eta_1,\xi_1],\ldots,[\eta_m,\xi_n]\}$ is a nonempty multiset of fuzzy intervals and let $[\eta,\xi]=([\eta_1,\xi_1]\cap\ldots\cap[\eta_m,\xi_n])$. Then *F* is a combination function if $F(M)$ return a fuzzy interval $[\eta'',\xi'']$ that satisfies the following conditions:

▪Identity: If $[\eta_1,\xi_1]=\ldots=[\eta_m,\xi_n]$, then $[\eta'',\xi'']=[\eta,\xi]$.

▪Max: $\eta''\leq\max_{[\eta i,\xi i]\in M}(\eta_i)$.

▪Interval conflict: A multiset *M* of fuzzy time intervals conflict iff $\cap_{[\eta,\xi]\in M}[\eta,\xi]=\varnothing$.

▪Interval equity: An equity combination function F_E is a combination function where $\cap_{[\eta,\xi]\in M}[\eta,\xi]=\varnothing\Rightarrow F_E(M)=\cap_{[\eta,\xi]\in M}[\eta,\xi]$.

F is an equity combination function if $[\eta'',\xi'']=[\eta,\xi]$ whenever $[\eta,\xi]\neq\varnothing$.

The fuzzy temporal left outerjoin, $r \overset{CPF,ftm}{\underset{\alpha}{\dashv\bowtie}} r'$, of two fuzzy temporal relations r and r' is defined as follows.

$r \overset{CPF,ftm}{\underset{\alpha}{\dashv\bowtie}} r' = \{\langle z, (t), \eta, \xi, \rangle | \exists x \in ftc(r), \exists y \in ftc(r')\ x[A] = y[A'] \wedge z[A''] = x[A] \wedge y[A'] = z[A'']) \wedge ftr(r') \leq ftr(r) \wedge (D, t, \eta_i, \xi_i) \in ftt(r) \vee ([\eta', \xi'] = [\eta, \xi] \wedge \eta' \leq \max_{[\eta i, \xi i] \in M}(\eta_i) \wedge \cap_{[\eta, \xi] \in M} [\eta, \xi] = \varnothing \Rightarrow F_E(M) = \cap_{[\eta, \xi] \in M}[\eta, \xi])\}$.

Similary, a temporal outer Cartesian product is a fuzzy temporal outerjoin without the equijoin condition $r[A] = r'[A']$.

3. Evaluation join algorithms

Previous work has largely ignored the fact that conventional query evaluation algorithms can be easily modified to evaluate temporal joins. In this section, we show how the three paradigms of query evaluation can support fuzzy temporal join evaluation. We adapt the Sort-merge-based fuzzy temporal algorithms to support fuzzy temporal joins.

3.1 Taxonomy

Because joins are so frequently used in relational queries and because joins are so expensive, much effort has gone into developing efficient join algorithms. The simple nested-loop join is applicable in all cases, but imposes quadratic performance. For equijoins, sort-merge join was found to be much more effective, with excellent performance over a wide range of relation sizes, given adequate main memory. Later, researchers became interested in hash-based join algorithms and it has been shown that in many situations, hash-based algorithms perform better than sort-based algorithms. However, there exist cases in which the performance of hash-based joins falls short. If there are several relations that will participate in multiple joins, the "interesting order" will often determine that sort-based join is better, to enable the joins to run in a pipeline fashion, because the output of sort-merge join is sorted, thereby possibly obviating the need for sorting in subsequent sort-merge joins. Graefe has exposed many dualities between the two types of algorithms and their costs differ mostly by percentages[13]. Most DBMSs now include both sort-merge and hash-based, as well as nested-loop and index-based join algorithms.

