Joining relations under discrete uncertainty

Matteo Magnani, Danilo Montesi *

March 19, 2016

Abstract

In this paper we introduce and experimentally compare alternative algorithms to join uncertain relations. Different algorithms are based on specific principles, e.g., sorting, indexing, or building intermediate relational tables to apply traditional approaches. As a consequence their performance is affected by different features of the input data, and each algorithm is shown to be more efficient than the others in specific cases. In this way statistics explicitly representing the amount and kind of uncertainty in the input uncertain relations can be used to choose the most efficient algorithm.

1 Introduction

One fundamental step of a traditional relational query optimization process is the choice of the best algorithm to execute each query operator. This is particularly important with regard to join operators, for which naive executions are computationally impractical and efficient algorithms have been developed. However, traditional join algorithms cannot be directly applied when the underlying model is extended to accommodate uncertain data.

In this paper we extend traditional join algorithms to work with uncertain data and discuss the outcomes of their experimental comparison. As a result we show that no algorithm is always more efficient than the others and we study their behavior with respect to parameters (statistics) that can be used to choose the best one depending on the input data.

In particular, we will focus on what is often called *discrete* uncertainty. As we will detail in Section 2 the design of a join algorithm depends on the underlying model used to represent uncertainty. Discrete uncertainty models assume to have finite sets of alternative values instead of single certain values, e.g., the set {45,46,47} to represent the (partially unknown) age of an individual, and are currently implemented in the main uncertain data management systems like

---

*M. Magnani is with the Department of Computer Science, Aarhus University, Denmark (magnanim@cs.au.dk), D. Montesi is with the Department of Computer Science, University of Bologna, Italy (montesi@cs.unibo.it)

1

xid	Name	Surname	Net Worth	Age
t1	William H.	Gates III	53	54
t2	Warren	Buffett	47	79
t3	Paul	Allen	14	57
t4	Lawrence	Page	18	37
t5	Lawrence	Ellison	28	65
t6	Michael	Dell	14	45

Figure 1: A table with the wealth of some notable American people in Billion USD (source: Forbes)

Trio [28] or Orion [7], where it is also possible to specify continuous uncertainty distributions.

In the following we introduce the problem of joining uncertain relations by example showing why traditional methods cannot be applied, and list the main contributions of this work. Then we present an overview of existing works on the topic. In Section 3 we introduce the algorithms analyzed in the paper, in Section 4 we report on their experimental comparison and in Section 5 we provide an interpretation of the results of the experimental evaluation. We conclude the article with a summary and discussion of the main results.

1.1 A brief recall of traditional join approaches and why they do not work on uncertain data

Consider the relational table illustrated in Figure 1, representing the wealth of some notable American people. In this paper we focus on the equi-join operator, i.e., a join where tuples are concatenated and included in the result when some of their attributes have the same value. For example, we may want to join this table with itself on the `Net Worth` attribute to pair people with the same wealth:

SELECT p1.surname, p1.networth,
p2.surname, p2.networth
FROM WEALTH p1, WEALTH p2
WHERE p1.networth = p2.networth

The result of this query is illustrated in Figure 2.

A well known naive approach to obtain this result, the one available on the early relational systems and known as *nested loop join*, consists in comparing each row of the first table (called *external* relation) with all rows of the other (*internal*): in our working example we would start comparing row t1 against t1, then (t1, t2), (t1, t3) and so on.

In Figure 3 we have represented the order of comparisons between tuples and the found matches. Evidently this method requires $n_1 \times n_2$ tuple comparisons where n_1 and n_2 are the cardinalities of the two input relations (36 comparisons

Surname	Net Worth	Surname	Net Worth
Gates III	53	Gates III	53
Buffett	47	Buffett	47
Ellison	28	Ellison	28
Page	18	Page	18
Allen	14	Allen	14
Allen	14	Dell	14
Dell	14	Dell	14
Dell	14	Allen	14

Figure 2: Result of a self-join

	t1	t2	t3	t4	t5	t6
t1	**1**	2	3	4	5	6
t2	7	**8**	9	10	11	12
t3	13	14	**15**	16	17	**18**
t4	19	20	21	**22**	23	24
t5	25	26	27	28	**29**	30
t6	31	32	**33**	34	35	**36**

Figure 3: Comparisons performed using a nested loop join. Numbers indicate the order of comparisons and boldface the matches found

in our example). Although potentially useful for small datasets or as a subroutine of more sophisticated join methods, this behavior does not scale to large datasets.

To reduce the number of comparisons several approaches have been proposed and are currently used in relational systems. All approaches are based on reorganizing the data such that for every row of one table there is no need to scan all the other table but we can look at specific locations where potentially matching records have been collected — this is usually done by sorting the tables, using hash functions, or building indexes on the join columns. In this way it is not necessary to check all the other rows. For example, consider Figure 4: here the input data have been sorted on the join attribute. When we compare row t6 (Michael Dell **14** 45) against row t4 (Lawrence Page **18** 37), which is the third comparison performed by this algorithm as indicated in the figure, we know that we will find no other matches, because all the following tuples in the internal input relation will have a value equal or greater than 18 and thus cannot match 14. As a consequence, we can proceed to row t5 without checking all other tuples in the second relation.

Finally, Figure 5 shows the comparisons performed using an index. Here we can use the index to obtain directly the pointers to the matching tuples, if present, but we need to access the index for each row in the external relation —

	t6	t5	t4	t3	t2	t1
t6	**1**	**2**	3			
t3	**4**	**5**	6			
t4	7	8	**9**	10		
t5			11	**12**	13	
t2				14	**15**	16
t1					17	**18**

Figure 4: Comparisons performed using a sort join algorithm. Notice that the input tuples have been sorted on the join attribute

	t1	t2	t3	t4	t5	t6
INDEX(t1) →	**1**					
INDEX(t2) →		**2**				
INDEX(t3) →			**3**			**4**
INDEX(t4) →				**5**		
INDEX(t5) →					**6**	
INDEX(t6) →			**7**			**8**

Figure 5: Comparisons performed using an index join. Matching tuples are identified directly, but every tuple in the external relation requires an access to the index. In this example we are using only the index on one of the two input relations

we will further discuss the impact of this approach when applied to uncertain data in the section on the experimental comparison of the algorithms.

Now assume that the table has been filled using a Web information extraction tool, or even a manual browsing of different Web sites. Given the uncertainty of the data generation process (first case) or the inconsistency of on-line information (second case) the collected data would be uncertain. As a working example in Figure 6 we have represented the same table of Figure 1 augmented with uncertain data coming from alternative Web sites.

