01-EP.
f-papers

IEEE TRANSACTIONS ON

KNOWLEDGE AND DATA ENGINEERING

A publication of the IEEE Computer Society

NOVEMBER / DECEMBER 2001 VOLUME 13 NUMBER 6 ITKEEH (ISSN 1041-4347)

REGULAR PAPERS

Data Mining

Mining Associations with the Collective Strength Approach
C.C. Aggarwal and P.S. Yu 863

Data Modeling

Accurate Modeling of Region Data
G. Proietti and C. Faloutsos 874

Fuzzy Databases

Efficient Processing of Nested Fuzzy SQL Queries in a Fuzzy Database
Q. Yang, W. Zhang, C. Liu, J. Wu, C. Yu, H. Nakajima, and N.D. Rishe 884

Imprecise Relational Databases

Aggregation of Imprecise and Uncertain Information in Databases
S. McClean, B. Scotney, and M. Shapcott 902

Knowledge-Based Systems

Structured Development of Problem Solving Methods
D. Fensel and E. Motta 913

Multimedia Information Systems

Disk Scheduling in Video Editing Systems
W.G. Aref, I. Kamel, and S. Ghandeharizadeh 933

Virtual Images for Similarity Retrieval in Image Databases
G. Petraglia, M. Sebillo, M. Tucci, and G. Tortora 951

Nonmonotonic Reasoning

Nonmonotonic Reasoning as Prioritized Argumentation
J.-H. You, X. Wang, and L.-Y. Yuan 968

Query Processing

Rewriting Queries Using Views
S. Flesca and S. Greco 980

Robust Databases

Transaction Repair for Integrity Enforcement
L.V. Orman 996

Security and Privacy

Protecting Respondents' Identities in Microdata Release
P. Samarati 1010

SHORT PAPERS

Data Structures and Algorithms

Time-Space Trade-Off Analysis of Morphic Trie Images
R.W.P. Luk 1028

Multimedia Information Processing

Spatio-Temporal Composition of Video Objects: Representation and Querying in Video Database Systems
N. Pissinou, I. Radev, K. Makki, and W.J. Campbell 1033

Index 1041

Call for Papers for Special Issue on Online Analysis and Querying of Continuous Data Streams 1052

Call for Papers for Transactions on Mobile Computing 1053

IEEE COMPUTER SOCIETY

http://computer.org • tkde@computer.org

Efficient Processing of Nested Fuzzy SQL Queries in a Fuzzy Database

Qi Yang, Weining Zhang, *Member, IEEE, Computer Society*, Chengwen Liu, *Member, IEEE*, Jing Wu, Clement Yu, *Member, IEEE, Computer Society*, Hiroshi Nakajima, *Member, IEEE, Computer Society*, and Naphtali David Rishe

Abstract—In a fuzzy relational database where a relation is a fuzzy set of tuples and ill-known data are represented by possibility distributions, nested fuzzy queries can be expressed in the Fuzzy SQL language, as defined in [25], [23]. Although it provides a very convenient way for users to express complex queries, a nested fuzzy query may be very inefficient to process with the naive evaluation method based on its semantics. In conventional databases, nested queries are unnested to improve the efficiency of their evaluation. In this paper, we extend the unnesting techniques to process several types of nested fuzzy queries. An extended merge-join is used to evaluate the unnested fuzzy queries. As shown by both theoretical analysis and experimental results, the unnesting techniques with the extended merge-join significantly improve the performance of evaluating nested fuzzy queries.

Index Terms—Fuzzy database, fuzzy SQL, nested fuzzy query, query optimization, query transformation, possibility distribution, performance evaluation,fuzzy equijoin.

1 Introduction

In order to extend the applicability of traditional databases, some new techniques have been proposed to deal with uncertain or imprecise information [7], [17], [41], [1], [13], [12]. One interesting area of research is the fuzzy database [28], [29], [30], [4], [20], [32], [43], which results from combining the fuzzy set theory [39] with database technology. Several approaches have been taken to define fuzzy relational data models. In one approach [16], [3], a relation is defined as a fuzzy set of crisp tuples. In such a relation, the attribute values in tuples remain crisp, but each tuple is assigned a membership degree, in the range of $[0, 1]$, to indicate the relevancy of the tuple with respect to the relation. In another approach [28], [29], [30], [10], a relation is defined as an ordinary set of fuzzy tuples. Here, a fuzzy tuple may have uncertain or imprecise attribute values, represented by possibility distributions, but, no membership degree is associated with the tuples. In a third approach [3], [23], [5], a relation is defined as a fuzzy set of fuzzy tuples. Thus, each tuple has a membership degree and fuzzy attribute values are represented by possibility distributions. The last approach is more natural than the previous two and is taken in this paper.

A fuzzy relational database system has been built by Omron Corporation. This database system supports an extended SQL query language, named Fuzzy SQL, as defined in [25], [23]. More on the data model and Fuzzy SQL language will be given in the next section. In this fuzzy database, a query may be vague and the data may be ill-known. Unlike standard SQL queries, for which an answer is a relation where each tuple completely satisfies the query condition, the answer to a Fuzzy SQL query is a fuzzy relation where each tuple satisfies the query condition to the extent as indicated by its membership degree.[1] The fact that the answer to a query is a fuzzy relation may greatly affect the query processing in a fuzzy relational database. A common strategy of query optimization is to decompose a complex query, such as a nested query, into subqueries and to store the answer to a subquery in an intermediate relation for subsequent evaluation. For intermediate fuzzy relations, the membership degrees of tuples must be maintained from one processing stage to the next.

In standard SQL, nested query is an important mechanism to ease the pain of expressing complex queries. However, a naive execution of a nested query may incur heavy performance penalty. A common technique to evaluate a nested standard SQL query is to transform (or unnest) the query into an equivalent flat query and then to

- *Q. Yang is with the Department of Computer Science, University of Wisconsin at Platteville, Platteville, WI 53818. E-mail: Yangq@UWPLATT.EDU.*
- *W. Zhang is with the Department of Computer Science, University of Texas at San Antonio, 6900 North Loop, 1604 West San Antonio, TX 78249-0667. E-mail: wzhang@cs.utsa.edu.*
- *C. Liu is with the School of Computer Science, Telecommunicaion, and Information Science, DePaul University, Chicago, IL 60604. E-mail: liu@cs.depaul.edu.*
- *J. Wu and C. Yu are with the Department of Electrical Engineering and Computer Science at the University of Illinois at Chicago, Chicago, IL 60607-7053. E-mail: yu@eecs.uic.edu.*
- *H. Nakajima is with the OMRON Corp., Verbal Interaction Technology Lab., Information Technology Research Center, Shimokaiinji, Nagaokakyo-City, Kyoto, 617-8510 Japan. E-mail: Hiroshi_Nakajima@omron.co.jp.*
- *N.D. Rishe is with the High Performance Database Research Center, School of Computer Science, Florida International University, University Park, Miami, FL 33199. E-mail: rishe@fiu.edu.*

Manuscript received 28 Jan. 1997; revised 28 May 1998; accepted 25 May 2000; posted to Digital Library 12 June 2001.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 103679.

[1] In the usual possibility framework, the answer to a fuzzy query consists of two fuzzy relations, one containing tuples that possibly satisfy the query and the other containing tuples that certainly satisfy the query.

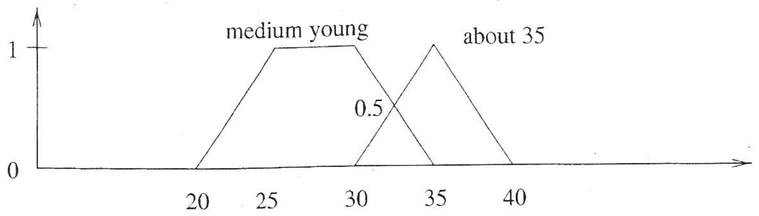


Fig. 1. Membership functions of "medium young" and "about 35."

evaluate the flat query. This unnesting technique has been studied extensively in the context of conventional relational database systems [18], [19], [15], [8], [22]. The key to the success of unnesting is that the unnested queries are evaluated using various join algorithms which are much more efficient than the nested-loop algorithm used to evaluate the nested queries.

In this paper, we investigate the problem of processing nested Fuzzy SQL queries by means of unnesting. Nested query is an important mechanism in Fuzzy SQL as well and its efficient execution is a more important issue in Fuzzy SQL than in standard SQL. Since ill-known data needs more storage space than crisp data does, it takes more I/O time to transfer ill-known data between main memory and secondary memory than does crisp data. Furthermore, since fuzzy queries require non-Boolean degrees of satisfaction to be computed, it takes more CPU time to evaluate a fuzzy query condition than does a crisp query condition. Thus, the study of unnesting techniques is both interesting and crucial to the practical use of fuzzy database systems. In this paper, we extend and augment unnesting techniques of conventional relational databases to process nested queries in fuzzy relational databases. As far as we are aware, this issue has not been studied before.

In conventional relational databases, a join can be processed using different methods, such as hash-join, merge-join, and nested loop join [14]. The most efficient join method, hash-join, is based on the fact that two tuples join only if they have identical values on join attributes. However, this join criteria is no longer sufficient in fuzzy relational databases. For instance, given a join condition $M.AGE = F.AGE$, a pair of tuples from relations M and F may partially join even if one has AGE "young" and the other has AGE" about 35." Intuitively, it is possible in reality for two persons whose ages are vaguely known as "young" and "about 35," respectively, to actually have the same age. Thus, in general hash-join is not applicable in fuzzy relational databases. Although the nested loop join method is always applicable in a fuzzy database, the cost, both I/O and CPU, will be rather high. In order to perform efficient fuzzy joins, we extend the merge-join method. (A complete comparison of different fuzzy join methods is beyond the scope of this paper.) We define a linear order on the domain of a fuzzy attribute and show that under some reasonable assumptions, the extended merge-join performs much more efficiently than the nested loop join method.

We have implemented the extended merge-join method on our fuzzy database system. Experiments have been conducted to compare the performances of the extended merge-join method with that of the nested loop join method. The experimental results show that the extended merge-join method outperforms, in both I/O and CPU, the nested loop method by a wide margin. A further optimization of the merge-join is presented in [42]. We note that our discussion in this paper is based on a fuzzy database system in which the satisfaction degree of query conditions is measured exclusively by possibility, rather than by both possibility and necessity as in [28], [30]. As discussed in Section 2, one of the reasons for not using the double-measure system is that in this system, algebraic operations can not be composed, therefore, unnesting is not possible.

The rest of the paper is organized as follows: In Section 2, we present some background of fuzzy databases. In Section 3, we extend the merge-join method for fuzzy equi-join and analyze the complexity of the response time. In Sections 4 to 7, we discuss techniques for unnesting various types of 2-level nested Fuzzy SQL queries. In Section 8, we discuss a subclass of K-level nested fuzzy queries. We present experimental results in Section 9 and conclude the paper in Section 10.

2 Fuzzy Databases

2.1 Fuzzy Sets and the Theory of Possibilities

Fuzzy sets are defined on a nonfuzzy *universe of discourse*, which is an ordinary set. A fuzzy (sub)set F of a universe of discourse U is characterized by a membership function $\mu_F()$ which assigns to every element $x \in U$, a membership degree $\mu_F(x) \in [0, 1]$. An element $x \in U$ is said to be in a fuzzy set F if and only if $\mu_F(x) > 0$ and to be a full member if and only if $\mu_F(x) = 1$. Fig. 1 shows membership functions of fuzzy sets "medium young" and "about 35," defined on the universe of age. The fuzzy set "medium young" contains as a full member any age between 25 and 30, and as a partial member, the ages 24 and 31 with membership degree 0.8, 23, and 32 with membership degree 0.6, etc. Any age less than 20 or more than 35 is not a member of "medium young" at all.

In the possibility theory [40], the possible values of an ill-known data is described (or restricted) by a fuzzy set. For example, suppose the age of a person is not known precisely but can be described as "medium young." Then, the person's actual age is restricted to be one of the members of "medium young." Thus, the possibility for the age to be 25 is 1, to be 24 is 0.8, etc. Thus, the membership function of "medium young" defines a possibility distribution of the person's age. For this reason, a possibility distribution can be denoted by either a fuzzy set or its membership function. In this paper, we consider only those possibility distributions that have trapezoidal shapes because they are typical in practice.

Note that triangular and rectangular shapes are special cases of trapezoidal shapes.

2.2 A Fuzzy Relational Database

In this section, we present briefly the fuzzy relation and the Fuzzy SQL language as described in [25], [23]. Interested readers may refer to the reference for details.

In the rest of this paper, we denote relations with upper case letters such as R, S, and T, tuples with lower case letters such as r, s, and t, and the attribute A of the relation R (respectively, tuple r) with $R.A$ (respectively, $r.A$). When it is appropriate, indexing may be used on these letters.

Each attribute has a crisp set of crisp data values as its *domain*. Each data value v of an attribute is associated with a possibility distribution defined over the domain of the attribute and has a membership function denoted by μ_v. If the data value is crisp, its possibility distribution is defined by

$$\mu_v(x) = \begin{cases} 1, & \text{if } x = v, \\ 0, & \text{otherwise.} \end{cases}$$

Let $\mathcal{P}(A)$ denote the set of all possibility distributions that may be defined over the domain of an attribute A. A fuzzy relation R with a schema $A_1, \ldots, A_n$, where A_i is an attribute, is defined as

$$R = \mathcal{P}(A_1) \times \mathcal{P}(A_2) \times \cdots \times \mathcal{P}(A_n) \times D,$$

where D is a system-supplied attribute for membership degree with a domain $[0, 1]$ and × denotes the cross product. A tuple r is said to be in relation R if and only if its membership degree $\mu_R(r)$ (which is also denoted as $r.D$) is greater than 0. In general, a tuple's membership degree indicates to what extent the tuple belongs to the concept represented by the relation. More specifically, for a query, the membership degree of a tuple in the answer indicates to what extent the tuple satisfies the query condition.