3.2 Nested-loop algorithm

The nested block join algorithm is a more efficient version of the nested loop join algorithm which is used in a relation environment. The nested loop algorithm works by reading one tuple from one relation, the outer relation, and passing over each record of the other relation, the inner relation, joining the tuple of the outer relation with all the appropriate records of the inner relation. The next tuple from the outer relation is then read and the whole of the inner relation is again scanned, and so on. The nested block algorithm works by reading a relation of tuples, typically one tuple from the outer relation and passing over each tuple of the inner relation (also read in relation) joining the tuples of the outer relation with those of the inner relation. Note that under Hagmann's analysis [] half the available memory is devoted to pages from the inner relation, and half to the outer relation.A slightly more efficient version of this algorithm can be obtained by rocking backwards and forwards across the inner relation. That is, for the first tuple of the outer relation the inner relation is read forwards and for the second tuple of the outer relation the inner relation is read backwards. This eliminates the need for reading in one set of blocks of the inner relation at the start of each pass, except the first, because the relation will already be in memory. This is the version of the algorithm the following analysis represents. We assume that the memory based part of the operation is based on hashing. That is, the tuples of the outer relation are formed into a hash table, and the records of the inner relation are hashed against this table to find records to join with.

Nested-loop join algorithms match tuples by exhaustively comparing pairs of tuples from the input relations. As an I/O optimization, blocks of the input relations are read into memory, with comparisons performed between all tuples in the input blocks. The size of the input blocks is constrainted by the available main memory buffer space. The algorithm operates as that one relation is designated the outer relation, the other the inner relation. The fuzzy temporal nested-loop join is easily constructed from this basic algorithm. All that is required is that the timestamp and fuzzy values predicate be evaluated at the same time as the predicate on the explicit attributes. Figure 1 shows the fuzzy temporal algorithm.(In the figure, *Or* is the outer relation and *Ir* is the inner relation.). We assume their schemas are as defined in section 2.

```
NestedLoop(Or, Ir):
    result ← ∅;
      for each block B_or ∈ Or
       read(B_or);
          for each block B_ir ∈ Ir
              read(B_ir);
              for each tuple ftt(Or) ∈ B_or
                  for each tuple ftt(Ir) ∈ B_ir
                    if ftt(Or)[η_i, ξ_i] = ftt(Ir)[η_i, ξ_i] and
                       overlap(ftt(Or)[T], ftt(Ir)[T]) ≠ ∅
                        ftt(r)[A] ← ftt(Or)[A];
                        ftt(r)[B] ← ftt(Ir)[B];
```

```
        ftt(r)[C] ← ftt(Or)[C];
        ftt(r)[T]←overlap(ftt(Or)‹T,η_r,ξ_r›,ftt(Ir)‹T,η_r,ξ_r›);
        result ←result ∪ {ftt(r)};
return result;
```

Fig. 1. Algorithm fuzzy temporal NestedLoop

3.3 Sort-merge algorithm

The sort-merge join algorithm works in two phases. In the first, sorting, phase, each relation is sorted on the join attributes. In the second, merging, phase, a tuple(or block of tuples) is read from each relation and they are joined if possible, otherwise the one with the smaller sort order is discarded and a new tuple read from that relation. In this way, each relation is only read once after it has been sorted. The variant of the sort-merge algorithm whose cost we present below is a simplified version of that used in the deductive database system. Instead of completely sorting each relation,each relation is partitioned into sorted partitions that are the size of the memory buffer. To perform this partitioning the pages are read from a relation, sorted, and written out, then another pages, and so on. During the merge phase, the partitions of each relation are merged together and the records from each relation are joined. The same pass over the partitions both merges the partitions of each relation and joins the relations. Up to partitions of both relations may be created prior to the merge phase. In the equations in this section, P_{ro} and P_{ri} represent the number of pages allocated to each partition of relations R_{ro} and R_{ri} respectively.