Uncertainty is a state of limited knowledge about a past, current or future state of the world. Usually, uncertainty is represented using a set of alternative information items, e.g., {18,19} to represent the fact that the correct wealth of a person is either 18 or 19 million USD. In addition numerical or symbolic values are associated to these alternative elements, e.g., to indicate their probability or degree of possibility. However, in the following we use a possible world data model without numerical uncertainty measures. While computing a join, we first look for pairs of tuples to be concatenated and included in the result, and our algorithms aim at reducing the number of comparisons and I/Os. In this way, they can be used independently of the adopted uncertainty management theory — probabilities, possibilities or other measures can be computed after

xid	Name	Surname	Net Worth	Age
ut1	William H.	Gates III	{53,50,40,58}	54
ut2	Warren	Buffett	{47,40,37,42}	79
ut3	Paul	Allen	{14,16,22}	57
ut4	Lawrence	Page	{18,19}	37
ut5	Lawrence	Ellison	{28,23,25}	65
ut6	Michael	Dell	{14,16,18}	45

Figure 6: Our working example, containing a `NetWorth` uncertain attribute

the generation of the result.

Looking at the table in Figure 6 it appears why some efficient traditional methods can no longer be applied. For example, we can no longer sort its rows on the join attribute: both sets {18,19} (ut4) and {14,16,22} (ut3) contain values greater and smaller than the values of the other set. This motivates the extension of the aforementioned algorithms that will be presented in the next section and later experimentally evaluated.

1.2 Main contributions

In this article we provide the following main contributions:

- We define several algorithms to join uncertain relations. One algorithm is an extension of the relational sort join, and reduces to it when the data is certain. The other (tuple-based, presented in two variations) is an original algorithm that uses traditional joins between the simple tuples composing the uncertain relations. We also include in our evaluation an index-join, as suggested in other works, and two base algorithms (a nested loop approach and the one chosen by the underlying relational query optimizer used in our experiments) to be used as baselines. As we will see, except for the nested loop join no approach is always better than the others.

- We compare the algorithms on several data sets to find the relationships between the input data and their performance.

The main results obtained after the experimental evaluation of the proposed algorithms show that:

- Tuple-based approaches, which transform uncertain tables into larger traditional relational tables with one tuple for each alternative value, have a time complexity which is independent of the distribution of uncertain values in the data, assuming a linear complexity of the underlying relational joins. However, their performance is affected by the number of alternative values contained inside each uncertain tuple.

- When alternative values contained inside each uncertain tuple are similar, as it may be the case when we measure temperatures, lengths, etc., the

5

sort-based approach is almost as efficient as a traditional sort-join. There-fore, this algorithm can be either more or less efficient than the tuple-based approach, depending on: the percentage of uncertain tuples, the number of alternative values for each uncertain field, the distribution (spreading) of these values, the cardinality of the input relations and the number of required tuples in the result in case it is not necessary to build all the resulting relation.

- The sort-based approach improves significantly when we require only the first few tuples of the result, as it usually happens in modern database system GUIs where additional tuples are fetched into memory only if explicitly required by the user. However, in this case using an index results to be the most efficient approach.

- Using the most appropriate algorithm we are able to efficiently join tables containing millions of uncertain tuples.

2   Related work

Uncertain relational models have been studied since the early 90's, and the first works on this topic mainly focused on theoretical aspects of probabilistic and possibilistic data management [4, 18, 24, 9, 5, 17, 10]. More recently, there have been successful initiatives to build working systems for the efficient execution of queries over uncertain data [6, 7, 28, 1, 27, 25, 12].

In addition to specific works on uncertain data models and systems, this article builds over the large literature on relational join algorithms which is today consolidated and can be found in any text book on relational database management system architectures, e.g., [11]. With regard to traditional join algorithms, we have reported and exemplified the few concepts necessary to understand the remaining of the paper in the introduction.

On the contrary, the problem of direct optimization of uncertain data is more recent and to the best of our knowledge the first work suggesting the usage of specific statistics on the uncertainty of the input relations is [26]. Other works have also dealt with query optimization on probabilistic data without focusing on join algorithms [8, 25, 3].

Given the importance of the join operator a number of probabilistic join algorithms have been proposed in the literature, and are complementary to our work because they deal with different aspects of this problem, as follows.

Probabilistic joins are useful when objects may match *up to a certain degree*, differently from our work where we compute exact joins and the additional workload depends on our ignorance of the real values we are manipulating. These approximate probabilistic joins have been studied in [15], dealing with joins between similar objects, and [16], focusing on nearest-neighbor joins.

Other works have studied the execution of probabilistic joins on streaming data [19, 20], focusing on the specific constraints of this context. Another situation where we often have uncertain data is that of spatial databases, where

we may not know with certainty the shape of the objects. [21] studied how to join this kind of data, and here the underlying (spatial) uncertainty model is different with respect to the one adopted in our work.

Finally, in this article we have studied methods to reduce the number of comparisons between certain or uncertain tuples. When we have numerical values like probabilities attached to our uncertain tuples we may also use these values (confidences) to exclude some pairs of tuples or to return first those pairs maximizing their joint confidence. This topic is of particular interest and complexity within the framework of continuous uncertainty models, and has been studied with regard to indexes for uncertain data and also with regard to the join operator in [2]. In this case we often speak of Threshold Probabilistic Join Queries. Preliminary main memory versions of some algorithms presented in this work were also sketched in [22] together with an experimental testing on two example data sets.

For a more extensive comparative discussion of some of these algorithms the interested reader may consult a recent survey on this topic [14].

3   Join algorithms

In this section we introduce the algorithms object of this paper, with a preliminary discussion of their main characteristics. Code listings are in pl/pgSQL[1] and should be understandable with some basic familiarity with SQL and the usage of cursors. For simplicity we will assume that joins are performed on a single attribute (called `val` in the code). All the approaches are exemplified on a self-join of the table illustrated in Figure 6.