Like standard SQL, queries in Fuzzy SQL are specified in SELECT statement of the following form:

```
SELECT    Attributes
FROM      Relations
WHERE     Selection condition
```

For this paper, the selection condition is assumed to be a conjunction of predicates of the form $X \ \theta \ Y$, where X is an attribute, Y is an attribute or a value, and θ is an comparison operator. An optional WITH clause of the form WITH $D \geq x$, where D is the membership degree attribute of the answer relation and x is a threshold value in $[0, 1]$, can be used to indicate that, among all tuples resulting from evaluating the query, only those satisfying the query condition with a degree no less than x should be included in the answer. If a query does not have a WITH clause, WITH $D > 0$ is assumed. The GROUPBY and HAVING clauses are also optional and are similar to their counterpart in standard SQL. Aggregate functions such as MAX, MIN, etc., quantifiers such as SOME, ALL, etc., and other key words such as DISTINCT are also defined.

As an example, suppose we have two fuzzy relations M and F in a dating service database that contain information about male and female clients, respectively. The following query finds all pairs of male and female persons who are about the same age, and the male person has a more than "medium high" income.

Query 1

```
SELECT    F.NAME, M.NAME
FROM      F, M
WHERE     F.AGE = M.AGE AND
                  M.INCOME> "medium high"
```

Notice that, since both AGE and $INCOME$ may have fuzzy values, all comparisons are fuzzy.

The semantics of a Fuzzy SQL query is defined based on satisfaction degrees of query conditions. Consider a predicate $X \ \theta \ Y$ in a WHERE clause. The satisfaction degree, denoted by $d(X \ \theta \ Y)$, is evaluated for values of X and Y. Let the value of X be U and that of Y be V. Then,

$$d(X \ \theta \ Y) = max_{x,y}(min(\mu_U(x), \mu_V(y), \mu_\theta(x, y))),$$

where x and y are crisp values in the common domain over which U and V are defined. Notice that, in this definition, the comparison θ may be nonbinary, i.e., defined by similarity relations, and the values U and V may be fuzzy or crisp. If U is crisp, V is fuzzy, and θ is binary equality (=), $d(X = Y) = \mu_V(U)$. If both U and V are fuzzy with trapezoidal membership functions and binary equality is considered, then $d(X = Y)$ is the height of the highest intersection point of the two possibility distributions.

For example, in Query 1, if a tuple t in F has an age 24, and a tuple s in M has an age "medium young" with a membership function as defined in Fig. 1, then,

$$d(F.AGE = M.AGE) = \mu_{medium\ young}(24) = 0.8.$$

If $t.AGE$ is "about 35," $d(F.AGE = M.AGE) = 0.5$, as shown in Fig. 1.

For a conjunction of independent simple predicates, say $p = p_1$ AND p_2 AND $\cdots$ AND p_k, the degree of satisfaction of p is given by $d(p) = min_{1 \leq i \leq k}(d(p_i))$. If not all predicates are independent, the conjunction can be partitioned into conjuncts so that the predicates in the same conjunct are dependent, but those in different conjuncts are independent. Then, the method can be applied to the conjuncts. For simplicity, we assume that the predicates are independent.

The semantics of a query uses the membership degree to indicate an overall possibility for the underlying data that generate an answer to satisfy the query conditions. The overall possibility is obtained based on fuzzy logic connectivities. Consider Query 1. For each pair of male person s and female person r, the satisfaction degree of the query condition is obtained based on the fuzzy AND of following conditions: s is in M, r is in F, $r.AGE = s.AGE$, and $s.INCOME >$ "medium high." Therefore,

$$d'_{r,s} = min(\mu_F(r), \mu_M(s), d(r.AGE = s.AGE),$$
$$d(s.INCOME > \text{"medium high"})).$$

If $d'_{r,s} > 0$, $r.NAME$ and $s.NAME$ form a tuple in the answer with $d'_{r,s}$ as the membership degree. Several identical pairs of names may occur during the query evaluation, but with different membership degrees. Since each pair of names satisfies the query, only one pair needs to be in the answer. Based on fuzzy OR, the highest membership degree of the identical name pairs will be chosen for the answer. Thus, in the end, if x, y is a pair of names in the answer, its satisfaction degree is given by $d(x, y) = \max_{r.NAME \equiv x, s.NAME \equiv y}(d'_{r,s})$. It is straightforward to generalize the semantics of Query 1 to that of general Fuzzy SQL queries including nested queries.

Discussion. Several methods that measure the degree of satisfaction have been proposed. The method in [28], [30] measures both possibility and necessity, which, for a predicate "$X\ \theta\ F$", are defined by

$$Poss(X\ \theta\ F) = max_{x,y}(min(\mu_X(x), \mu_F(y), \mu_\theta(x, y)))$$
$$Nece(X\ \theta\ F) = 1 - Poss(X\ \neg\theta\ F),$$

where x and y are taken from the domain over which X and F are defined and $\mu_{\neg\theta} = 1 - \mu_\theta$. Intuitively, the possibility measures the "best possibility" for the comparison to be successful and the necessity measures the "impossibility" for the opposite comparison to be successful. With convex and normal possibility distributions, such as those with trapezoidal shapes, necessity is always no greater than possibility. Although the use of both possibility and necessity leads to an explicit expression of the uncertainty of the satisfaction degree, it also causes several problems. As pointed out in [27], the main problem of the double-measure system is that a simple query will lead to two answer relations, one containing tuples possibly satisfying the query condition and the other containing tuples necessarily satisfying the query condition. As a result, the underlying algebraic operations, that is, selection, projection, join, etc., cannot be composed. This means that all queries must be evaluated by computing the cross product of all relations involved, followed by a selection, and then a projection. Such an evaluation is obviously inefficient. Furthermore, nested query is not supported in this framework. Another problem is that the double negation nature of the necessity is not intuitive to many people.

A different method, proposed in [38], defines the membership degree of a tuple in a fuzzy relation as a possibility distribution rather than a crisp value. Although the resulting algebraic operations can be composed, this method applies only to discrete possibility distributions, and, even in that case, the membership degrees may contain a large number of elements, thus reducing the efficiency of the system.

The method used in this paper is commonly used by other researchers [29], [41], [5], [23]. By using only the possibility measure, it is guaranteed that algebraic operations can be composed and nested query becomes practical. The price to pay for this simplification is that the "impossibility" for a possible answer to a query to be also a possible answer to the negation of the query is not explicitly measured and, therefore, one must assume that every possible answer to a query is also a completely possible answer to the negation of the query. This price may be worthwhile to pay in order to gain the convenience of having nested query capability and the ability of efficient query processing for two reasons. First, in practice, users are usually more concerned with finding possible answers to a query than knowing how impossible it is that these answers are also possible answers to the negation of the query. Second, if it is necessary, one can always issue the negation of a query and get a more direct and easy to understand measure than that given by necessity.

2.3 Evaluation of Nested Fuzzy Queries

The nested query in Fuzzy SQL provides a convenient way for a user to express a complex query. For example, the following nested query finds the name of medium young female persons who has a middle age male person's income.

Query 2
```
        SELECT    F.NAME
        FROM      F
        WHERE     F.AGE = "medium young" AND
                  F.INCOME IN
                  ( SELECT    M.INCOME
                    FROM      M
                    WHERE     M.AGE = "middle age" )
```

Among the two nested query blocks, the outer block involves only the relation F and the inner block involves only the relation M. A naive execution of this query is a nested loop in which the inner relation M is scanned once for every tuple of the outer relation F. If the number of tuples in M is large, the processing cost, especially the I/O cost, can be very high.

Since the inner block does not involve data of the outer relation, the evaluation may be speeded up by using an intermediate relation containing all tuples of the inner relation that satisfy the predicate $M.AGE =$ *"middle age"* with a degree higher than 0. If this intermediate relation is significantly smaller than M, it can be scanned much faster than M. To speed up the evaluation further, one must avoid scanning the entire intermediate relation for every tuple of the outer relation. In conventional databases, this is achieved by unnesting Query 2 to obtain the following equivalent flat query:

Query 3
```
        SELECT    F.NAME
        FROM      F, M
        WHERE     F.AGE = "medium young" AND
                  M.AGE = "middle age" AND
                  F.INCOME = M.INCOME
```

In this query, the predicate $F.INCOME = M.INCOME$ is a join condition. Since unnested queries frequently involve joins, their evaluations in a conventional relational database can be very efficient by following an optimal join strategy formulated by a good query optimizer. In a fuzzy relational database, the equivalence between two fuzzy queries is more complex than in a conventional database since it requires that not only the answers contain the same set of tuples but also the corresponding tuples have the same membership degree.

3 Processing of Fuzzy Equi-Join

As mentioned in Section 1, the most efficient join method in conventional databases, hash-join, is not applicable in a fuzzy database. In this section, we extend the merge-join in the context of fuzzy databases. For convenience, we assume that relations R and S are joined based on the join condition $R.X = S.X$ and R is the outer relation in the merge-join algorithm. We first define a partial order on the set of R-tuples (or S-tuples). Since data values have trapezoidal membership functions, each data value v represents an interval $[b(v), e(v)]$, in which μ_v is greater than 0. For a crisp value v, we let $b(v) = e(v) = v$. For example, the crisp value 28 represents $[28, 28]$ and the fuzzy value "medium young" represents $[20, 35]$. Now, the data values can be ordered according to their intervals.

Definiton 3.1.

1. *For two values v_1 and v_2, $v_1 \prec v_2$ if $b(v_1) < b(v_2)$ or $b(v_1) = b(v_2)$ and $e(v_1) < e(v_2)$; $v_1 \preceq v_2$ if $v_1 \prec v_2$ or $v_1 \equiv v_2$.*
2. *For two tuples r_1 and r_2, $r_1 \prec r_2$ wrt an attribute X if $r_1.X \prec r_2.X$; $r_1 \preceq r_2$ wrt X if $r_1.X \preceq r_2.X$.*

Example 3.1. Let r_1, r_2, and r_3 be tuples of R, and s_1, s_2, and s_3 be tuples of S. Assume that $r_1.X, r_2.X$, and $r_3.X$ represent, respectively, $[30, 35]$, $[20, 28]$, and $[20, 35]$; and $s_1.X$, $s_2.X$, and $s_3.X$ represent, respectively, $[32, 34]$, $[20, 25]$ and $[30, 40]$. By Definition 3.1, $[20, 28] \prec [20, 35] \prec [30, 35]$, thus, $r_2.X \prec r_3.X \prec r_1.X$. Similarly, $s_2.X \prec s_3.X \prec s_1.X$.

Notice that, for any two values a and b, $d(a = b) = 0$ if their intervals do not intersect. As a result, only those tuples r and s whose $r.X$ and $s.X$ have intersecting intervals need to be considered for the equal-join. For the merge-join, both R and S are first sorted on X based on $\prec$. Then, for each tuple r in R, the S-tuples are scanned in the sorted order. The scan should terminate as soon as all S-tuples that have intervals intersecting that of $r.X$ have been examined. For Example 3.1, r_2 joins with s_2. But, since the interval $[30, 40]$ of $s_3.X$ falls completely to the right of the interval $[20, 28]$ of $r_2.X$, r_2 will not join with any S-tuple succeeding s_2, therefore, the scan of S for r_1 should stop at s_3. The sequence of S-tuples that have to be examined for an R-tuple is defined as follows:

Definition 3.2.

1. *For a tuple r in R, $sml(r)$ is the smallest value v (according to the order $\prec$) that appears in $S.X$ and intersects $r.X$ and $lrg(r)$ is the largest value v that appears in $S.X$ and intersects $r.X$.*
2. *The range of a tuple r of R is a subset of S defined by $Rng(r) = \{s : s \in S \ \ and \ \ sml(r) \preceq s.X \preceq lrg(r)\}$ and $Rng(r) = \emptyset$ if $r.X$ does not intersect v for any value v in $S.X$.*

After both R and S are sorted, the join phase is carried out as follows: The tuples of R are loaded into the main memory one page at a time in the sorted order. For the ith R-tuple, r_i, the S-tuples in $Rng(r_i)$ are identified during a scan of S. The scan starts at the place where the first S-tuple in the range of the previous R tuple $(Rng(r_{i-1}))$ was found, and proceeds towards the end of S. The first and the last S-tuple of $Rng(r_i)$ can be identified because every S-tuple s appearing before $Rng(r_i)$ satisfies $e(s.X) < b(r_i.X)$ and every S-tuple s appearing after $Rng(r_i)$ satisfies $b(s.X) > e(r_i.X)$. The pages of S are loaded into the main memory one by one in the sorted order. If a page of S contains only those tuples that appear before $Rng(r_i)$, it will not be scanned again in the rest of the process because any S-tuple which precedes $Rng(r_i)$ will also precede every $Rng(r_k)$ for $k > i$. If a page of S contains some tuples in $Rng(r_i)$, then the join is performed and the page stays in the main memory since some tuples in the page may join with the next R-tuple r_{i+1} (either in the current or the next page of R). If a page contains one tuple that follows the tuples in $Rng(r_i)$, the join between r_i and S is completed when this tuple is encountered. The join between the next R-tuple, r_{i+1}, and S will then start by scanning the pages of S already in the main memory. The process is repeated for all tuples of R.