In this paper, sort merge join algorithm consist of two phases. In the first phase, the input relations *Or* and *Ir* are sorted by their join attributes. In the second phase, the result is produced by simultaneously scanning *Or* and *Ir*, merging tuples with identical values for their join attributes.Complications arise if the join attributes are not key attributes of the input relations. In this case, multiple tuples in *Or* and in *Ir* may have identical join attribute values. Hence a given *Or* tuple may join with many *Ir* tuples, and vice versa.We designate one relation as the outer relation and the other as the inner relation. When consecutive tuples in the outer relation have identical values for their explicit join attributes, i.e., their nontimestamp join attributes, the scan of the inner relation is backed up to ensure that all possible matches are found. Prior to showing the SortMerge algorithm, we define a suite of algorithms that manage the scans of the input relations. For each scan, we maintain the state structure shown in Fig.2. The fields *Timestamp_Block* and *Timestamp_Tuple* together indicate the current block and index of the current tuple within that block. The *first_block* and *first_tuple* are used to record the state at the beginning of a scan of the inner relation in order to back up the scan later if needed. Finally, tuples stores the block of relation the timestamp in memory. For conveniece, we treat the block as an array of tuples. Essentially, counters are set to guarantee that the first block read and the first tuple scaned are the first block and first tuple within that block in the input relation. We assume that a operation is available that repositions the file pointer associated with a relation to a given block number. The algorithm *Advances_Scan* the scan of the argument relation and state to the next tuple in the sorted relation. The state is updated to mark the next tuple in the current block as the next tuple in the scan. The *Timestamp_Tuple* algorithm merely returns the next tuple in the scan, as indicated by the scan state. The backup and *TimeScan_Start* algorithms manage the backing up of the inner relation scan. And the *Scan_Backup* reverts the cuurent block and tuple counters to their last values. The values are stored in the state at the beginning of a scan by the *TimeScan_Start* algorithm. The algorithm accepts four parameters, the input relations *Or* , *Ir* , the join fuzzy value μ and *Timestamp* T. We assume that the schemas of *Or* and *Ir* are scaned in order. For each outer tuple, if the tuple matches the previous outer tuple, the scan of the inner relation is backed up to the first matching inner tuple. The starting location of the scan is recorded in case backing up is needed by the next outer tuple, and the scan proceeds forward as normal. The complexity of the algorithm, as well as its performance degradation as compared with conventional sort-merge, is due largely to the keeping required to back up the inner relation scan.

```
Structure Relation_TimeState
    integer Timestamp_block;
    integer Timestamp_tuple;
    integer Timestamp_first_block;
    integer Timestamp_first_tuple;
block  tuples;
initRelationTimeState(relation, timestate)
      timestate.Timestamp_block←1;
      timestate.Timestamp_tuple ←0;
      timestate. Timestamp_first_block←1;
     timestate. Timestamp_first_tuple ←0;
  timestate.Timestamp_tuples ←read_block(relation);
Advances_Scan(relation,timestate)
    if ( timestate.Timestamp_tuple=bottom_tuples )
        timestate.Timestamp_tuple← read_block(relation);
        timestate.Timestamp_block ← ++1;
        timestate.Timestamp_tuple ← 1;
    else
        timestate.Timestamp_tuple ← ++1;
Scan_Backup(relation,timestate)
   If(timestate.Timestamp_block ≠ timestate.
Timestamp_first_block)
     timestate.Timestamp_block ← timestate.
Timestamp_first_block;
     timestate.Timestamp_tuple ←read_block(relation);
```

```
    timestate.tuple ← timestate. Timestamp_first_tuple;
    timestate. first_block = timestate. Timestamp_first_block;
    timestate. first_tuple = timestate. Timestamp_first_tuple;
FuzzyTemporalSortMerge(Or, Ir, μ,T)
      Or′← sort(Or, μ,T);
      Ir′ ← sort(Ir, μ,T);
      InitTimeState(Or′,outer_timestate);
      InitTimeState(Ir′,inner_timestate)
    ftt(Or)′ ← block_bottom;
result ←∅;
    Advances_Scan(Ir′,inner_timestate);
   ftt(Ir) ← inner_timestate.Timestamp_tuple;
for (i=1,i,i=|ftt(r)′|)
        Advances_Scan(Or′,outer_timestate);
       ftt(Or) ← outer_timestate.Timestamp_tuple;
    if ftt(Or)[μ]= ftt(Or)′[μ];
      Scan_Backup(Ir′,inner_timestate);
            ftt(Ir) ← Timestamp_tuple(Ir′,inner_timestate);
            ftt(Or)′[μ] ← ftt(Or)[μ];
    while (ftt(Or)[μ] > ftt(Ir)[μ])
                Advances_Scan(Ir′,inner_timestate);
                ftt(Ir) ← inner_timestate.Timestamp_tuple;
                RecordScanStart(inner_timestate);
    while (ftt(Or)[μ] = ftt(Ir)[μ])
                if overlap(ftt(Or)[T], ftt(Ir)[T])≠∅
                  ftt(r)[A, μ] ← ftt(Or)[A, μ];
                  ftt(r)[B, μ] ← ftt(Ir)[B, μ];
                  ftt(r)[C, μ] ← ftt(Or)[C, μ];
                  ftt(r)[T] ← overlap(ftt(Or)‹T,η,,ξ,›,
ftt(Ir)‹T,η,,ξ,›);
                   Advances_Scan(Ir′,inner_timestate);
                ftt(Ir) ← Timestamp_tuple(Ir′,inner_timestate);
return result;
```