3.1   Nested loop and base join

The nested loop approach is not different from the one presented in the introduction: it compares all uncertain tuples in the external relation with all uncertain tuples in the internal relation and the order of comparisons is the same of the nested loop example presented in the introduction — there are however more matches due to the additional values introduced in the uncertain version of our working example (Figure 6), as illustrated in Figure 7. The only specificity of the version for uncertain data is that an intersection operator (&&) is used to find matching values instead of a simple equality comparison (line 11):

```
1 CREATE FUNCTION nestedloopjoin(tab1, tab2)
2 RETURNS TABLE AS
3     OPEN cur1 FOR SELECT * FROM tab1;
4     OPEN cur2 FOR SELECT * FROM tab2;
5     LOOP
6         FETCH NEXT FROM cur1 into rec1;
7         EXIT WHEN rec1 IS NULL;
8         LOOP
```

---

[1] http://www.postgresql.org/docs/8.4/static/plpgsql.html

	ut1	ut2	ut3	ut4	ut5	ut6
ut1	**1**	**2**	3	4	5	6
ut2	**7**	**8**	9	10	11	12
ut3	13	14	**15**	16	17	**18**
ut4	19	20	21	**22**	23	**24**
ut5	25	26	27	28	**29**	30
ut6	31	32	**33**	**34**	35	**36**

Figure 7: Comparisons performed using a nested loop join on our uncertain working example. Numbers indicate the order of comparisons and boldface the matches found

```
9          FETCH NEXT FROM cur2 into rec2;
10         EXIT WHEN rec2 IS NULL;
11         IF rec1.val && rec2.val
12            RETURN NEXT concat(rec1, rec2);
13         END IF;
14         END LOOP;
15      MOVE ABSOLUTE 0 FROM cur2;
16      END LOOP;
17   RETURN;
```

In addition to our implementation of the uncertain nested loop algorithm we will also let the system execute directly the query:

SELECT * FROM tab1 JOIN tab2 ON tab1.val && tab2.val

to compare our results with the one obtained using the system relational query optimizer, i.e., without explicit support for uncertain data joins. This will be called base join in the following experiments and corresponds to the following code:

```
1 CREATE FUNCTION basejoin(tab1, tab2)
2 RETURNS TABLE AS
3    OPEN cur FOR SELECT * FROM tab1
4       JOIN tab2 ON tab1.val && tab2.val;
5    LOOP
6       FETCH NEXT FROM cur into res;
7       EXIT WHEN res IS NULL;
8       RETURN NEXT res;
9       END LOOP;
10   RETURN;
```

3.2   Sort join

Our extension of the traditional sort join algorithm considers each uncertain value as a range, from its lower alternative value to the upper one, and every

8

record is extended with two attributes (`lb` and `ub`) representing the bounds of the range. For example, the tuple ut1 in Figure 6 has ut1.lb=40 and ut1.ub=58. In this way, two tuples potentially match only if their ranges have a non-empty intersection. The code of the algorithm is the following:

```
1  CREATE FUNCTION sortjoin(tab1, tab2)
2  RETURNS TABLE AS
3     offset = 0;
4     OPEN cur1 FOR SELECT * FROM tab1
5         order by lb, ub;
6     OPEN cur2 FOR SELECT * FROM tab2
7         order by lb, ub;
8     LOOP
9        FETCH NEXT FROM cur1 into rec1;
10       EXIT WHEN rec1 IS NULL;
11       MOVE −offset FROM cur2;
12        offset = 0;
13       LOOP
14          FETCH NEXT FROM cur2 into rec2;
15           offset = offset + 1;
16          EXIT WHEN rec2 IS NULL;
17          EXIT WHEN rec1.ub < rec2.lb;
18          IF rec2.ub < rec1.lb AND offset = 1
19              offset = 0;
20          END IF;
21          IF rec2.ub >= rec1.lb AND
22              rec1.val && rec2.val
23             RETURN NEXT concat(rec1, rec2);
24          END IF;
25          END LOOP;
26       END LOOP;
27     RETURN;
```

First uncertain tables are sorted by the lower bounds of the uncertain attributes (`lb`, lines 4-5 and 6-7). Then we compare every record in the external relation (fetched at line 9) with its potential matches in the internal one (fetched at line 14). Let us consider our working example: we compare ut6 (having range [14,18]) against ut6, then with ut3 ([14,22]), ut4 ([18,19]) and ut5 ([**23**,28]). At this point thanks to the ordering we already know that ut6 will not match neither ut2 nor ut1, because the upper value in ut6 is 18 and this is less than 23. This condition is verified at line 17 and guarantees that no subsequent uncertain tuples in the (ordered) internal relation will match the uncertain tuple under consideration, as it happens with traditional sort joins over certain relations. This is then repeated for all tuples in the external input relation.

It is worth noticing that this approach reduces to a traditional sort join when the join attribute is not uncertain, or when it contains a single option. Intuitively the difference in performance between this approach and a traditional sort join will be small when the uncertainty does not change significantly the join attribute, that is when the range of values remains close to a single point, while

9

	ut6	ut3	ut4	ut5	ut2	ut1
ut6	**1**	**2**	**3**	4		
ut3	**5**	**6**	7	8		
ut4	**9**	10	**11**	12		
ut5	13	14	15	**16**	17	
ut2				18	**19**	**20**
ut1					**21**	**22**

Figure 8: Comparisons performed using an extended sort join. Numbers indicate the order and boldface the matches — notice that the input tuples have been sorted on the lower bounds of the join attribute

very large ranges may result in many unnecessary comparisons — this intuition will be verified in the experimental testing. In Figure 8 we have illustrated the order of comparisons and the found matches applying this extended algorithm to our working example.

3.3 Tuple-based join

This approach transforms the input uncertain relations into traditional relations and joins them using existing algorithms. Then a postprocessing phase is required to re-build an uncertain relation. The main advantage of this approach is that joins are performed on traditional relations, therefore they depend only on the values contained in the input relations but not on how uncertainty is distributed. The price to pay comes from pre- and post-processing phases, together with the size and cardinality of the intermediate relations. Therefore, intuitively this approach can be used when the specific distribution of uncertain values makes direct methods like the extended sort join algorithm too complex.

