A Treatment of Correlated Attribute Uncertainty in Array Database Systems

David Grabiner
Department of Computer Science
Brown University
grabiner@cs.brown.edu

ABSTRACT

The purpose of this document is to summarize ongoing research into handling uncertainty in database systems, and present my contributions to that research. Uncertainty presents a particular problem for databases because the relational model leaves no room for ambiguity in database fields. Traditional database operators such as selection and join operate on constant values as opposed to probability distributions. In order for a database system to correctly handle this uncertainty, values must be sampled from the distributions and the operators must operate on those sampled values. If this process is not done intelligently, one join operation on deterministic data can turn into one thousand join operations on probabilistic data. Additionally, if the uncertainty of different attributes is correlated, the entire joint distribution of all the correlated attributes must be sampled at once, as opposed to sampling from each distribution individually. In this paper, I describe the query semantics for handling correlated uncertainty, jointly developed with Tingjian Ge. I then describe the S-Join algorithm that efficiently performs a join when the join attribute is a continuous distribution, also codeveloped with Tingjian Ge. Finally, I present experimental results that I obtained from an implementation of the algorithms that I constructed. This work is discussed in much greater detail in a paper of the same name, authored by Tingjian Ge, David Grabiner, and Stan Zdonik [1].

1. INTRODUCTION

Uncertainty arises in multiple applications, especially in the scientific world. Any sensor that monitors a continuous value, such as temperature or position, is likely to have uncertainty in that value. A database system needs to account for that uncertainty in answering queries. For example, let's imagine a temperature sensor whose readings have a known variance of one degree, and the results of the sensor are stored in a database. Imagine this database contains a tuple T_1 which indicates that on Sunday the temperature was 79 degrees. Now someone queries the database for all rows with temperature greater or equal to 80. Should R_1 be included in the result set? In a traditional database, the answer would be no. However, there is a 16% chance that the temperature represented by T_1 was actually over 80 degrees because of the uncertainty in the sensor. Databases for scientific applications need to be able to handle this uncertainty and it should manifest itself in query results.

The uncertainty problem is further complicated when the uncertainty between different values is correlated. Imagine that my database can handle the situation above and when queried for temperatures above 80 will tell me that the probability that T_1 is in the result set is 16%. Now imagine that the database contains another tuple T_2 which indicates that on Monday the temperature was also 79 degrees. Now the database is queried for two consecutive days in which the temperature was above 80 degrees. Since T_1 and T_2 both have a probability of .16 of representing a temperature over 80 degrees, the probability of the result set containing the combination (T_1,T_2) is 0.16*0.16 = 0.0256, assuming the measurements are independent. However, the measurements are not independent; they come from the same sensor. What if the sensor is precise but inaccurate and all measurements have the same error due to the sensor itself? In that case, the database needs to handle this correlated uncertainty and report that combination (T_1,T_2) should actually be in the result set with probability 0.16 rather than probability 0.0256.

The rest of the paper is organized as follows. Section 2 describes the query semantics developed to handle correlated uncertainty. Section 3 presents a join algorithm designed to efficiently handle uncertain join attributes. Section 4 summarizes experiments performed on an implementation of the join algorithm, and Section 5 concludes the paper.

2. QUERY SEMANTICS

It is not immediately obvious what the semantics of queries in the presence of correlated uncertainty should be. Not only must the semantics incorporate probabilistic membership in the result set, they must also incorporate uncertainty in result values. As an example, consider the query from the introduction of all rows with temperatures greater than 80. Tuple T_1 has a temperature of 79 with a variance of 1, and should thus be in the result set with probability 0.16. However, it does not make sense for a row in the result set to have a mean temperature value of 79. Since the results should only include values greater than 80, all rows in the result set must have means above 80. To rectify this problem, we must treat the distribution on the temperature attribute of the tuple in the result set as a conditional probability distribution. The value in the temperature field should be the mean value of the temperature given that it is above 80 degrees, even though its unconditioned mean was 79. In this case, the result set should contain the result row RT_1, derived from T_1, with probability 0.16. The value of the temperature field in RT_1 must be 80.5, which is the conditioned mean of T_1 having a temperature above 80.

Thus, two uncertain elements must be well-defined in the query semantics: The probability of membership in the result set and the value of uncertain fields given membership in the result set.

Tingjian Ge and I have jointly developed two formalizations of what the query semantics should be. The first is integral-based, and the second, more practical formalization is sampling-based.