We now analyze the complexity of the merge join method. Since we are more interested in joins on fuzzy data, we assume that the join attributes are nonkey. In the following, let n_T and b_T be the number of tuples and the number of pages, respectively, of a relation T. Let M be the number of pages of the main memory buffer.

For the join phase, if the buffer is large enough to hold one page of R and all pages of the largest $Rng(r)$, only one scan of both R and S is needed, therefore, the I/O cost is $O(b_R + b_S)$. During the join phase, each R-tuple is scanned exactly once. For each tuple $r \in R$, all S-tuples in $Rng(r)$ need to be examined. If an S-tuple appears in $Rng(r)$ for more than one R-tuple, it will be scanned once for each of those R-tuples. We assume that the number of tuples in $Rng(r)$ is proportional to the number of S-tuples joining with r and that each R-tuple joins with a constant number of S-tuples. Under these assumptions, the CPU time for the join phase is of order $O(n_R + n_S)$.

Sorting a relation based on the order $\prec$ is similar to sorting a relation based on the standard $<$ linear order, but two comparisons may be needed to compare two tuples: The left end points are compared first if they are the same, then the right end points are compared. When the ordering between the two tuples is decided, the sorting algorithm proceeds as it does for sorting ordinary relations. Thus, the CPU time for sorting a relation T is still $O(n_T \log n_T)$. If b_T is much larger than M, the I/O cost for sorting is $b_T \log_M b_T$. In practice, however, M is usually smaller than but still comparable with b_T, thus a relation can be sorted in a linear I/O time (two passes) [37], [9].

Put it together and notice that the number of pages of a relation can be assumed to be proportional to the number of tuples of the relation, the response time of the extended merge-join method is of order $O(n_R \log n_R + n_S \log n_S)$.

For the naive nested loop method, each tuple of R needs to be compared with each tuple of S and the CPU time will be of order $O(n_R \times n_S)$. When $b_R \leq b_S$, we can allocate one page in the main memory to relation S and the remaining pages to relation R so that the I/O time for the join will be of order $O(b_R + \frac{b_R}{(M-1)} \times b_S)$. Thus, the response time of the nested loop method is of order $O(n_R \times n_S)$. We will use these notations when discussing the response time in later sections.

Notice that, unlike the merge-join in a conventional database, $Rng(r_i)$ may not be compact in the sense that some tuples in $Rng(r_i)$ may not actually join with r_i. For example, if $r.X$ has an interval $[30, 40]$ and $s.X$ has an interval $[10, 35]$, then any S-tuple s' with an interval of $s'.X$ in between of 10 and 30 will be in $Rng(r)$ yet not join with r. If such dangling tuples exist in $Rng(r)$, the performance of the extended merge-join will not be as efficient as the merge-join in a conventional database. However, in many applications, data values may be fuzzy but not excessively so, thus they have small intervals. In this case, the number of dangling tuples in $Rng(r)$ will be very small. Another limitation of the extended merge-join is that the possibility distributions must be continuous. An efficient join algorithm for both continuous and discrete possibility distributions is yet to be discovered.

Fuzzy joins are similar to the band join in conventional databases [9] and the valid-time natural join in temporal databases [36]. In a band join, each value of the joining attribute is a crisp value and represents an interval. However, all intervals for different values are of the same length. In a valid-time natural joins, a time interval can be of arbitrary length. Fuzzy joins are more general than the two kinds of joins since the interval associated with a fuzzy value varies from one fuzzy value to another and a fuzzy join predicate yields a value between 0 and 1. In both [9] and [36], partitioned joins based on sampling are suggested. More research is needed to decide the optimal join method (and the way to conduct sampling in fuzzy databases). An unnested query can be evaluated by any of the methods, including the naive nested loop method, but a nested query can be evaluated only by the nested loop method.

4 Simple Nested Queries

In this section, we consider two simple types of nested queries, namely the type N and type J, named after their counterpart in standard SQL [18]. These queries contain neither set exclusion operators nor aggregate functions. The difference between them is that the inner block of a type J query has a join predicate referencing the outer relation and that of a type N query does not. In the rest of the paper, we use p_1 to denote the conjunction of predicates involving only the outer relation, and p_2 those involving only the inner relation.

The following Query N is a type N query.

Query N

```
SELECT   R.X
FROM     R
WHERE    p1 AND R.Y is in
            ( SELECT  S.Z
              FROM    S
              WHERE   p2 )
```

The execution semantics of Query N is as follows: Each S-tuple s satisfies p_2 with a degree $d_s = min(\mu_S(s), d(p_2(s)))$. If $d_s > 0$, $s.Z$ belongs to a temporary relation T with the membership degree d_s. If several tuples in T have the same value, only the one with the highest membership degree will stay and others are removed. Thus, for each tuple z in T, the membership degree is

$$\mu_T(z) = max_{s.Z\equiv z}(d_s) = max_{s.Z\equiv z}(min(\mu_S(s), d(p_2(s)))).$$

Then, each R-tuple r satisfies the selection condition with a degree

$$d_r = min(\mu_R(r), d(p_1(r)), d(r.Y \text{ is in } T)),$$

where $d(r.Y\ is\ in\ T)$ is, based on [25], given by

$$d(r.Y\ is\ in\ T) = \begin{cases} max_{z\in T}(min(\mu_T(z), d(r.Y = z))), & T \neq \emptyset; \\ 0, & \text{otherwise.} \end{cases}$$

Intuitively, $d(r.Y\ is\ in\ T)$ is the possibility for $r.Y$ to be equal to any value in set T. If $d_r > 0$, $r.X$ belongs to the answer with the membership degree d_r. The duplicate tuples in the answer are also removed by keeping the one with the highest membership degree. Thus, each tuple x of the answer to Query N has a membership degree

$$\mu_N(x) = \max_{r.X\equiv x}(min(\mu_R(r), d(p_1(r)), \max_{z\in T}(min(\mu_T(z), d(r.Y = z)))).$$

Example 4.1 A type N query is Query 2 given in Section 2. Suppose the relations are as follows and the membership functions of AGE and $INCOME$ are as given in Fig. 2:

F				
ID	NAME	AGE	INCOME	D
101	Ann	*about* 35	*about* 60*K*	1
102	Ann	*medium young*	*medium high*	1
103	Betty	*middle age*	*high*	1
104	Cathy	*about* 50	*low*	1

M				
ID	NAME	AGE	INCOME	D
201	Allen	24	*about* 25*K*	1
202	Allen	*about* 50	*about* 40*K*	1
203	Bill	*middle age*	*high*	1
204	Carl	*about* 29	*medium low*	1

The temporary relation T, the set T_2 of all tuples with $d_F > 0$, and the final answer relation are given below.

T	
INCOME	D
about 40*K*	0.4
high	1

T_2	
NAME	D
Ann	0.3
Ann	0.7
Betty	0.7

Answer	
NAME	D
Ann	0.7
Betty	0.7

The following unnested Query N' for Query N is identical to that given in [18].

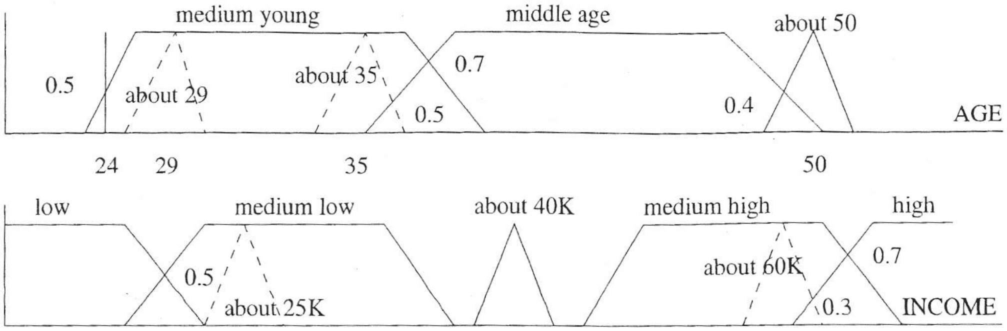


Fig. 2. Data used in Example 4.1.

Query N′

```
SELECT  R.X
FROM    R, S
WHERE   p1 AND R.Y = S.Z AND p2
```

The execution semantics of Query N′ is as follows: Each pair of tuples r in R and s in S satisfy the selection condition with a degree

$$d'_{r,s} = min(\mu_R(r), \mu_S(s), d(p_1(r)), d(p_2(s)), d(r.Y = s.Z)).$$

If $d'_{r,s} > 0$, $r.X$ is in the answer. After eliminating duplicates, each value x in the answer will have the degree $\mu_{N'}(x) = \max_{r.X \equiv x} \max_{s \in S}(d'_{r,s})$.

Theorem 4.1. *Query N′ is equivalent to Query N.*

Proof. We show that, for each x in the domain of $R.X$, $\mu_N(x) = \mu_{N'}(x)$, thus the answers to Query N and Query N′ are identical fuzzy relations.

According to the execution semantics,

$$\mu_{N'}(x) = \max_{r.X \equiv x} \max_{s \in S}(min(\mu_R(r), \mu_S(s), d(p_1(r)), d(p_2(s)), d(r.Y = s.Z)))$$

and

$$\begin{aligned}
\mu_N(x) &= \max_{r.X \equiv x}(min(\mu_R(r), d(p_1(r)), \max_{z \in T}(min(\mu_T(z), d(r.Y = z)))))\\
&= \max_{r.X \equiv x}(\max_{z \in T}(min(\mu_R(r), d(p_1(r)), min(\mu_T(z), d(r.Y = z)))))\\
&= \max_{r.X \equiv x}(\max_{z \in T}(min(\mu_R(r), d(p_1(r)), \mu_T(z), d(r.Y = z))))\\
&= \max_{r.X \equiv x}(\max_{z \in T}(min(\mu_R(r), d(p_1(r)), \max_{s.Z \equiv z}(d_s), d(r.Y = z))))\\
&= \max_{r.X \equiv x}(\max_{z \in T}(\max_{s.Z \equiv z}(min(\mu_R(r), d(p_1(r)), d_s, d(r.Y = z)))))\\
&= \max_{r.X \equiv x}(\max_{z \in T}(\max_{s.Z \equiv z}(min(\mu_R(r), d(p_1(r)), \mu_S(s), d(p_2(s)), d(r.Y = z)))))\\
&= \max_{r.X \equiv x} \max_{s \in S}(min(\mu_R(r), d(p_1(r)), \mu_S(s), d(p_2(s)), d(r.Y = s.Z))).
\end{aligned}$$

In the last step, $\max_{r.X \equiv x}\max_{z \in T}\max_{s.Z \equiv z}(min(*))$ is replaced by $\max_{r.X \equiv x}\max_{s \in S}(min(*))$. A pair of tuples r and s will not be covered by the former, but will be by the latter if $r.X \equiv x$ and $s.Z$ is not in T. However, by the execution semantics, if $s.Z$ is not in T, we have $d(p_2(s)) = 0$, which in turn implies $min(*) = 0$ since $d(p_2(s))$ is inside the expression $(*)$. Therefore, each pair of r and s that is covered only by the latter expression yields a value of zero. We can replace the former expression by the latter one because $(min(*))$ always gives nonnegative values and the maximum value is chosen from all values produced by $(min(*))$. For instance, if $T = \emptyset$, that is, $d(p_2(s)) = 0$ for all tuples s, then $\mu_N(x) = \mu_{N'}(x) = 0$ for any x, although the former covers no pairs of tuples. Thus, $\mu_N(x) = \mu_{N'}(x)$ for any value x. □

Query N′ can be evaluated using the merge-join as discussed in Section 3. After sorting R on $R.Y$ and S on $S.Z$, a tuple r is joined with all S-tuples in $Rng(r)$. The value $r.X$ is inserted into the answer with degree $max_{s \in Rgn(r)}(d'_{r,s})$, which is the same as $max_{s \in S}(d'_{r,s})$ since $d(r.Y = s.Z)$ and, hence, $d'_{r,s}$ is 0 for any s not in $Rng(r)$. The degree $\mu_{N'}(x)$ is obtained by keeping the tuple with the highest membership degree when eliminating duplicate tuples. To reduce the cost, only those tuples in R (respectively, S) that satisfy p_1 (respectively, p_2) positively should be sorted. Let n_R and n_S be the reduced sizes of R and S, respectively. The response time of the modified merge-join for Query N′ is of the order $O(n_R \log n_R + n_S \log n_S)$, while the response time of the nested loop method on Query N is of the order $O(n_R \times n_S)$.

Now, consider the following type J nested query.

Query J

```
SELECT  R.X
FROM    R
WHERE   p1 AND R.Y is in
        ( SELECT  S.Z
          FROM    S
          WHERE   p2 AND S.V = R.U )
```

The execution semantics of Query J is as follows: For each tuple r of R, the inner block produces a temporary relation, $T(r)$. For each tuple s of S, $s.Z$ belongs to $T(r)$ with the membership degree

$$d_r(s) = min(\mu_S(s), d(p_2(s)), d(s.V = r.U)).$$

After eliminating duplicates, each tuple z in $T(r)$ has the degree $\mu_{T(r)}(z) = \max_{s.Z \equiv z}(d_r(s))$. Then, r satisfies the query with the degree

$$d_r = min(\mu_R(r), d(p_1(r)), d(r.Y \text{ is in } T(r))),$$

and $r.X$ belongs to the answer if $d_r > 0$. After eliminating duplicates, each tuple x in the answer has the degree $\mu_J(x) = \max_{r.X \equiv x}(d_r)$.