To use this approach we need each tuple to be identified uniquely by an attribute that we call xid and that in our working example corresponds to the primary key of the relation.

```
1 CREATE FUNCTION tuplejoin1(tab1, tab2)
2 RETURNS TABLE AS
3    OPEN cur FOR SELECT tab1.*, tab2.*
4       FROM flatjoin1(tab1, tab2) idxt
5       join tab1 ON idxt.xid1 = tab1.xid
6       join tab2 ON idxt.xid2 = tab2.xid;
7    LOOP
8       FETCH NEXT FROM cur into res;
9       EXIT WHEN res IS NULL;
10      RETURN NEXT res;
11      END LOOP;
12   RETURN;
```

The first part of the algorithm consists in the flattening of the input relations,

xid	Net Worth
ut3	14
ut6	14
ut6	16
ut3	16
ut4	18
ut6	18
ut4	19
ut3	22
ut5	23
ut5	25
ut5	28
ut2	37
ut2	40
ut1	40
ut2	42
ut2	47
ut1	50
ut1	53
ut1	58

Figure 9: Flattening of our working uncertain table used to compute `flatjoin1` (notice that the same table is used as both external and internal relation in this example, as we are computing a self-join)

xid	Net Worth	xid	Net Worth
ut3	14	ut3	14
ut3	14	ut6	14
ut6	14	ut3	14
ut6	14	ut6	14
ut6	16	ut6	16
ut6	16	ut3	16
ut3	16	ut6	16
ut3	16	ut3	16
ut4	18	ut4	18
ut4	18	ut6	18
ut6	18	ut4	18
ut6	18	ut6	18
ut4	19	ut4	19
ut3	22	ut3	22
ut5	23	ut5	23
ut5	25	ut5	25
ut5	28	ut5	28
ut2	37	ut2	37
ut2	40	ut2	40
ut2	40	ut1	40
ut1	40	ut2	40
ut1	40	ut1	40
ut2	42	ut2	42
ut2	47	ut2	47
ut1	50	ut1	50
ut1	53	ut1	53
ut1	58	ut1	58

xid	xid
ut3	ut3
ut3	ut6
ut6	ut3
ut6	ut6
ut4	ut4
ut4	ut6
ut6	ut4
ut6	ut6
ut5	ut5
ut2	ut2
ut2	ut1
ut1	ut2
ut1	ut1

Figure 10: Partial and final result (with all distinct pairs of ids of matching uncertain tuples) of the join between the flattening of the input tables, line 4 in the code

whose result is illustrated in Figure 9. Having all the alternative values of the uncertain attribute and the corresponding record identifiers we can thus find matching records using a traditional join. The result of this phase is illustrated in Figure 10, and this first part of the algorithm corresponds to the `flatjoin1` function at line 4 of the code. Finally, the original uncertain tuples are recovered by joining the obtained pairs of identifiers with the two input uncertain relations, to recollect all the other attributes.

In summary, this approach consists in building traditional relational tables, join them using traditional algorithms and get back all the uncertain tuples with matching values. This general approach can be implemented in different ways. First, it is interesting to notice that in the Trio system relations are already stored in this format, and the flattening phase would not be necessary. In addition, notice that in the flattening phase exemplified in Figure 9 we may decide not to project only on the **xid** and **Net Worth** attributes, but to keep also all the other attributes from the input tables. As a result, we will not need the subsequent joins (lines 5–6) to recollect the information projected out in the previous approach. However, in this way we have to work with larger intermediate tables replicating all the other attributes for each alternative value of the join attribute. In the following experiments we will evaluate also this variation, that will be indicated as *tuple join 2*.

```
1 CREATE FUNCTION tuplejoin2(tab1, tab2)
2 RETURNS TABLE AS
3     OPEN cur FOR SELECT distinct tab1.*, tab2.*
4         FROM flatjoin2(tab1, tab2)
5     LOOP
6         FETCH NEXT FROM cur into res;
7         EXIT WHEN res IS NULL;
8         RETURN NEXT res;
9         END LOOP;
10     RETURN;
```

Notice that we now perform only one join, indicated at line 4 (`flatjoin2`). This is computed on flattened tables like the one represented in Figure 11 with regard to our working example. Intuitively, this version of the tuple join approach has the advantage of having to compute only one join but in general it has to deal with larger intermediate relations keeping all the data from the input tables, so that it does not have to retrieve it later.

3.4   Index join

We conclude this section with a variation of a traditional index join. Here every record `ut` is augmented with an attribute (called `b` in the following code) representing a segment with extreme points ut.lb and ut.ub. In addition we use a spatial index on this attribute, so that when we want to look at potential matches we access the index and extract all records with an intersecting segment:

xid	Name	Surname	Net Worth	Age	val
ut1	William H.	Gates III	{53,50,40,58}	54	53
ut1	William H.	Gates III	{53,50,40,58}	54	50
ut1	William H.	Gates III	{53,50,40,58}	54	40
ut1	William H.	Gates III	{53,50,40,58}	54	58
ut2	Warren	Buffett	{47,40,37,42}	79	47
ut2	Warren	Buffett	{47,40,37,42}	79	40
ut2	Warren	Buffett	{47,40,37,42}	79	37
ut2	Warren	Buffett	{47,40,37,42}	79	42
ut3	Paul	Allen	{14,16,22}	57	14
ut3	Paul	Allen	{14,16,22}	57	16
ut3	Paul	Allen	{14,16,22}	57	22
ut4	Lawrence	Page	{18,19}	37	18
ut4	Lawrence	Page	{18,19}	37	19
ut5	Lawrence	Ellison	{28,23,25}	65	28
ut5	Lawrence	Ellison	{28,23,25}	65	23
ut5	Lawrence	Ellison	{28,23,25}	65	25
ut6	Michael	Dell	{14,16,18}	45	14
ut6	Michael	Dell	{14,16,18}	45	16
ut6	Michael	Dell	{14,16,18}	45	18

Figure 11: Flattening of our working example used to compute `flatjoin2`

```
 1 CREATE FUNCTION indexjoin(tab1, tab2)
 2 RETURNS TABLE AS
 3     OPEN cur FOR SELECT * FROM tab1
 4         JOIN tab2 ON tab1.b && tab2.b
 5         WHERE tab1.val && tab2.val;
 6     LOOP
 7         FETCH NEXT FROM cur into res;
 8         EXIT WHEN res IS NULL;
 9         RETURN NEXT res;
10     END LOOP;
11     RETURN;
```

The effect of this code is to execute a query very similar to the one of the base join algorithm (lines 3-5):

```
SELECT *
FROM tab1 JOIN tab2 ON tab1.b && tab2.b
WHERE tab1.val && tab2.val
```

The difference consists in the fact that the join is now performed on the indexed attributes. In this way the relational optimizer can use the index to identify potentially matching records, i.e., with a non empty intersection on the b attribute. All these are then checked against the WHERE predicate, selecting only those with a real match.

4   Experimental results

The algorithms presented in the previous section have been tested on several synthetic datasets and a real uncertain dataset obtained by extending the DBLP database with information about author institutions.

Synthetic uncertain tables contain a primary key, an uncertain join attribute and the additional attributes needed by the algorithms as previously explained (lower bound, upper bound - for sort join - and indexed uncertainty interval - for index join). An example of these data is presented in Table 1.