Fig. 2. Algorithm fuzzy temporal Sort-merge

For every join, we focus in the sort-merge based algorithm that assumes that the input relations are sorted primarily by their surrogate attributes and secondarily by their starting fuzzy timestamps. The result is produced by a single scan of both input relations.

4. Conclusions

The paper developed evaluation strategies for the fuzzy temporal join and compared the evaluation strategies in a sequence of empirical performance studies. By the taxonomy that it classifies the temporal join operators, we extended the conventional join operators, irrespective of special joining attributes or other model restrictions. We extended the two main paradigms of query evaluation algorithms to fuzzy temporal databases, thereby defining the space of possible fuzzy temporal evaluation algorithms, and representing the space of such possible algorithms that placed the existing work into this framework. The salient point of the extended is that simple modifications to an existing conventional evaluation algorithm can be used to effect fuzzy temporal joins at relatively small development cost.

References

[1] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *Proc. VLDB*, 2004.

[2] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *Proc. VLDB*,2004.

[3] J. Widom. Trio:A system for integrated management of data,accuracy,and lineage. In *Proc. CIDR*, 2005

[4] A. Yazici, A. Soysal, B. Buckles, and F. Petry. Uncertainty in a nested relational database model.*Elsevier Data and Knowledge Engineering*, 30, 1999.

[5] D.Pfoser and C. Jensen. Capturing the uncertainty of moving-objects representations. In *Proc. SSDBM*, 1999.

[6] J. Enderle, M. Hampel, and T. Seidl. Joining interval data in relational databases. In *Proc.SIGMOD*. 2004

[7] R. Cheng, D. Kalashnikov, and S. Prabhakar.Evaluating probabilistic queries over imprecise data. In *Proc. SIGMOD*, 2003.

[8] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. Q-uerying imprecise data in moving object environments. *IEEE Transactions on Knowledge and Data Engineerin-g (To-appear)*, 2004.

[9] R. Cheng and S. Prabhakar. Managing uncertainty i-n sensor databases. In *SIGMOD Record-issue on Sens-or Technology*, December 2003.

[10] H. Kriegel, M. Potke, and T. Seidl. Managing inte-rvals efficiently in object-relational databases. In *Proc. of the 26th Intl. Conf. on VLDB*, Cairo, Egypt,2000

[11] B. Lin, H. Mokhtar, R. Pelaez-Aguilera, and J. Su. Querying moving objects with uncertainty. In *Proceedi-ngs of IEEE Semiannual Vehicular Technology Confe-rence, 2003.*

[12]D.V. McDerott. A Temporal Logic for Reasoning About Processes and Plans. *Cognitive Science*,*1982,*P 101-155

[13]Michael D. Soo, Richard T. Snodgrass and C. S. Jensen, "Efficient Evaluation of the Valid-Time Natural Join," in *Proceedings of the International Conference on Data Engineering*,Houston, TX, February, 1994, pp. 282–292.