The following unnested Query J′ for Query J is also identical to that in [18]:

```
Query J′
        SELECT   R.X
        FROM     R, S
        WHERE    p1 AND p2 AND R.Y = S.Z
                    AND R.U = S.V
```

By the execution semantics, each tuple x in the answer has the degree

$$\mu_{J'}(x) = \max_{r.X \equiv x} \max_{s \in S}(min(\mu_R(r), \mu_S(s), d(p_1(r)), d(p_2(s)), \\ d(r.Y = s.Z), d(r.U = s.V))).$$

Theorem 4.2. *Query J′ is equivalent to Query J.*

Proof. We show that $\mu_J(x) = \mu_{J'}(x)$ for every value x of $R.X$. Notice that $\mu_J(x) = \max_{r.X \equiv x}(d_r)$ and

$$\mu_{J'}(x) = \max_{r.X \equiv x} \max_{s \in S}(d'_{r,s}).$$

If, for each r in R, we let $d'_r = \max_{s \in S}(d'_{r,s})$, we will have $\mu_{J'}(x) = \max_{r.X \equiv x}(d'_r)$ and it suffices to prove $d'_r = d_r$ for each r in R. There are two cases, depending on whether $T(r)$ is empty or not.

Case 1: $T(r)$ is empty. We immediately have $d_r = 0$ since $d(r.Y \text{ } is \text{ } in \text{ } T(r)) = 0$. Now, consider d'_r. Since $T(r)$ is empty, we must have, for every s in S,

$$d_r(s) = min(\mu_S(s), d(p_2(s)), d(s.V = r.U)) = 0,$$

that is, either $d(p_2(s)) = 0$, or $d(s.V = r.U) = 0$ and, therefore,

$$d'_r = \max_{s \in S}(min(\mu_R(r), \mu_S(s), d(p_1(r)), d(p_2(s)), \\ d(r.Y = s.Z), d(r.U = s.V))) = 0.$$

Case 2: $T(r)$ is not empty. For Query J,

$$\begin{aligned} d_r &= min(\mu_R(r), d(p_1(r)), d(r.Y \text{ is in } T(r))) \\ &= min(\mu_R(r), d(p_1(r)), \max_{z \in T(r)}(min(\mu_{T(r)}(z), d(r.Y = z)))) \\ &= \max_{z \in T(r)}(min(\mu_R(r), d(p_1(r)), \mu_{T(r)}(z), d(r.Y = z))) \\ &= \max_{z \in T(r)}(min(\mu_R(r), d(p_1(r)), \max_{s.Z \equiv z}(d_r(s)), d(r.Y = z))) \\ &= \max_{z \in T(r)} \max_{s.Z \equiv z}(min(\mu_R(r), d(p_1(r)), \\ &\qquad min(\mu_S(s), d(p_2(s)), d(s.V = r.U)), d(r.Y = z))) \\ &= x \max_{z \in T(r)} \max_{s.Z \equiv z}(min(\mu_R(r), d(p_1(r)), \mu_S(s), d(p_2(s)), \\ &\qquad d(s.V = r.U), d(r.Y = z))). \end{aligned}$$

For Query J′,

$$\begin{aligned} d'_{r,s} &= min(\mu_R(r), \mu_S(s), d(p_1(r)), d(p_2(s)), \\ &\qquad d(r.Y = s.Z), d(r.U = s.V)) \\ d'_r &= \max_{s \in S}(min(\mu_R(r), \mu_S(s), d(p_1(r)), d(p_2(s)), \\ &\qquad d(r.Y = s.Z), d(r.U = s.V))). \end{aligned}$$

Let $*$ denote

$$min(\mu_R(r), \mu_S(s), d(p_1(r)), d(p_2(s)), \\ d(r.Y = s.Z), d(r.U = s.V)).$$

Then, $d'_r = \max_{s \in S}(*)$ and $d_r = \max_{z \in T(r)} \max_{s.Z \equiv z}(*)$.

The expression for d'_r covers all tuples of S, but that for d_r may not. A tuple s is not covered by the expression for d_r if $s.Z$ is not in $T(r)$, which implies $d_r(s) = 0$ since $s.Z$ is retrieved into $T(r)$ only if $d_r(s) > 0$. Because $d_r(s) = min(\mu_S(s), d(p_2(s)), d(s.V = r.U)) = 0$, we have either $d(p_2(s)) = 0$ or $d(s.V = r.U) = 0$. In either case, the value of $*$ is zero. Since all values appearing in $*$ are nonnegative and the maximum value is computed from these $*$ values, we have $d'_r = d_r$. □

Query J′ can be evaluated using the extended merge-join in the same way as Query N′ can. The only differences are that the sorting can be based on either (Y, Z) or (U, V) and that two join predicates must be evaluated when computing the value of $d'_{r,s}$. Consequently, the response time of the extended merge-join for Query J′ is of order $O(n_R \log n_R + n_S \log n_S)$, while that of the nested loop method is of order $O(n_R \times n_S)$.

5 The Set Exclusion Operator

Kim [18] used an *antijoin* predicate to unnest a type N or type J query when the set inclusion operator *is in* is replaced by the set exclusion operator *is not in*. Kim pointed out that unnesting such a nested query requires careful consideration. We only discuss the following Query JX, which is of type J with the set exclusion operator. (The discussion for a type N query with the set exclusion operator is similar and simpler.)

```
Query JX
        SELECT   R.X
        FROM     R
        WHERE    R.Y is not in
                 ( SELECT   S.Z
                   FROM     S
                   WHERE    S.V = R.U )
```

For the sake of simplicity of presentation, we have left out predicates p_1 and p_2. But, the result holds if either or both of them exists.

The execution semantics of Query JX is as follows: For each tuple r of R, a temporary relation $T(r)$ is generated from the inner block. Each value z in $S.Z$ is in $T(r)$ with the degree

$$\begin{aligned} \mu_{T(r)}(z) &= \max{}_{s.Z \equiv_z}(d_r(s)) \\ &= \max{}_{s.Z \equiv_z}(min(\mu_S(s),\ d(r{:}U = s.V))). \end{aligned}$$

A value x in $R.X$ will be in the answer with the degree

$$\begin{aligned}\mu_{JX}(x) &= \max_{r.X \equiv x}(d_r)\\ &= \max_{r.X \equiv x}(min(\mu_R(r), d(r.Y\ is\ not\ in\ T(r)))),\end{aligned}$$

where $d(r.Y\ is\ not\ in\ T(r)) = 1 - d(r.Y\ is\ in\ T(r))$. As an example, the following query is type JX and finds the name of employees of the Sales department who do not have an income of any employee of the Research department with his/her age.

Query 4

```
    SELECT    R.NAME
    FROM      EMP_SALES R
    WHERE     R.INCOME is not in
              ( SELECT  S.INCOME
                FROM    EMP_RESEARCH S
                WHERE   S.AGE = R.AGE )
```

To unnest Query JX, we have to define a temporary relation that uses both the *WITH* and the *GROUPBY* clauses, and explicitly refers to the membership degrees of R, S, and the answer relations. Let $R.K$ be a key of R. Query JX can be unnested to the following Query JX′:

```
JXT(K, X) = ( SELECT    R.K, R.X, MIN(D)
              FROM      R, S
              WHERE     R.D AND ¬(S.D AND
                        R.Y =S.Z AND R.U = S.V)
              WITH      D ≥ 0
              GROUPBY   R.K)
```

Query JX′

```
              SELECT    X
              FROM      JXT
```

Notice that the query to obtain JXT is flat and Query JX′ is only used to get rid of attribute K. The direct use of membership degree attributes $R.D$ and $S.D$ are unconventional. Since each predicate is evaluated to a satisfaction degree, and the membership degree can also be a satisfaction degree (of a complex query condition), a membership degree attribute can be used by itself as a predicate. The satisfaction degree of $R.D$ is then defined to be the membership degree of the R-tuple being referred to.

The execution semantics of Query JX′ is as follows: For each R-tuple r and every S-tuple s, the degree for (r, s) to satisfy the WHERE clause of the JXT query is given by:

$$\begin{aligned}d'_{r,s} =&min(\mu_R(r), 1 - min(\mu_S(s),\\ &d(r.U = s.V), d(r.Y = s.Z))),\end{aligned}$$

where $1 - d(p)$ is the satisfaction degree of $\neg p$. Because of the *WITH* clause, all (r, s) pairs are kept around even if $d'_{r,s} = 0$. The (r, s) pairs for the same r form a group according to the GROUPBY clause and the minimum degree of each group is given by $d'_r = min_{s \in S}(d'_{r,s})$, according to the $MIN(D)$ in the *SELECT* clause. If $d'_r > 0$, $r.X$ is included in relation JXT with that degree. Query JX′ retrieves distinct values of attribute $JXT.X$. After eliminating duplicates, a value x in the answer has the degree $\mu_{JX'}(x) = \max_{r.X \equiv x}(d'_r)$.

Theorem 5.1. *Query JX′ is equivalent to Query JX.*

Proof. We prove that $\mu_{JX}(x) = \mu_{JX'}(x)$ for each value x in the domain of $R.X$. It suffices to prove $d_r = d'_r$ for every r of R, since

$$\mu_{JX}(x) = \max_{r.X \equiv x}(d_r)$$

and

$$\mu_{JX'}(x) = \max_{r.X \equiv x}(d'_r).$$

Case 1: $T(r)$ **is empty.** For Query JX,

$$d(r.Y\ is\ not\ in\ T(r)) = 1$$

and

$$d_r = min(\mu_R(r), d(r.Y\ is\ not\ in\ T(r))) = \mu_R(r).$$

For Query JX′, that $T(r)$ is empty implies $d(s.V = r.U) = 0$ for all tuples s of S. Thus,

$$\begin{aligned}d'_{r,s} =&min(\mu_R(r), 1 - min(\mu_S(s),\\ &d(r.Y = s.Z), d(r.U = s.V))) = \mu_R(r),\end{aligned}$$

Case 2: $T(r)$ **is not empty.** For Query JX,

$$\begin{aligned}d_r &= min(\mu_R(r), 1 - d(r.Y\ is\ in\ T(r)))\\ &= min(\mu_R(r), 1 - \max_{z \in T(r)}(min(\mu_{T(r)}(z), d(r.Y = z)))\\ &= min(\mu_R(r), \min_{z \in T(r)}(1 - min(\mu_{T(r)}(z), d(r.Y = z))))\\ &= \min_{z \in T(r)}(min(\mu_R(r), 1 - min(\max_{s.Z \equiv z}(d_r(s)), d(r.Y = z))))\\ &= \min_{z \in T(r)}(min(\mu_R(r), \min_{s.Z \equiv z}(1 - min(d_r(s), d(r.Y = z)))))\\ &= \min_{z \in T(r)} \min_{s.Z \equiv z}(min(\mu_R(r), 1 - min(min(\mu_S(s),\\ &\quad d(s.V = r.U)), d(r.Y = z)))))\\ &= \min_{z \in T(r)} \min_{s.Z \equiv z}(min(\mu_R(r), 1 - min(\mu_S(s),\\ &\quad d(s.V = r.U)), d(r.Y = z)))).\end{aligned}$$

For Query JX′,

$$\begin{aligned}d'_r =&\min_{s \in S}(d'_{r,s}) = \min_{s \in S}(min(\mu_R(r), 1 - min(\mu_S(s),\\ &d(s.V = r.U), d(r.Y = z)))).\end{aligned}$$

Let

$$* = min(\mu_R(r), 1 - min(\mu_S(s), d(s.V = r.U), d(r.Y = z))),$$

then $d'_r = \min_{s \in S}(*)$ and $d_r = \min_{z \in T(r)}\min_{s.Z \equiv z}(*)$. The expression for d'_r covers all tuples of S and the expression for d_r may not. A tuple s of S will not be covered by the expression for d_r only if $d(s.V = r.U) = 0$, thus it will cause $*$ to yield $\mu_R(r)$, which is the largest possible value $*$ can ever yield. Therefore, $d'_r = d_r$ for any tuple r of R. □

Although Query JX′ and Query J′ are quite different, Query JX′ can be evaluated in a way similar to that Query J′ can. We can use either $R.U = S.V$ or $R.Y = S.Z$ in the merge-join. For any pair of r and s, $d'_{r,s} \leq \mu_R(r)$. If s is not in $Rng(r)$, then $d(r.U = s.V) = 0$ or $d(r.Y = s.Z) = 0$ and $d'_{r,s} = \mu_R(r)$. That is, $d'_r = min_{s \in S}(d'_{r,s}) = min_{s \in Rgn(r)}(d'_{r,s})$. So we join a tuple r with all S-tuples in $Rng(r)$ while they are in the main memory, compute d'_r, and retrieve $r.X$ when $d'_r > 0$. Thus, the response time of the extended merge-join method for Query JX' is again of order $O(n_R \log n_R + n_S \log n_S)$, and that for the nested loop method is of order $O(n_R \times n_S)$.