To easily understand the following experiments we can use this table to do some examples of the terminology adopted in the remaining of the section. We call *cardinality of an uncertain table* the number of uncertain records, four in the example (t0, t1, t2, t3). We call *cardinality of an uncertain tuple* the number of alternative values contained in it, three for each tuple in the example (224.480, 224.482 and 224.484 for the first tuple). In the following experiments it is also important to consider the width of the interval spanned by these alternative values ([224.480, 224.484] for the first tuple in the example): when values of one uncertain tuple are spread over a larger interval the number of records in the other input relation potentially matching it will typically increase. We will thus indicate with *spreading* the number of potentially matching records. Another tested parameter is the *percentage* of uncertain tuples, which is 100% in our example. The DBLP dataset with the uncertain affiliations of the authors is described in [13].

In this section we describe the experimental settings and the obtained results, and in the next section we provide an interpretation of these results. All the experiments have been conducted on a workstation with Linux Ubuntu 2.6.32.4, a 2GHz CPU, a Toshiba MK2555GS ATA disk and 3GB RAM. To be sure that the results are not influenced by the actual status of the buffers we restart the DBMS after every join and empty main memory using the Linux instruction `sync; echo 3 | sudo tee /proc/sys/vm/drop_caches`. The DBMS used in the experiment is PostgreSQL 8.4, and statistics on disk accesses are obtained using the `iostat` shell command. The sequence of tests, system restarting, memory cleaning and time and I/O statistics gathering has been automated using a Java program connecting to the database through JDBC.

Table 1: Structure of the synthetic tables used in the experiments with some example records

pk	lb	ub	b	val
t0	224.480	224.484	((224.480,0), (224.484,0))	{224.480, 224.482, 224.484 }
t1	954.463	954.467	((954.463,0), (954.467,0))	{954.463, 954.465, 954.467 }
t2	133.374	133.378	((133.374,0), (133.378,0))	{133.374,133.376,133.378 }
t3	12.000	12.004	((12.000,0), (12.004,0))	{12.000, 12.002, 12.004 }

4.1 Varying cardinality of input relations

In these experiments joins are performed between tables with a varying number of uncertain tuples and the following parameters:

- **Input table cardinality:** Varying from 1.000 (all algorithms) to 10.000.000 (only sort- and tuple-joins)

- **Cardinality of result:** Same as input relations

- **Cardinality of uncertain tuples:** 3

- **Percentage of uncertain tuples:** 100%

- **Spreading:** 1

The results of the experiments are illustrated in Figures 12, 13, 14.

In Figure 12 we show the execution time of all algorithms run on small relations. Here sort join and tuple join algorithms prove to be more efficient than the others, taking 0 to 1 second to perform every join task. Index-join is also very fast, while base join and nested loop join are slow even with these small datasets. Sort and Tuple joins are the only algorithms for which we could compute joins between tables with millions of uncertain tuples. In Figures 13 and 14 we have shown the execution time and the number of disk accesses for
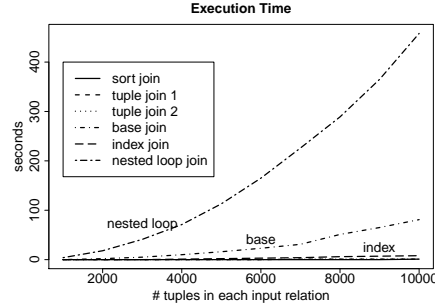
Figure 12: All algorithms tested on small input relations (curves regarding sort and tuple joins are all overlapping on the lower curve with y=0, as these algorithms take less than one seconds on all test datasets)
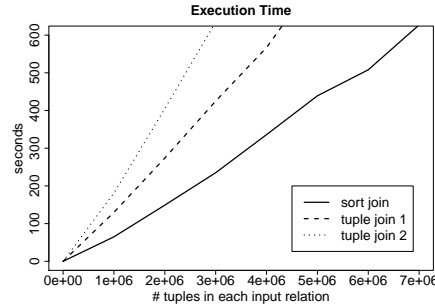

Figure 13: Tuple joins and Sort join tested on large relations (time)

these algorithms, varying the number of uncertain tuples in each input relation. If we compare the two plots we can see that time complexity is determined by the number of I/O operations — when this will be the case for the following experiments we will only include the plots with execution times.

4.2 Varying size of input relations, top results

When we perform joins using GUIs the typical behavior of modern database management systems consists in retrieving only the first few tuples of the result, usually with a default of about 100 tuples. Additional tuples are fetched into memory only if explicitly required by the user. As a consequence, it is interesting to evaluate how fast our algorithms are to start producing results and to compute a small subset of the resulting tuples. The results of these tests have been illustrated in Figures 15, 16.

First, notice that differently from the previous experiments all algorithms may deal with large tables with millions of uncertain tuples. Here, the algo-
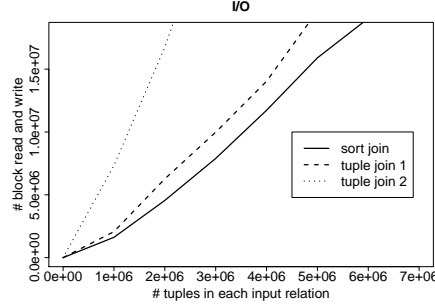
Figure 14: Tuple joins and Sort join tested on large relations (IO)
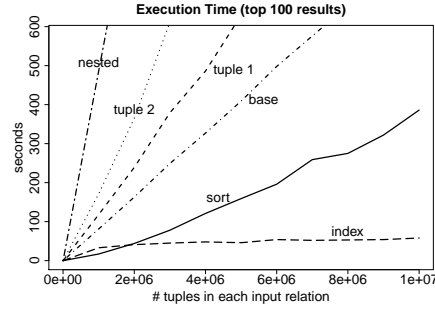
Figure 15: Time to retrieve the first 100 tuples in the result

rithms in order of increasing efficiency are: nested loop, tuple (second and first version), base join, sort join, and index join, which is by far the most efficient approach.

Second, the performance of tuple-join does not improve significantly with respect to the previous tests.

Finally, looking at Figure 16 we can notice that the relative efficiency of the algorithms with respect to execution time or number of I/O operations changes, showing that the execution time is not only determined by disk accesses.

4.3 Varying spreading of alternative values

In the previous tests the alternative values inside each tuple were very close to the same value, e.g., {10,12,14}. In this way every uncertain tuple in one relation potentially matched only one tuple in the other relation. In this test we increase the interval spanned by the alternative values to increase the number of potential matches, indicated by the *spreading* parameter.

- **Input table cardinality:** 1.000.000

18
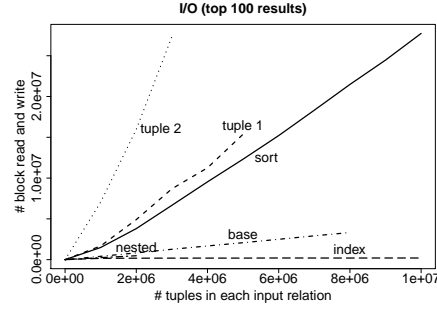
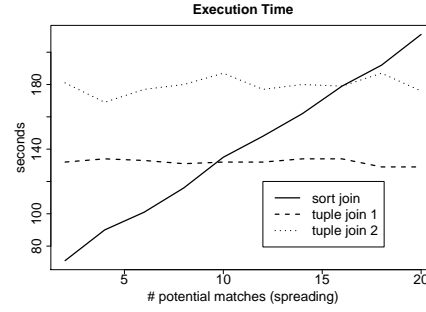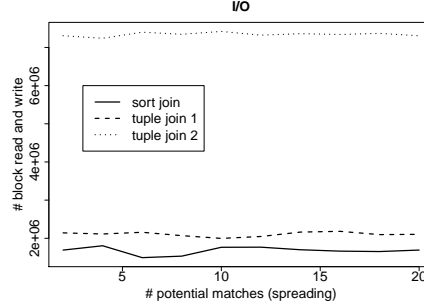Figure 16: Number of I/O operations to retrieve the first 100 tuples in the result

Figure 17: Tuple-joins and Sort-join tested on relations whose uncertain tuples contain values spanning an increasingly large interval (time)

- **Cardinality of result:** 1.000.000

- **Cardinality of uncertain tuples:** 3

- **Percentage of uncertain tuples:** 100%

- **Spreading:** Varying from 1 to 20

The results of the test are presented in Figures 17, 18.

Figure 17 shows an interesting behavior: the performance of both tuple join approaches is constant, while the execution time of the sort join depends on the variation of the *spreading* parameter.

Figure 18 shows the number of I/O operations needed by the three algorithms for the same experiment. In this case all curves are constant.

Figure 18: Tuple-joins and Sort-join tested on relations whose uncertain tuples contain values spanning an increasingly large interval (IO)

4.4 Varying tuple cardinality and percentage of uncertain records

In these experiments we vary the percentage of uncertain tuples in each input relation, and the cardinality of uncertain tuples, i.e., the number of alternative values contained in each tuple. The first set of tests corresponds to the following parameters:

- **Input table cardinality:** 1.000.000
- **Cardinality of result:** 1.000.000
- **Cardinality of uncertain tuples:** 3
- **Percentage of uncertain tuples:** Varying from 10% to 100%
- **Spreading:** 1

while the tests on the cardinality of the uncertain tuples have been performed with the following parameters:

- **Input table cardinality:** 1.000.000
- **Cardinality of result:** 1.000.000
- **Cardinality of uncertain tuples:** Varying from 2 to 10
- **Percentage of uncertain tuples:** 100%
- **Spreading:** 1

The results of the test are presented in Figures 19, 20, 21.

Figures 19 and 20 show an opposite behavior with regard to the previous experiments: now the complexity of tuple joins depends on the tested parameters, while the sort join has almost a constant behavior. Figure 21, showing the
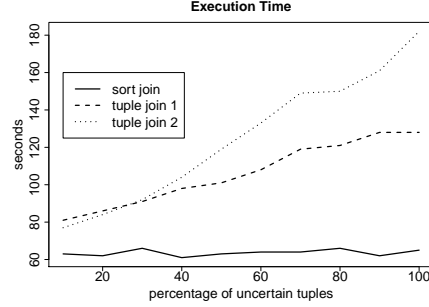
20

Figure 19: Tuple-joins and Sort-join tested on relations with varying percentage of uncertain tuples
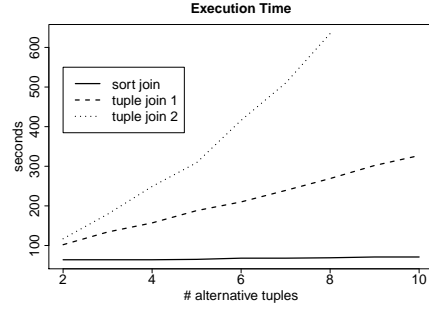

Figure 20: Tuple-joins and Sort-join tested on relations whose uncertain tuples contain an increasing number of alternative values (time)

number of I/O operations performed by the algorithms, allows us to appreciate a small dependency of the sort join on the number of alternative values without a perceivable effect on execution time with respect to the other algorithms.

4.5   Query behavior on DBLP dataset

We conclude the experimental analysis of our algorithms with a real dataset already used to evaluate indexes on uncertain data. While it is still difficult to find large and publicly available uncertain datasets because uncertain data management capabilities have not been incorporated into the mainstream relational database management systems, uncertain datasets are easily generated as a result of the integration of certain databases [23]. The data used in the following tests have been obtained by integrating DBLP author data with author affiliations, not present in the DBLP database and obtained via the Google API. This integration process is evidently uncertain, and up to ten possible alternative institutions have been recorded for each author. Additional details
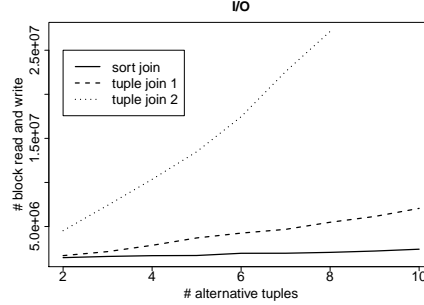
21

Figure 21: Tuple-joins and Sort-join tested on relations whose uncertain tuples contain an increasing number of alternative values (IO)

about these data can be found in [13].

In this section we evaluate the following query:

```
SELECT * FROM author A JOIN institution I
on A.institution = I.id
```

over two tables with respectively about 700.000 authors and about 6.000 institutions, using all the algorithms. The resulting table contains about 2.600.000 records.

In this case the test shows three orders of efficiency: the two versions of the tuple join approach, taking a few hundred seconds, index and base join, taking a few thousand seconds, and sort and nested loop join, taking a few ten thousand seconds. In addition, it is interesting to evaluate the time needed to compute not only the whole result, but also smaller parts of the output.

In Figure 22 we have represented for each join its behavior in time, i.e., the number of records computed after t seconds. Therefore, the slope of the curve represents the speed of producing results at time t. This will be clear by looking at the plots: both tuple join approaches do not output any result for a while (about 230 and 150 seconds respectively), then they start producing tuples very quickly until all the join has been computed. All the other approaches start computing tuples at once, with an almost linear behavior for index, sort and nested loop join and a decreasing speed for the base join, which is very fast at the beginning and slows down in time.