6 Nested Queries with Aggregate

In this section, we consider the unnesting of a type JA query [18] in which the inner block has a join predicate referencing the outer relation and the *SELECT* clause contains an aggregate function which produces a nonnull value from a nonempty fuzzy set of (maybe fuzzy) values.

Different semantics of aggregate functions in fuzzy databases have been proposed in the past [31], [32], [11], [25] and a standard semantics is yet to come. To give the reader a concrete semantics, we briefly present the aggregate functions in Fuzzy SQL as described in [23]. Fuzzy SQL has aggregate functions COUNT, AVG, SUM, MIN, and MAX. COUNT returns the number of values in a fuzzy set. AVG and SUM are defined based on fuzzy arithmetic operations. With a trapezoidal (or triangular) membership function, a fuzzy value induces two intervals (conventionally termed *α-cut*). One interval contains all values in the domain whose membership degree is 1 (i.e., 1-cut), and the other interval contains all values whose membership degree is greater than 0 (i.e., 0-cut). Fuzzy arithmetic operations take two values and determine the two intervals of the resulting value. For example, let x and y be two values with 0-cuts $[x_1, x_4]$ and $[y_1, y_4]$, and 1-cuts $[x_2, x_3]$ and $[y_2, y_3]$, respectively. $x + y$ produces a value z with 0-cut $[x_1 + y_1, x_4 + y_4]$ and 1-cut $[x_2 + y_2, x_3 + y_3]$. Other arithmetic operations are defined similarly. AVG is defined by fuzzy addition and division, and SUM is defined by fuzzy addition. MIN and MAX are defined by using a defuzzification method which allows fuzzy values to be sorted based on the center of their 1-cuts. For empty fuzzy set of values, AVG, SUM, MIN, and MAX produces NULL value.

Despite the special semantics of aggregate functions in Fuzzy SQL, the technique given in this section is general and can be applied to other systems as long as the aggregate functions produce deterministically a nonnull value from a given nonempty fuzzy set and a NULL value for empty fuzzy sets. The key idea is to show that the same function is applied in both the nested and the unnested queries to the same set of values. The following Query JA is a type JA nested query.:

Query JA

SELECT R.X
FROM R
WHERE p_1 AND $R.Yop_1$
(SELECT $AGG(S.Z)$
FROM S
WHERE p_2 AND
$S.V\ op_2\ R.U$)

Both op_1 and op_2 are comparison operators in $\{<, >, \leq, \geq, =\}$ and AGG is one of the aggregate functions MAX, MIN, AVG, SUM, and $COUNT$. Notice that if no join predicate exists in the inner block, the inner block produces the same single value for every tuple of R and no unnesting is needed.

The execution semantics of Query JA is as follows: For each tuple r of R, a temporary relation $T(r)$ is generated from the inner block where

$$\begin{aligned} T(r) =& \{z : \exists s \in S \text{ such that } s.Z \equiv z, \\ & d(p_2(s)) > 0, \text{and } d(s.V\ op_2\ r.U) > 0\}. \end{aligned}$$

A value z in $S.Z$ is in $T(r)$ with the degree

$$\mu_{T(r)}(z) = \max_{s.Z \equiv z}(min(\mu_S(s), d(p_2(s)), d(s.V\ op_2\ r.U))).$$

Then, the aggregate function AGG is applied on $T(r).Z$ to obtain a value $A(r)$ with a degree $D(A(r))$. We assume that $D(A(r))$ is a function of $T(r)$. For Fuzzy SQL, $D(A(r)) = 1$. But it can also be defined as the average membership degree, or weighted average membership degree of $T(r)$. The tuple r satisfies the query condition with the degree

$$d_r = min(\mu_R(r), d(p_1(r)), D(A(r)), d(r.Y\ op_1\ A(r))).$$

If $d_r > 0$, $r.X$ is in the answer. After eliminating duplicates, each value x of $R.X$ in the answer has the degree $\mu_{JA}(x) = \max_{r.X \equiv x}(d_r)$. Notice that, if $T(r) = \emptyset$, and the function is $COUNT$, then $A(r) = 0$ and

$$d_r = min(\mu_R(r), d(p_1(r)), D(A(r)), d(r.Y\ op_1\ 0));$$

however, if the function is not $COUNT$, $A(r) \equiv null$ and $d_r = 0$.

As an example, the following type JA query finds the names of cities in region A, each of which has an average household-income greater than the maximum average household-income of cities in region B with similar population.

Query 5

SELECT *R.NAME*
FROM CITIES_REGION_ AR
WHERE *R.AVE_HOME_INCOME>*
(SELECT *MAX(S.AVE_HOME_*
INCOME)
FROM CITIES_REGION_B *S*
WHERE *S.POPULATION*
= *R.POPULATION*)

We unnest Query JA using two temporary relations, T_1 and T_2, defined as follows:

$T_1(U)$ = (SELECT $R.U$
FROM R
WHERE p_1)
$T_2(U, A)$ = (SELECT $T_1.U$, $AGG(S.Z)$
FROM T_1, S
WHERE p_2
$S.V\ op_2\ T_1.U$
GROUPBY $T_1.U$)

Intuitively, T_1 is the set of all values of $R.U$ that can ever be used to evaluate the inner block of Query JA and T_2 is the set of all aggregated values that can ever be obtained in the inner block, with each value accompanied by the corresponding value of $R.U$ that produces it. T_1 is obtained from the R-tuples that satisfy p_1 by projecting on $R.U$ with duplicates removed and all membership degrees set to 1. T_2 is obtained by joining T_1 with S-tuples that satisfy p_2 on the join condition $S.V op_2\ T_1.U$, grouping the result based on $T_1.U$, and then, applying AGG to each group. For each value u in $T_1.U$, the group is

$$\begin{aligned} T'(u) = &\{z : \exists s \in S \text{ such that } s.Z \equiv z, d(p_2(s)) > 0 \\ &\text{and } d(s.V\ op_2\ u) > 0\}; \end{aligned}$$

and a value z of $S.Z$ is in $T'(u)$ with the degree

$$\mu_{T'(u)}(z) = \max_{s.Z \equiv z}(min(\mu_S(s), d(p_2(s)), d(s.V\ op_2\ u))).$$

If $T'(u) \neq \emptyset$, a result, $A'(u)$, with degree $D(A'(u))$ is computed from $T'(u)$ by AGG, and a tuple $(u, A'(u))$ is inserted into T_2 with degree $D(A'(u))$. If $T'(u) = \emptyset$, T_2 contains no tuple for u. Notice that, by their definitions, for every R-tuple r, $T(r)$ and $T'(r.U)$ are identical and, for every z in $S.Z$, $\mu_{T(r)}(z) = \mu_{T'(r.U)}(z)$.

Query JA can then be unnested to either one of the following two flat queries, namely, Query JA′, if AGG is not $COUNT$, or Query COUNT′, if otherwise.

```
Query JA′
        SELECT    R.X
        FROM      R, T_2
        WHERE     p_1 AND R.U ≡ T_2.U
                      AND R.Y op_1 T_2.A
```

For each R-tuple r and each tuple $t = (u, A'(u))$ in T_2, r and t will satisfy the query condition with the degree

$$\begin{aligned} d'_{r,t} = min(&\mu_R(r), d(p_1(r)), d(r.U \equiv u), \\ &D(A'(u)), d(r.Y\ op_1\ A'(u))). \end{aligned}$$

Notice that $d(r.U \equiv u)$ is binary, and there can be at most one tuple in T_2 for which $d(r.U \equiv u) = 1$, thus, r will satisfy the query condition with the degree $d'_r = \max_{t \in T_2}(d'_{r,t})$. If $d'_r > 0$, $r.X$ belongs to the answer with degree d'_r. After eliminating duplicates, a value x is in the answer with degree $\mu_{JA'}(x) = \max_{r.X \equiv x}(d'_r)$.

```
Query COUNT′
        SELECT    R.X
        FROM      R, T_2
        WHERE     p_1 AND R.U+ ≡ T_2.U
                  [R.Y op_1 T_2.A : RY op_1 0]
```

The *WHERE* clause is a conjunction of the predicate p_1 and a left outer join predicate (denoted by $+\equiv$) followed by an *IF-THEN-ELSE* structure enclosed in a pair of square brackets. The left outer join operator [21], [7], [33] is used to preserve the tuples of the left relation R since only $R.X$ is projected. As in [22], the *IF-THEN-ELSE* structure has two components separated by a colon and the *WHERE* clause is evaluated as follows: If a tuple r of R joins with a tuple $t = (u, A'(u))$ of T_2, that is, $r.U \equiv u$, the degree for r to satisfy the outer join predicate is the degree by which r satisfies the first component in the square brackets, thus, r satisfies the query condition with the degree

$$d'_r = min(\mu_R(r), D(A'(u)), d(p_1(r)), d(r.Y\ op_1\ A'(u))).$$

Otherwise, the degree for r to satisfy the outer join predicate will be the degree by which r satisfies the second component in the square brackets, that is,

$$d'_r = min(\mu_R(r), d(p_1(r)), d(r.Y\ op_1\ 0)).$$

After eliminating duplicates, a value x is in the answer with the degree $\mu_{COUNT'}(x) = \max_{r.X \equiv x}(d'_r)$.

Theorem 6.1. *Query COUNT′ (Query JA′) is equivalent to Query JA if AGG is (not) COUNT.*

Proof. We shall prove that $d_r = d'_r$ for each tuple r of R, which implies $\mu_{JA}(x) = \mu_{JA'}(x)$ for each x in the domain of $R.X$. Recall that $T(r)$ and $T'(r.U)$ are identical for every r in R.

We first consider the situation where AGG is not $COUNT$. If $T(r)$ is empty, $A(r) \equiv null$ and $d_r = 0$. Since $T'(r.U)$ and $T(r)$ are identical, relation T_2 in the unnested query has no tuple $(r.U, A'(r.U))$. As a result, $d(r.U \equiv u) = 0$ for every tuple $(u, A'(u))$ in T_2 and $d'_r = 0$. If $T(r)$ is not empty, we have

$$A(r) \equiv A'(r.U), D(A(r)) = D(A'(r.U))$$

since the same aggregate function AGG is applied to the identical sets $T(r)$ and $T'(r.U)$. Since there is exactly one tuple, namely $(r.U, A'(r.U))$, in T_2 such that $d(r.U \equiv r.U) = 1$, we have $d_r = d'_r$.

Now, assume that AGG is $COUNT$. If $T(r)$ is empty, we have $A(r) = 0$ and

$$d_r = min(\mu_R(r), d(p_1(r)), d(r.Y\ op_1\ 0)).$$

Since $T'(r.U)$ is also empty, T_2 does not have the tuple $(r.U, A'(r.U))$. Thus, in Query COUNT′, tuple r does not join with any tuple of T_2, and

$$d'_r = min(\mu_R(r), d(p_1(r)), d(r.Y\ op_1\ 0)).$$

The rest of the proof is similar to that for the case where AGG is not $COUNT$. □

Although the unnested Query JA consists of three queries instead of one, by pipelining the result of one query to another, the three flat queries can be evaluated in parallel in the main memory. The evaluation is similar to that of the merge-join. First, the set of R-tuples satisfying p_1 is sorted on $R.U$ and the set of S-tuples satisfying p_2 is sorted on $S.V$. To simplify the discussion, assume that op_2 is the equality. Let r_1 be the first tuple in the sorted R. Then, $u_1 = r_1.U$ is the smallest $R.U$ value (according to $\prec$) and becomes the first tuple in T_1. As soon as u_1 is obtained, it is pipelined to Query T_2 and joins with $Rng(r_1)$ to generate $T'(u_1)$. If $T'(u_1) \neq \emptyset$, the aggregate function is applied to obtain $A'(u_1)$ and $D(A'(u_1))$. Once the two values are obtained, they are immediately pipelined to Query JA′ or

Query COUNT′, depending on the aggregate function. Then, for all R-tuples r with $r.U \equiv u_1$, including r_1, the degree d'_r is computed and the value $r.X$ is projected accordingly. If $T'(u_1) = \emptyset$ and AGG is $COUNT$, the left outer join in Query COUNT′ is evaluated, and $r.X$ may still be retrieved for a tuple r with $r.U \equiv u_1$. If $T'(u_1) = \emptyset$ and AGG is not $COUNT$, the processing for u_1 ends. This process is then repeated for the next tuple in the sorted R and so on until all R-tuples are processed. Since the operations are pipelined, this process is essentially the extended merge-join. Assume that the time complexity of the AGG function is $O(m)$, where m is the number of values in the group to which AGG is applied. Since, by assumption, each tuple of R joins with a constant number of tuples of S, the time spend on the AGG function in the process is in the order of $O(n_1)$ and the response time for Query JA′ (Query COUNT′) is still in the order of $O(n_R \log n_R + n_S \log n_S)$. Since Query JA can only be evaluated using the nested loop method, its response time is of order $O(n_R \times n_S)$.

7 Nested Queries with Quantifier All

In this section, we consider the unnesting of a type JALL query, which has the quantifier ALL in the outer block and a join predicate in the inner block referencing the outer relation. Nested queries with quantifier $EXIST$ or $SOME$ can be unnested similarly. The basic method is similar to that in [15], namely, to replace the quantifier with an appropriate aggregate function.

The following Query JALL is a type JALL nested query:

```
Query JALL
          SELECT    R.X
          FROM      R
          WHERE     R.Y <  ALL
                    (SELECT = S.Z
                    FROM     S
                    WHERE    S.V = R.U)
```

Although, for simplification, we have chosen not to include predicates p_1 and p_2 and to use $<$ instead of the more general op, the results of this section hold for the more general cases.

The execution semantics of Query JALL is as follows: For every tuple r of R, a temporary relation $T(r)$ is produced in the inner block. For each tuple s of S, $s.Z$ belongs to $T(r)$ with the degree $d_r(s) = min(\mu_S(s), d(r.U = s.V))$. After removing the duplicates, a value z in $T(r)$ has the degree $\mu_{T(r)}(z) = max_{s.Z \equiv z}(d_r(s))$. The degree for r to satisfy the query condition is $d_r = min(\mu_R(r), d(r.Y < ALL\ T(r)))$, where, for a value v and a set F, $d(v < ALL\ F)$ is given by

$$d(v < ALL\ F) = \begin{cases} 1 - \max_{x \in F}(min(\mu_F(x), 1 - \mu_<(v, x)), & F \neq \emptyset; \\ 1, & \text{otherwise.} \end{cases}$$

If $d_r > 0$, $r.X$ is in the answer with degree d_r. After removing duplicates, each value x in the answer has the degree $\mu_{JALL}(x) = \max_{r.X \equiv x}(d_r)$.

Query JALL can be unnested to the following Query JALL′ which uses a temporary relation T_1:

```
T1(K,X,D)= (SELECT     R.K, R.X, MIN(D)
           FROM        R,S
           WHERE       R.D AND ¬ (S.D  AND
                       R.U = S.V  AND ¬(R.Y> S.Z))
           WITH        D ≥ 0
           GROUPBY     R.K)
```

```
Query JALL′
      SELECT   T1.X
      FROM     T1
```

The execution semantics of JALL′ is the following. For each r in R and each s in S, the degree for r and s to satisfy the selection condition for T_1 is

$$d'_{r,s} = min(\mu_R(r), 1 - min(\mu_S(s), d(r.U = s.V), 1 - d(r.Y < s.Z))).$$

As indicated by the WITH clause, a pair of tuples (r, s) is kept around as long as $d'_{r,s} \geq 0$. Then, because of the GROUPBY clause and the MIN(D) aggregate function, the minimum $d'_{r,s}$ among tuple pairs (r, s) that have the same R-tuple is obtained by $d'_r = \min_{s \in S}(d'_{r,s})$. If $d'_r > 0$, a tuple $(r.K, r.X)$ is included in T_1 with d'_r as the membership degree. Query JALL′ then retrieves distinct values in $T_1.X$ by eliminating redundant values and, for each value x in the answer, the membership degree of x is given by $\mu_{JALL'}(x) = \max_{r.X \equiv x}(d'_r)$.

Theorem 7.1. *Query JALL′ is equivalent to Query JALL.*

Proof. We shall prove that $\mu_{JALL'}(x) = \mu_{JALL}(x)$ for each value x of $R.X$. It suffices to show that $d'_r = d_r$ for each tuple r of R. Notice that the following proof does not depend on that the predicate involving ALL is $<$. There are two cases.

Case 1: **$T(r)$ is empty.** In this case, for every s in S, $d_r(s) = 0$, that is, $d(r.U = s.V) = 0$. This implies that

$$d_r = min(\mu_R(r), d(r.Y < ALL\ T(r))) = \mu_R(r)$$

since $d(r.Y < ALL\ \emptyset) = 1$. For the unnested query, since $d(r.U = s.V) = 0$, $d'_{r,s} = \mu_R(r)$, for every s in S. Thus, $d'_r = \min_{s \in S}(\mu_R(r)) = \mu_R(r)$.

Case 2: **$T(r)$ is not empty.** For the nested query, by substituting $\mu_{T(r)}(z)$ into $d(r.Y < ALL\ T(r))$, and $d(r.Y < ALL\ T(r))$ into d_r, we have:

$$
\begin{aligned}
d_r &= min(\mu_R(r), d(r.Y < ALL\ T(r))) \\
&= min(\mu_R(r), 1 - \max_{z \in T(r)}(min(\mu_{T(r)}(z), 1 - d(r.Y < z)))) \\
&= min(\mu_R(r), 1 - \max_{z \in T(r)}(min(\max_{s.Z \equiv z}(min(\mu_S(s), \\
&\qquad d(s.V = r.U))), 1 - d(r.Y < z)))) \\
&= min(\mu_R(r), \min_{z \in T(r), s.Z \equiv z}(max(max(1 - \mu_S(s), \\
&\qquad 1 - d(s.V = r.U), d(r.Y < z))))) \\
&= min(\mu_R(r), \min_{s \in S}(max(1 - \mu_S(s), 1 - d(s.V = r.U), \\
&\qquad d(r.Y < z)))) \\
&= \min_{s \in S}(min(\mu_R(r), max(1 - \mu_S(s), 1 - d(s.V = r.U), \\
&\qquad d(r.Y < z)))) \\
&= \min_{s \in S}(min(\mu_R(r), 1 - min(\mu_S(s), d(s.V = r.U), \\
&\qquad 1 - d(r.Y < z)))) \\
&= d'_r.
\end{aligned}
$$

□

By pipelining the evaluation of T_1 with that of Query JALL′ in a similar way as described in Section 6, Query JALL′ can also be evaluated in time $O(n_R \log n_R + n_S \log n_S)$. Since Query JALL can be evaluated only by the nested loop method, its response time is $O(n_R \times n_S)$.

8 General Nested Fuzzy Queries

For conventional databases, the issue of unnesting general nested queries has been studied in [18], [8], [15], [22]. A complete discussion of unnesting general nested fuzzy queries is beyond the scope of the paper. In this section, we consider a subclass of nested fuzzy queries known as the *chain queries* or *linear queries* [22]. The following query is a chain query of three blocks.

```
Query 6
  SELECT  $R_1.X_1$
  FROM    $R_1$
  WHERE   $p_1(R_1)$ AND $R_1.Y_1$ in
          ( SELECT $R_2.X_2$
            FROM   $R_2$
            WHERE  $p_2(R_2)$ AND $R_2.U_2 = R_1.U_1$
                          AND $R_2.X_2$ in
                   ( SELECT  $R_3.X_3$
                     FROM    $R_3$
                     WHERE   $p_3(R_3)$
                             AND $R_3.V_3 = R_1.V_1$
                             AND $R_3.W_3 = R_2.W_2$ ) )
```

In general, a chain query Q_K has $K \geq 2$ subquery blocks, one block per nesting level. The blocks are numbered sequentially from the outermost to the innermost, starting with 1. The relation in block i is denoted by R_i. The query contains no set-exclusive predicate, aggregate function, or quantifier. However, a block may have a correlation predicate referencing a relation in any of its outer blocks. A correlation predicate in block i referencing relation R_j is denoted by $p_{i,j}(R_i, R_j)$. If a block does not have a correlation predicate referencing a particular outer block, we simply add the predicate $TRUE$ in its place, which will always be completely satisfied. Thus, in general, block k has the following typical structure:

```
SELECT  $R_k.X_k$
FROM    $R_k$
WHERE   $p_k(R_k)$ AND $\Pi_{1 \leq i \leq k-1}(p_{k,i}(R_k, R_i))$ AND
        $R_k.Y_k$ in = ( SELECT $R_{k+1}.X_{k+1}$ ... )
```

where $\Pi_{1 \leq i \leq K}(p_i)$ denotes the conjunction of predicates p_1 AND p_2 AND $\cdots$ AND p_K.

Query Q_K can be unnested to the following query Q'_K.

```
Query $Q'_K$
    SELECT  $R_1.X_1$
    FROM    $R_1, R_2, \ldots, R_K$
    WHERE   $\Pi_{1 \leq i \leq K}(p_i(R_i))$
                  AND $\Pi_{2 \leq i \leq K}\Pi_{1 \leq j \leq i-1}(p_{i,j}(R_i, R_j))$
            AND $\Pi_{1 \leq i \leq K-1}(R_i.Y_i = R_{i+1}.X_{i+1})$
```

Theorem 8.1. *Query Q'_K is equivalent to Query Q_K.*

Proof. We shall prove the theorem by induction. For $K = 2$, Q_2 is Query J (or Query N if $p_{2,1}(R_2, R_1)$ is $TRUE$) and Q'_2 is Query J′ (or Query N′). By Theorem 4.2 (or Theorem 4.1), Q'_2 is equivalent to Q_2.

Now, let $K \geq 3$, and assume that Query Q'_{K-1} is equivalent to Query Q_{K-1}. We shall prove that Q'_K is equivalent to Q_K.

Query Q_K can be written in the following form, where $Q_{K-1}(R_2, R_K)$ denotes the $(K-1)$ inner blocks of Q_K:

```
Query $Q_K$
    SELECT  $R_1.X_1$
    FROM    $R_1$
    WHERE   $p_1(R_1)$ AND $R_1.Y_1$ in $Q_{K-1}(R_2, R_K)$
```

For each tuple r_1 of R_1, $Q_{K-1}(R_2, R_K)$ is to be evaluated. When evaluating $Q_{K-1}(R_2, R_K)$ with respect to r_1, each appearance of R_1 is replaced by r_1 and each predicate $p_{i,1}(R_i, r_1)$, $2 \leq i \leq K$, can be considered as a predicate involving only R_i but with a constant r_1. Therefore, $Q_{K-1}(R_2, R_K)$ is a chain query of $K - 1$ blocks with a parameter R_1. By induction hypothesis, $Q_{K-1}(R_2, R_K)$ is equivalent to the following query:

```
Query $Q'_{K-1}(R_2, R_K)$
      SELECT  $R_2.X_2$
      FROM    $R_2, R_3, \ldots, R_K$
      WHERE   $\Pi_{2 \leq i \leq K}(p_i(R_i))$ AND
                    $\Pi_{2 \leq i \leq K}\Pi_{1 \leq j \leq i-1}(p_{i,j}(R_i, R_j))$
              AND $\Pi_{2 \leq i \leq K-1}(R_i.Y_i = R_{i+1}.X_{i+1})$
```

and the original query Q_K is equivalent to

```
Query $Q^1_K$
    SELECT  $R_1.X_1$
    FROM    $R_1$
    WHERE   $p_1(R_1)$ AND $R_1.Y_1$ in $Q'_{K-1}(R_2, R_K)$
```

In Query Q^1_K, for each tuple r_1 of R_1, the inner query $Q'_{K-1}(R_2, R_K)$ is evaluated with respect to r_1 and a temporary relation $T^1(r_1)$ is produced. For each combination of $r_i \in R_i$, $2 \leq i \leq K$, the degree for r_i, $2 \leq i \leq K$, to satisfy the selection condition of Q'_{K-1} is

$$\begin{aligned} d^1_{r_1}(r_2, \ldots, r_K) = \min_{2\leq i\leq K} \min_{1\leq j\leq i-1} (&\mu_{R_i}(r_i), d(p_i(r_i)), \\ &d(p_{i,j}(r_i, r_j)), d(r_i.Y_i = r_{i+1}.X_{i+1})). \end{aligned}$$

The value $r_2.X_2$ is in $T^1(r_1)$, if $d^1_{r_1}(r_2, \ldots, r_K) > 0$. After eliminating duplicates, each value z of $R_2.X_2$ in $T^1(r_1)$ will have with the degree

$$\mu_{T^1(r_1)}(z) = \max_{r_2.X_2 \equiv z} (d^1_{r_1}(r_2, \ldots, r_K)).$$

Therefore, r_1 satisfies Query Q^1_K with the degree

$$\begin{aligned} d^1_{r_1} &= min(\mu_{R_1}(r_1), d(p_1(r_1)), d(r_1.Y_1 \text{ in } T^1(r_1))) \\ &= min(\mu_{R_1}(r_1), d(p_1(r_1)), \max_{z\in T^1(r_1)} (min(\mu_{T^1(r_1)}(z), \\ &\quad d(r_1.Y_1 = z)))) \\ &= min(\mu_{R_1}(r_1), d(p_1(r_1)), \\ &\quad \max_{z\in T^1(r_1)} (min(\max_{r_2.X_2\equiv z} (d^1_{r_1}(r_2, \ldots, r_K)), d(r_1.Y_1 = z)))) \\ &= \max_{z\in T^1(r_1)} (min(\mu_{R_1}(r_1), d(p_1(r_1)), \\ &\quad min(\max_{r_2.X_2\equiv z} (d^1_{r_1}(r_2, \ldots, r_K)), d(r_1.Y_1 = z)))) \\ &= \max_{z\in T^1(r_1)} (\max_{r_2.X_2\equiv z} (\min_{2\leq i\leq K} \min_{1\leq j\leq i-1} (\mu_{R_1}(r_1), d(p_1(r_1)), \\ &\quad d(r_1.Y_1 = r_2.X_2), \mu_{R_i}(r_i), d(p_i(r_i)), d(p_{i,j}(r_i, r_j)), \\ &\quad d(r_i.Y_i = r_{i+1}.X_{i+1}))). \end{aligned}$$

The value $r_1.X_1$ is in the answer if $d^1_{r_1} > 0$. After eliminating duplicates, each value x of $R_1.X_1$ is in the answer with the degree $\mu_{Q^1_K}(x) = \max_{r_1.X_1\equiv x}(d^1_{r_1})$.