5 Interpretation of the experimental results

In this section we provide our interpretations of the experimental results presented in Section 4.

The cardinality of the input relations obviously affects the performance of all algorithms. Among these, base join and nested-loop join are in general the
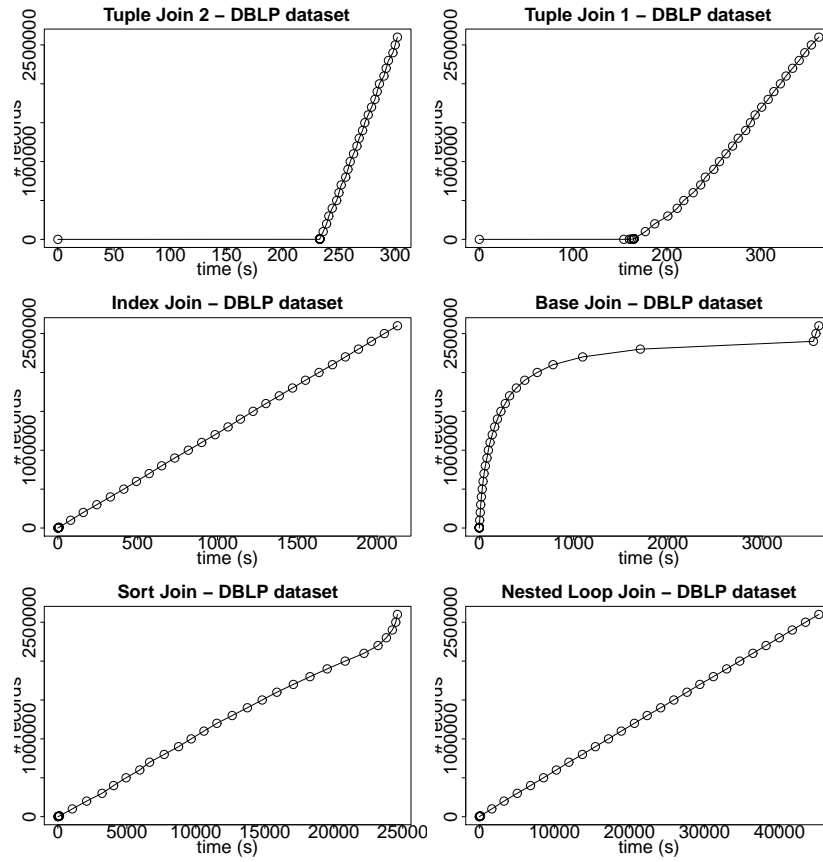
Figure 22: Behavior of all join algorithms on the DBLP query.

slower methods (Figure 12) because they do not provide any specific support for uncertain data (we remind the reader that the base join is the method chosen by the Postgres Query Optimizer). The index join is more efficient but still requires many disk accesses to retrieve the matching tuples. Figure 13 shows that when alternative values are distributed in a small interval (where the meaning of *small* is not absolute but depends on the number of matches with records in the other relation, as tested in following experiments) both sort join and tuple joins can deal with very large relations.

When it comes to compute only a few results, however, the index-based approach is the best one, because in our experimental setting it just requires one disk access for each tuple in the result, in addition to an index search (Figure 15). It is interesting to notice that the only algorithms that do not significantly improve their performance in this case are the tuple joins, because before starting to output the results they have to compute the complete join (tuple join 1) and perform a `distinct` operation (tuple join 2). Therefore, when they start producing the output they have already done most of their work.

Figures 14 and 16 show that the execution time of these tests is determined by the number of I/O operations for all approaches except base and nested loop join, where the execution time depends on the number of main memory operations.

Figures 17 and 18 show that the execution time of tuple-based approaches does not depend on the distribution of the alternative values inside each uncertain attribute. In fact, these methods split the alternative values into traditional relational tuples and thus perform a join between relations from which uncertainty has been temporarily removed. The sort join approach works directly on uncertain tuples, and is thus influenced by the distribution of uncertainty. In particular, when the interval inside which the alternative values are distributed increases this method has to perform a lot of potentially unnecessary comparisons, and its performance decreases — in the worst case it behaves like a nested loop (or nested block loop) join, as it happens with traditional sort joins in traditional relational systems when many records have the same value on the join attribute in both input relations.

On the contrary, Figures 19 and 20 highlight how the performance of the sort join depends on the cardinality of the uncertain relations, i.e., the number of uncertain tuples, while tuple based approaches depend on the number of alternative values. As an example, if an input relation contains 1.000.000 uncertain tuples with 10 alternative values each, the sort join will manipulate 1.000.000 uncertain tuples while the tuple join algorithm will operate on a traditional relation with 10.000.000 tuples. Therefore, when we add more alternative values or increase the percentage of tuples with multiple alternative values without spreading these values into a large interval, the performance of the sort join will be almost constant while tuple joins will be significantly affected.

Figure 21 shows that also the sort join method has a small dependency on the cardinality of the uncertain tuples: adding alternative values to an uncertain tuple increases the size of the input relations and thus requires additional I/O operations, as highlighted by the slight slope of the sort join curve.

Finally, Figure 22 provides some evidence regarding the behavior of these methods in time with respect to a single join, supporting and motivating some of the behaviors observed in the previous experiments. From the figure it is clear how both tuple join approaches have a preprocessing phase followed by a fast generation of the results. These tests also highlight how the sort join approach should not be used when alternative values are not distributed around specific values: in this case every author was associated to different institutions, and there is no reason why two institutions associated to the same author should be close to each other in the `Institution` table. As a consequence the behavior of this method is similar to the one of the nested loop join, though faster[2].

6   Discussion and conclusion

Besides other specific results, the experimental analysis described in Section 4 highlights two main points. The first, as expected, is that joining uncertain relations with the methods introduced in the first part of this paper is in general more complex than joining certain relations. Depending on the specific algorithm, the increment in size of the input relations due to the additional values to store, the distribution of these values or the need to recompose uncertain tuples increase the computation time. The second point is that different algorithms are influenced by different factors. Therefore it is possible to choose an efficient algorithm depending on the input data.