In Query Q'_K, for each combination of tuples $r_i \in R_i$, $1 \leq i \leq K$, the degree for them to satisfy the query condition is

$$\begin{aligned} d'(r_1, r_2, \ldots, r_K) = \min_{2\leq i\leq K} \min_{1\leq j\leq i-1} (&\mu_{R_1}(r_1), d(p_1(r_1)), \\ &d(r_1.Y_1 = r_2.X_2), \mu_{R_i}(r_i), d(p_i(r_i)), \\ &d(p_{i,j}(r_i, r_j)), d(r_i.Y_i = r_{i+1}.X_{i+1})). \end{aligned}$$

The $r_1.X_1$ belongs to the answer if $d'(r_1, r_2, \ldots, r_K) > 0$. After eliminating duplicates, each value x of $R_1.X_1$ in the answer has the degree

$$\mu_{Q'_K}(x) = \max_{r_1.X_1\equiv x} (d'(r_1, r_2, \ldots, r_K)).$$

Thus, $\mu_{Q'_K}(x) = \max_{r_1.X_1\equiv x}(*)$ and

$$\mu_{Q^1_K}(x) = \max{}_{r_1 \cdot X_1} \equiv x(\max{}_{z\in T^1(r_1)}(\max{}_{r_2 \cdot X_2} \equiv z(*))),$$

where $*$ denotes the common expression of the two degrees. The expression of $\mu_{Q'_K}(x)$ covers all K-tuple combinations $r_i \in R_i$, $1 \leq I \leq K$, in which $r_1.X_1 = x$. But, some of these combinations may not be covered by $\mu_{Q_K}(x)$ if $r_2.X_2$ is not in $T^1(r_1)$. However, for such a combination, $d^1_{r_1}(r_2, \ldots, r_K) = 0$, hence, the common expression $*$ yields 0 as well. Since $*$ always evaluates to a nonnegative value, we have $\mu_{Q'_K}(x) = \mu_{Q^1_K}(x)$. □

TABLE 1
Response Time in Seconds of the Nested Loop and Merge-Join Methods

Relation Size	1MB	2MB	4MB	8MB	16MB	32MB
Nested Loop	501	1965	7754	30879	—	—
Merge-join	40	84	223	852	1897	3733
Speedup	12.5	23.4	34.8	36.2	—	—

To evaluate Query Q'_K, an optimal join order may be determined by using, say, a dynamic programming [35] method, to minimize the sizes of the intermediate relations. If, as assumed, each tuple of a relation joins with a constant number of tuples of another relation, the size of an intermediate relation will be proportional to a joining relation, and the response time of Q'_K using the extended merge-join method will be of order $O(\Sigma_{1\leq i\leq K} n_i \log n_i)$, where n_i is the number of tuples of relation R_i. The response time of Q_K using the nested loop method could be of order $O(\Pi_{1\leq i\leq K} n_i)$.

9 Experimental Results

We conducted experiments to study the performance of the unnesting techniques. In this section, the type J queries are used to illustrate the experimental results.

The experiments are conducted on a SUN SPARC/IPC workstation, which has an 8-Megabyte main memory and is dedicated to the experiments. Both the nested loop and the merge-join methods are implemented using the Omron fuzzy database library [25]. A 2-Megabyte buffer is available to both methods. For the nested loop method, one buffer page (8 k-bytes) is allocated to the inner relation and the rest to the outer relation in order to minimize I/O cost [14]. For the merge-join, the sorting is done by Opt-Tech Sort [26], a commercial external sorting software that uses a user-specified amount of memory. Tuples of the relations are randomly generated and a tuple of one relation joins, on the average, C tuples of the other relation. Both the I/O and the CPU costs are measured. The experimental results confirm that the extended merge-join outperforms the nested loop method by an order of magnitude.

Four experiments are discussed below. In the first experiment, both R and S relations contain n tuples of 128 bytes, where n ranges from 8,000 to 256,000 resulting in relations of a size from 1 to 32 MB. Each tuple of R joins, on the average, with seven tuples of S. Table 1 shows the response times of the two methods in seconds and indicates that the speedup of the extended merge-join with respect to the nested loop method is from 12 to more than 36 as the size of the relations increases. For a relation size no less than 16MB, the nested loop method takes too long to terminate.

For the second experiment, the size of the outer relation is fixed at 4MB and that of the inner relation is ranging from 2 to 16 MB. The tuple size and the value C are the same as those in the first experiment. The result, shown in Table 2 indicates that the response time of the nested loop method increases linearly with the size of the inner relation. This is consistent with our analytical results. For the merge-join, the numbers do not closely match the analytical results,

TABLE 2
Response Time in Seconds: Changing the Size of Inner Relation

Inner Relation Size	2MB	4MB	8MB	16MB
Nested Loop	3912	7790	15489	31049
Merge-join	156	205	476	2152
Speedup	25.1	38	32.5	14.4

namely $O(n\ log\ n)$. In order to understand the behavior of the algorithm, we collected more detailed information about the algorithm, as shown in Table 3, where the first row represents the CPU time spent on sorting, merging, and joining as a percentage of response time and the second row represents the percentage of response time spent on sorting (including CPU time and IO time). The results show that, as the size of the inner table increases, the join becomes more IO intensive and the majority of the time is spent on sorting. The jump of the response time near the columns of 8MB and 16MB we believe is caused by the memory management of the operating system. The speedup increases as the relation size increase until the size reaches 4MB and then, decreases afterward. This change is expected. Since one relation has a fixed size, the complexity of the nested loop becomes $O(n)$, while that of merge-join remains $O(n \log n)$. As a result, the improvement decreases as the size of the inner relation increases.

The third experiment was designed to test the impact of IO activities. The number of tuples of the relations is fixed to 8,000, but the tuple size is ranging from 128 bytes to 2,048 bytes. A tuple in R joins, on the average, with one tuple in S. The results, shown in Table 4, indicate that the merge-join is superior to the nested loop. Since the number of tuples is fixed, the CPU time spent on comparisons and fuzzy computation remains unchanged for both algorithms (15 seconds for the merge-join and 483 seconds for the nested loop). As the tuple size increases, the number of IOs also increases. As a result, the percentage of CPU time drops for both algorithms. We should also point out that the total CPU time also increases since, as the tuple size increases, more CPU time is needed for handling IOs.

The last experiment was designed to study the impact of the number of joining tuples on the performance. Both relations have a fixed size of 8MB (64,000 tuples). The average number C of tuples that a tuple will join ranges from 1 to 128. The results are show in Fig. 3. As C increases, the number of IOs remains more or less the same, but the CPU time increases due to the increase in the number of calls to the fuzzy library functions and the number of comparisons for merge and join.

TABLE 3
Time Break Down for Merge-Join Method

Inner Relation Size	2MB	4MB	8MB	16MB
CPU time (%)	76	63	51	24
Sorting Time (%)	38.7	52.5	61.9	84.1

TABLE 4
Response Time in Seconds: Changing Tuple Size

Tuple Size	128	256	512	1024	2048
Nested Loop	485	514	584	729	1077
merge–join	20	37	94	487	896

It should be pointed out that, in these experiments, both the intervals associated with the join attribute values and the average numbers of joining tuples are kept small. This is typical for fuzzy database applications in which data may be imprecise but not very vague. On the other hand, in temporal database applications, the intervals associated with the tuples can be much larger. This could have an adverse effect on the merge-join method.

10 Summary

In this paper, we present techniques to unnest various types of 2-level, and a subclass of K-level nested Fuzzy SQL queries for efficient evaluation. An extended merge-join is used to evaluate the unnested queries, and its performance is compared with that of the nested loop method which the nested queries must be evaluated with. Both analytical and experimental results regarding the performance of the two methods are presented in the paper. The techniques are likely to be applicable in database systems that allow vague queries on uncertain and imprecise data. To the best of our knowledge, the issues of unnesting fuzzy queries have not been studied before.

We will continue investigating techniques for optimal processing of fuzzy queries and study the application of those techniques to other types of databases with imprecise information. One such application is the picture retrieval [2].

Appendix

Interpretation of a Fuzzy Query

In this Appendix, we provide a clarification of the meaning of a Fuzzy SQL query.

A fuzzy relation, as defined in Section 2 of the paper, is a fuzzy set of tuples. That is, each tuple of a fuzzy relation is a member of the fuzzy set with a degree of belonging to the set. Based on the interpretation of possibility, we can think of a fuzzy relation as a representation of a fuzzy concept (or a fuzzy condition) and each tuple in the fuzzy relation as an object that possibly belongs to the concept (or satisfies the condition). Here, the membership degree of a tuple is interpreted as the possibility for the tuple to be a part of the concept (or to satisfy the condition). Thus, as stated in this paper, the answer to a Fuzzy SQL query is a fuzzy relation where each tuple satisfies the query condition to the extent as indicated by its membership degree (see Section 1) and, for the answer relation of a query, the membership degree of a tuple is interpreted as the degree for the tuple to satisfy the query condition (see Section 2).

For example, consider query Q: "select A from T where X," where A is a list of attributes, T is a set of fuzzy relations, and X is a query condition. The result is a fuzzy

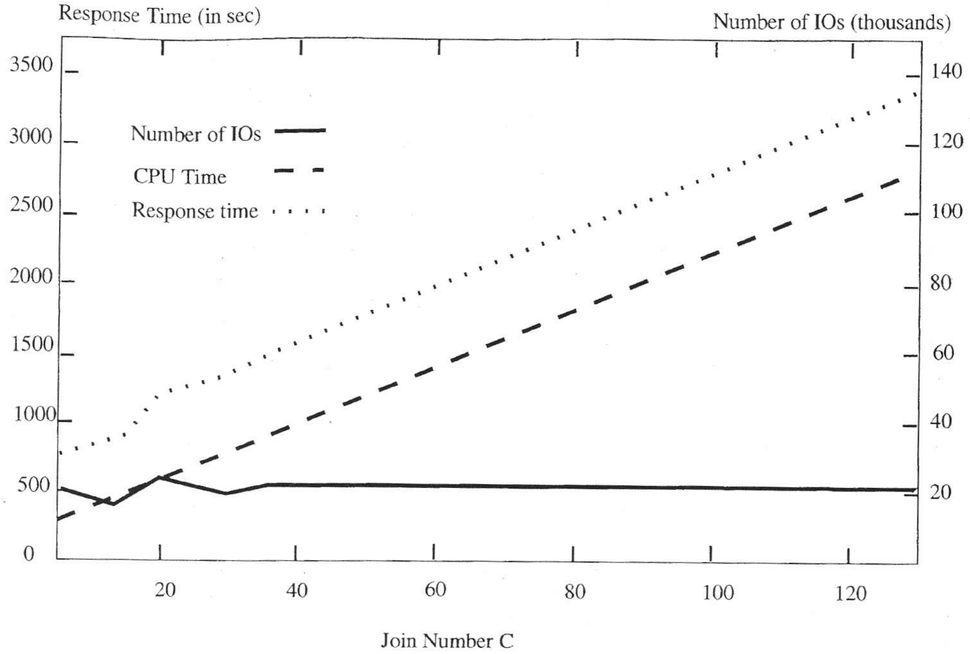


Fig. 3. Response time and number of IOs for merge-join.

relation R' such that each tuple r' in R' indicates a set of tuples in relations in T that collectively and possibly satisfies the condition X. In other words, the only thing in common among the tuples in the answer relation is that each one of them is a possible answer to the query (up to the membership degrees). Notice that this type of interpretation is a straightforward generalization of that of an ordinary, crisp query, for which each tuple in the result is a sure answer.

Consider another example. Let R and S be the following relations:

R	
X	Y
x_1	y_1
x_2	y_2

S	
Y	Z
$1/y_1 + .8/y_2$	z_1

Consider the query: "select $R.X$ from R, S where $R.Y = S.Y$."

First of all, the possibility distribution $1/y_1 + .8/y_2$ under attribute Y in relation S indicates an uncertain value of Y (which is possibly y_1 with possibility 1 or y_2 with possibility 0.8). It indicates that we do not know the precise value of Y, but it is possibly y_1 or y_2, therefore, both y_1 and y_2 are possible values of Y. With this uncertainty, we are unable to determine which tuple in R will actually join with the tuple in S. However, we are able to determine that either tuple in R has a possibility of joining with the tuple in S, thus, both of them are possible answers.

Thus, the semantics of the query: "select $R.X$ from R, S where $R.Y = S.Y$, should be to find all $R.X$ such that there exists some tuple in S and $R.Y = S.Y$ is satisfied (in a fuzzy sense) by these tuples of R and S. The answer obtained using the method presented in this paper is based on this interpretation. Thus, for this example, both x_1 and x_2 are possible answers, with a possibility 1 and 0.8, respectively.

It may be tempting to determine for the query which tuple of S will actually join with tuples of R. One way to do so may be the following: When joining R with S, each tuple in S is replaced by one of its possible values in its possibility distribution. This replacement is repeated for every combination of possible values for the tuples in S for the join operation. In the previous example, we can assign $1/y_1$ to $S.Y$ in one join and assign $0.8/y_2$ to $S.Y$ in a second join. One join is performed for each combination of value assignments. Each of these joins results in a possible answer relation. Thus, the answer to the previous query would be two relations: One contains x_1 with a membership degree 1 and the other contains x_2 with a membership degree 0.8. However, this method not only still gives a fuzzy answer (now one has to determine which answer relation should be the "true" answer), but also it does so very inefficiently, as illustrated in the following example:

Suppose the two relations are as follows:

R	
X	Y
x_1	y_1
x_2	y_2
x_3	y_3
x_4	y_4

S	
Y	Z
$1/y_1 + .8/y_2$	z_1
$.9/y_3 + .7/y_4$	z_2

Consider the query "select $R.X$ from R, S, where $R.Y = S.Y$."