More in detail, we have shown that tuple-based approaches have a time complexity which is independent of the distribution of uncertain values in the data, and corresponds to the execution of three traditional joins (or one join with duplicate removal) on larger certain relations. However, their performance depends on the number of alternative values contained inside each uncertain tuple. When these values are close to each other the sort-based approach is almost as efficient as a traditional sort-join. In addition the sort-based approach improves significantly when we require only the first few tuples from the result, as it usually happens in modern database system GUIs where additional tuples are fetched into memory only if explicitly required by the user. However, in this case using an index is the most efficient approach. On all our experiments the algorithm chosen by the underlying relational system, which is not aware of the uncertainty, is outperformed by at least one of the uncertainty-aware approaches, and using the most appropriate algorithm we have been able to efficiently join both synthetic and real data and tables containing millions of uncertain tuples.

---

[2]The increase in speed at the end of the execution depends on the fact that only a few records remain to be tested at the end, therefore the number of main memory comparisons decreases

References

[1] Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Shubha U. Nabar, Tomoe Sugihara, and Jennifer Widom. Trio: A system for data, uncertainty, and lineage. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 1151–1154. ACM, 2006.

[2] Parag Agrawal and Jennifer Widom. Confidence-aware join algorithms. In *25th International Conference on Data Engineering (ICDE 2009)*. Stanford InfoLab, 2009.

[3] L. Antova, T. Jansen, C. Koch, and D. Olteanu. Fast and simple relational processing of uncertain data. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 983 –992, 2008.

[4] Daniel Barbara, Hector Garcia-Molina, and Daryl Porter. The management of probabilistic data. *IEEE Transactions on Knowledge and Data Engineering*, 4(5):487–501, 1992.

[5] Patrick Bosc and Henri Prade. An introduction to the fuzzy set and possibility theory-based treatment of flexible queries and uncertain or imprecise databases. In *Uncertainty Management in Information Systems*, pages 285–324. 1996.

[6] Jihad Boulos, Nilesh N. Dalvi, Bhushan Mandhani, Shobhit Mathur, Christopher Ré, and Dan Suciu. Mystiq: a system for finding more answers by using probabilities. In *SIGMOD Conference*, pages 891–893, 2005.

[7] Reynold Cheng, Sarvjeet Singh, and Sunil Prabhakar. U-dbms: A database system for managing constantly-evolving data. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 1271–1274. ACM, 2005.

[8] Nilesh N. Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. In *Proceedings of the International Conference on Very Large Data Bases*, 2004.

[9] Debabrata Dey and Sumit Sarkar. A probabilistic relational model and algebra. *ACM Transactions on Database Systems*, 21(3):339–369, 1996.

[10] Norbert Fuhr and Thomas Rölleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Transactions on Information Systems*, 15(1):32–66, 1997.

[11] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[12] Jiewen Huang, Lyublena Antova, Christoph Koch, and Dan Olteanu. Maybms: a probabilistic database management system. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *SIGMOD Conference*, pages 1071–1074. ACM, 2009.

[13] Hideaki Kimura, Samuel Madden, and Stanley B. Zdonik. Upi: a primary index for uncertain databases. *Proc. VLDB Endow.*, 3:630–637, September 2010.

[14] Hans-Peter Kriegel, Thomas Bernecker, Matthias Renz, and Andreas Zuefle. Probabilistic join queries in uncertain databases. In Ahmed K. Elmagarmid and Charu C. Aggarwal, editors, *Managing and Mining Uncertain Data*, volume 35 of *Advances in Database Systems*, pages 257–298. Springer US, 2009.

[15] Hans-Peter Kriegel, Peter Kunath, Martin Pfeifle, and Matthias Renz. Probabilistic similarity join on uncertain data. In Mong-Li Lee, Kian-Lee Tan, and Vilas Wuwongse, editors, *DASFAA*, volume 3882 of *Lecture Notes in Computer Science*, pages 295–309. Springer, 2006.

[16] Hans-Peter Kriegel, Peter Kunath, and Matthias Renz. Probabilistic nearest-neighbor query on uncertain objects. In Kotagiri Ramamohanarao, P. Radha Krishna, Mukesh K. Mohania, and Ekawit Nantajeewarawat, editors, *DASFAA*, volume 4443 of *Lecture Notes in Computer Science*, pages 337–348. Springer, 2007.

[17] Laks V. S. Lakshmanan, Nicola Leone, Robert Ross, and V. S. Subrahmanian. ProbView: a flexible probabilistic database system. *ACM Transactions on Database Systems*, 22(3):419–469, 1997.

[18] Suk Kyoon Lee. An extended relational database model for uncertain and imprecise information. In Li-Yan Yuan, editor, *Proceedings of the International Conference on Very Large Data Bases*, 1992.

[19] Xiang Lian and Lei Chen. Efficient join processing on uncertain data streams. In *Proceeding of the 18th ACM conference on Information and knowledge management*, CIKM '09, pages 857–866, New York, NY, USA, 2009. ACM.

[20] Xiang Lian and Lei Chen. Similarity join processing on uncertain data streams. *IEEE Transactions on Knowledge and Data Engineering*, 99(PrePrints), 2010.

[21] Vebjorn Ljosa and Ambuj K. Singh. Top-k spatial joins of probabilistic objects. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 566–575, Washington, DC, USA, 2008. IEEE Computer Society.

[22] Matteo Magnani and Danilo Montesi. Optimization of queries over interval probabilistic data. In *Scalable Uncertainty Management, 2nd International Conference*, volume 5291 of *Lecture Notes in Computer Science*, pages 298–311. Springer, Berlin, 2008. isbn: 978-3-540-87992-3.

[23] Matteo Magnani and Danilo Montesi. A survey on uncertainty management in data integration. *ACM Journal of Data and Information Quality*, 2(1), 2010. ACM, New York, NY, USA, issn: 1936-1955.

[24] Michael Pittarelli. An algebra for probabilistic databases. *IEEE Transactions on Knowledge and Data Engineering*, 6(2):293–303, 1994.

[25] Christopher Re, Nilesh N. Dalvi, and Dan Suciu. Efficient top-k query evaluation on probabilistic data. In *Proceedings of the 23rd International Conference on Data Engineering*, pages 886–895. IEEE, 2007.

[26] Anish D. Sarma, Xin Dong, and Alon Y. Halevy. Bootstrapping pay-as-you-go data integration systems. In *SIGMOD Conference*, pages 861–874, 2008.

[27] Anish Das Sarma, Omar Benjelloun, Alon Y. Halevy, and Jennifer Widom. Working models for uncertain data. In *Proceedings of the 22nd International Conference on Data Engineering*, page 7. IEEE Computer Society, 2006.

[28] Jennifer Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, pages 262–276, 2005.