Using the method just outlined, the answer should be one of the four fuzzy sets $\{1/x_1, 0.9/x_3\}$, $\{1/x_1, 0.7/x_4\}$, $\{0.8/x_2, 0.9/x_3\}$, and $\{0.8/x_2, 0.7/x_4\}$. Thus, the answer becomes a fuzzy set of fuzzy sets (second order). There are several reasons why this is not a good approach.

1. The enumeration of all those (fuzzy) sets of answers does not provide much more information to the user as it is still uncertain which one of these sets is the sure answer.
2. The exponential nature exhibited in this example will only get worse when the values of Y are given by possibility density functions in both relations (say, on the real numbers) for, in that case, there will be an infinite number of fuzzy sets in the answer (notice that not only each x-value in a set has a membership degree, but also the set itself has a membership degree). Therefore, it is not practical to compute and present such an answer to the user.
3. Since, under this interpretation, each algebraic operation, such as selection and join, will result in multiple relations, the algebraic operations can not be composed, as discussed earlier in the paper.

Acknowledgments

The authors would like to thank the anonymous referees for their valuable comments and constructive suggestions on the presentation and technical accuracy of the paper. This research was supported in part by the US National Science Foundation (NSF) under IRI-9509253, Omron Corporation, Omron Management Center of America, and the Natural Science and Engineering Research Council (NSERC) of Canada. An earlier version of this paper was presented at the IEEE International Conference on Data Engineering, 1995.

References

not our grants.

[1] S. Abiteboul, P. Kanellaki
Representation and Queryin
Theoretical Computer Science, vol

[2] A. Aslandogan, C. Thier, C. Y
Implementation and Evaluatio
based REtrieval of Pictures)," *I*

[3] J.F. Baldwin, "A Fuzzy Relatio
Systems," *Proc. 13th IEEE Int'l S*
423, 1983.

[4] P. Bosc, M. Galibourg, and C
SQL: Extensions and Implem
Systems, 1988.

[5] P. Boscand and O. Pivert, "
Flexible Querying in Databa
and Soft Computing, R.R. Yag
pp. 368-395, New York: Van

[6] P. Bosc and O. Pivert, "SQLf:
Fuzzy Querying," *IEEE Trans.*
1995.

[7] E.F. Codd, "Extending the Da
More Meaning," *ACM Trans.*

[8] U. Dayal, "Of Nests and Trees: A Unified Approach to Processing Queries that Contain Nested Queries, Aggregates, and Quantifiers," *Proc. Very Large Databases,* 1987.

[9] D. DeWitt, J. Naughton, and D.A. Schneider, "An Evaluation of Non-Equijoin Algorithms," *Proc. Very Large Databases,* 1991.

[10] D. Dubois and H. Prade, *Possibility Theory: An Approach to Computerized Processing of Uncertainty.* chap. 6, New York: Plenum Press, 1988.

[11] D. Dubois and H. Prade, "Measuring Properties of Fuzzy Sets: A General Techniques and Its Use in Fuzzy Query Evaluation," *Fuzzy Sets and Systems,* vol. 38, pp. 137-152, 1990.

[12] D. Dubois and H. Prade, "Fuzzy Sets—A Survey of Eng. Applications," *Computer and Chemical Eng.,* vol. 17, pp. 373-380, 1993.

[13] P.F. Fisher, "First Experiments in Viewshed Uncertainty—Simulating Fuzzy Viewsheds," *Photogrammetric Eng. and Remote Sensing,* vol. 58, no. 3, pp. 345-352, Mar. 1992.

[14] G. Graefe, "Query Evaluation Techniques for Large Databases," *Computer Surveys,* June 1993

[15] R.A. Ganski and H.K.T. Wong, "Optimization of Nested SQL Queries Revisited," *Proc. ACM SIGMOD,* 1987.

[16] R.L. Haar, "A Fuzzy Relational Data Base System," Technical Report, TR-587,Computer Center, Univ. Maryland, Sept. 1977.

[17] T. Imielinski and W. Lipski Jr., "Incomplete Information in Relational Databases," *J. ACM,* vol. 31, 1984.

[18] W. Kim, "On Optimizing an SQL-like Nested Query," *ACM Trans. Data Systems,* Sept. 1982.

[19] G.M. Lohman et al., "Optimization of Nested Queries in a Distributed Relational Database," *Proc. Very Large Databases,* 1984.

[20] D. Li and D. Liu, *A Fuzzy Prolog Database System,* Taunton, England: Research Studies Press, 1990.

[21] M. Lacroix and A. Pirotte, "Generalized Joins," *SIGMOD Record,* vol. 8, no. 3, Sept. 1976.

[22] M. Muralikrishna, "Improved Unnesting Algorithms for Join Aggregate SQL Queries," *Proc. Very Large Databases,* 1992.

[23] H. Nakajima, T. Sogoh, and M. Arao, "Development of an Efficient Fuzzy SQL for Large Scale Fuzzy Relational Database," *Proc. Fifth Int'l Fuzzy Systems Assoc. World Congress '93,* 1993.

[24] E. Omiecinski and E.T. Lin, "The Adaptive-Hash Join Algorithms for a Hypercube Multicomputer," *IEEE Trans. Parallel and Distributed Systems,* 1992.

[25] *Fuzzy LUNA — Fuzzy Database System Library User's Manual,* and *Fuzzy LUNA Fuzzy Database System Library Reference Manual,* OMRON Corp. 1992.

[26] Opt-Tech Data Processing, Inc., Opt-Tech Sort User's Manual, Version 1.7, 1992.

[27] F. Petry, *Fuzzy Databases: Principles and Applications,* Kluwer Academic, 1996.

[28] H. Prade and C. Testemale, "Generalizing Database Relational Algebra for the Treatment of Incomplete or Uncertain Information and Vague Queries," *Information Sciences,* vol. 34, pp. 115-143, 1984.

[29] H. Prade and C. Testemale, "Fuzzy Relational Databases:
sues and Reduction Using Similarity Mea-
nformation Science, vol. 38, no. 20, pp. 118-126,

estemale, "Representation of Soft Constraints
te Values by Means of Possibility Distribution
alysis of Fuzzy Information, Vol. II: Artificial
sion Systems, J.C. Bezdek, ed. pp. 213-229, Boca
ress, 1987.

er and L. Bic, "Extending Fuzzy Relational
by Aggregates," *Proc. North Am. Fuzzy Informa-*
, pp. 201-205, June 1988.

er and L. Bic, "Aggregates in Possibilistic
15th Int'l Conf. Very Large Data Bases, pp. 287-

D. Reiner, "Extending the Algebraic Framework
ing To Handle Outer-Joins," *Proc. Very Large*

Melton, "An Extended Version of the Fuzzy
se Model," *Information Science,* 1990.

, and C. Yu, "Query Optimization in Distributed
)atabase Systems," *Computer J.,* pp. 98-107, 1992.

rass, and C. Jensen, "Efficient Evaluation of the
ral Join," *Proc. 10th Int'l Conf. Data Eng.,* Feb.

[37] B. Salzberg, A. Tsukerman, J. Gray, M. Stewart, S. Uren, and B. Vaughan, "FastSort: A Distributed Single-Input Single-Output External Sort," *Proc. SIGMOD* 1990.

[38] M. Umano and S. Fukami, "Fuzzy Relational Algebra for Possibility-Distribution-Fuzzy-Relational Model of Fuzzy Data," *J. Intelligent Information Systems,* no. 3, pp. 7-27, 1994.

[39] L.A. Zadeh, "Fuzzy Sets," *Information and Control,* vol. 8, pp. 338-353, 1965.

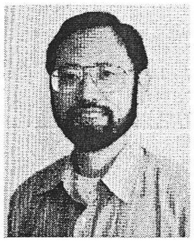
[40] L.A. Zadeh, "Fuzzy Sets as a Basis for a Theory of Possibility," *Fuzzy Sets and Systems,* vol. 1, no. 1, pp. 3-28, 1978.

[41] M. Zemankova and A. Kandel, "Implementing Imprecision in Information Systems," *Infomation Science,* 1985.

[42] W. Zhang and K. Wang, "An Efficient Evaluation of A Fuzzy Equi-Join Using Fuzzy Equality Indicators," *IEEE Trans. Knowledge and Data Eng.,* vol. 12, no. 2, pp. 225-237, Mar/Apr. 2000.

[43] W. Zhang, C. Yu, G. Wang, T. Pham, and H. Nakajima, "A Relational Model for Imprecise Queries," *Proc. Int'l Symp. Methodologies in Intelligent Systems,* 1993.

Qi Yang received the BS degree in mathematics in 1982 from Sichuan University, China, and the MS and PhD degrees in computer science in 1989 and 1994, respectively, from the University of Illinois at Chicago. Currently, he is teaching at the University of Wisconsin at Platteville.



Weining Zhang received the BEng degree in computer science and engineering from the Electronic Science and Technology University of China, People's Republic of China, in 1982, the MS and PhD degrees in computer science from the University of Illinois at Chicago, in 1985 and 1988, respectively. He is currently an associate professor in the Department of Computer Science, University of Texas at San Antonio. His research interests are in fuzzy databases, heterogeneous distributed databases, Web databases and data mining. He is a member of the IEEE Computer Society.

Chengwen Liu received the BS degree in electronics from Shandong University, China, in 1983; the MS degree in electrical and computer engineering from the Illinois Institute of Technology in 1986; and the PhD degree in computer science from the University of Illinois at Chicago in 1991. He is an associate professor in the School of Computer Science, Telecommunications, and Information Systems at DePaul University, Chicago. His research interests are in distributed database systems, fuzzy databases and data mining. Dr. Liu is a member of the ACM and has served as a member of the Publications Board of the IEEE Computer Society. He is a member of the IEEE.

Jing Wu biography and photo not available.

Clement Yu obtained the BS degree in applied mathematics from Columbia University in 1970 and the PhD degree in computer science from Cornell University in 1973. He is a professor in the Department of Computer Science at the University of Illinois at Chicago. His areas of interest include search engines and multimedia retrieval. He has publications in various journals such as *IEEE Transactions on Knowledge adn Data Engineering, ACM Transactions on Database Systems* and *JACM* and in various conferences such as VLDB, ACM SIGMOD and ACM SIGIR. He previously served as chairman of ACM SIGIR and as a member of the advisory committee to the US National Science Foundation. He is a member of the editorial board of the *IEEE Transactions on Knowledge adn Data Engineering, International Journal of Software Engineering,* and *Knowledge Engineering and Distributed and Parallel Databases.* He was cochair of the US National Science Foundation Information and Data Management Program Workshop for principal investigators for the year 2000 and is the cochair of the International Conference on Information Society in the 21st Century: Emerging Technologies and New Challenges to be held in Japan. He is a member of the IEEE Computer Society.

Hiroshi Nakajima received the BS degree in system engineering from Kobe University, Japan, in 1985. Since 1985, he has been with the Research and Development Division of OMRON Corporation, Kyoto, Japan, and has been working in the field of intelligent systems using artificial intelligence and cognitive science. He is currently a project leader in the Verbal Interaction Technology Laboratory of the Information Technology Research Center, Omron Corporation. His current research interests are human-machine interaction, intelligent agents, pet robots, and artificial mind models. He is a member of the IEEE Computer Society, IPS (Information Processing Society of Japan), and SOFT (Japan Society for Fuzzy Theory and Systems).

Naphtali David Rishe completed the PhD degree at Tel Aviv University in 1984. He worked as an assistant professor at the University of California, Santa Barbara (1984-1987), was an associate professor (1987-1992) and is currently a professor (1992-) at Florida International University (FIU). His expertise is in database management and methodology for the design of database applications. His work on the *Semantic Binary Database Model* was published as a book by Prentice Hall in 1988. His *Semantic Modeling* theory was published as a book by McGraw-Hill in 1992. His current research focuses on efficiency and flexibility of database systems (particularly of object-oriented, semantic, decision-support, and spatial/geographic DBMS), distributed DBMS, high-performance systems, database design tools, and Internet access to databases. He is the editor of four books and author of two patents, 24 papers in journals (including *IEEE Transactions on Knowledge and Data Engineering, Data and Knowledge Engineering, Information Systems, Fundamenta Informaticae*), seven chapters in books and serials (including three in Springer Verlag's *Lecture Notes in Computer Science*), three encyclopaedia articles, more than 80 papers published in proceedings (including ACM SIGMOD, VLDB, PDIS, IEEE DE, ACM SIGIR, SEKE, ARITH, FODO). Dr. Rishe has been awarded millions of dollars in research grants by government and industry. His research is currently sponsored by the US National Aeronautical Space Administration, (NASA) ($5.5M) the US National Science Foundation, (NSF) ($4M) BMDO, ARO, DoD, DoI, and other agencies. He also has extensive experience in database applications and database systems in industry. This includes eight years of employment as head of software and database projects (1976-1984) and later consulting for companies such as Hewlett-Packard and the telecommunications industry. He is the founder and director of the High Performance Database Research Center at FIU, which now employs 110 researchers, including 20 PhDs. Dr. Rishe chaired the program and steering committees of the PARBASE conference and is on the steering committee of the PDIS conference series.