2.1 Integral-Based Semantics

For a full description of the integral-based semantics, see section 2.1 of [1]. The intuition behind these semantics is that the probability density function of an uncertain attribute X covers some area A. Given a query q, there is a subregion A^+ of A in which q(X) produces a tuple t in the result set and a complimentary region in which no result tuple is produced. We would like to know the total mass of the probability density function that falls within A^+. To find this mass, we integrate over the region. In our temperature example query, the probably density function of T_1 is a normal distribution with mean 79 and variance 1 where the A^+ region is everything greater than 80. Thus $Pr(RT_1)$ is the integral from 80 to positive infinity of that distribution.

When multiple input tuples are correlated, their joint probability density function must be integrated over. The query q can be thought of as a function that takes in a series of values and returns the result tuple t when those values satisfy q's predicate. The integral that calculates the probability of the result tuple t, Pr(t), is presented below.

$$\Pr(t) = \int_{\substack{x_1, x_2, \ldots, x_n \\ q(x_1, \ldots, x_n) \to t}} f(X_1, X_2, \ldots, X_n) dA$$

In this integral, $f(X_1, X_2, \ldots, X_n)$ is the joint probability density function over correlated attributes X_1 through X_n. When the input tuples are correlated it will often be the case that result tuples are correlated as well. In this case, the query q maps input values to a set of result tuples t_1 through t_k. The probability of that set of tuples being in the result set is given by:

$$\Pr(t_1, t_2, \ldots, t_k) = \int_{\substack{x_1, x_2, \ldots, x_n \\ q(x_1, \ldots, x_n) \to t_1, \ldots, t_k}} f(X_1, X_2, \ldots, X_n) dA$$

Now consider the case of uncertain values in result tuples. If a result tuple t has an uncertain attribution y, then the probability densifty function over y, f(y), can be calculated using a similar procedure. In this case, the function q returns a value for y given a set of input tuple values. The function f(y) can be calculated as follows:

$$f(y) = \int_{\substack{x_1, x_2, \ldots, x_n \\ q(x_1, \ldots, x_n) \to y}} f(X_1, X_2, \ldots, X_n) dA$$

2.2 Sampling-Based Semantics

The integral-based semantics is correct but not very practical. To realize an actual database system, integration must be replaced by Monte Carlo sampling. As the number of samples N approaches infinity, the result Pr(t) will approach the true Pr(t) that would be obtained from solving the integral. In this framework, query evaluation proceeds as follows: For every uncertain value in an input tuple, a random sample is drawn from its distribution. The query then operates on these values as if they were deterministic values and produces a result set. This process is then repeated N times. Going back to our temperature example, this is akin to drawing N samples from the normal distributions represented by tuples T_1 and T_2 and evaluating the query every time a pair of samples is drawn.

Because result tuples will likely be correlated, the results of query evaluation must reflect this correlation. To achieve this end, each result tuple is augmented with a bit vector which delineates which sampling rounds yielded that tuple. For example, if N = 5 and tuple RT_1 was produced in the first and fourth round while tuple RT_2 was produced in the first and third round, RT_1 will be augmented with the bit string 10010 and RT_2 will be augmented with 10100. Now, $Pr(RT_1)$ can be obtained by taking the cardinality of the RT_1 bit string and dividing by N. In this case $Pr(RT_1) = 0.4$ and $Pr(RT_1) = 0.4$. $Pr(RT_1, RT_2)$ can be obtained by performing the logical AND of the two bit strings and dividing the cardinality of the result by N. In this case, $Pr(RT_1, RT_1) = 0.2$ as opposed to 0.16 which would be the result of multiplying the two marginal probabilities together. Performing query evaluation through Monte Carlo sampling thus provides a powerful framework that can handle and express correlated uncertainty in tuples.

3. SAMPLING-BASED JOIN (S-JOIN)

3.1 The S-Join Algorithm

One of the drawbacks to query evaluation through sampling is that it is a very time-intensive process. If 1000 rounds of sampling are used, then every query must be evaluated 1000 times. Traditionally, the join operation has been one of the larger bottlenecks in query evaluation. Performing 1000 joins every time a join query is posed would be a painful process. To alleviate this pain, we have developed a join algorithm that takes advantage of the structure of the sampling problem to provide a significant speedup over running a standard join algorithm over and over. We call our algorithm the S-Join algorithm.

The S-Join algorithm performs a join where the join attribute is uncertain. Returning to the temperature example, what if we are interested in finding two days that have the same temperature? To realize this query in SQL, a self-join must be performed where the join attribute is the uncertain temperature attribute:

```
SELECT temp1.day, temp2.day
FROM temperature as temp1, temperature as temp2
WHERE ABS(temp1.value – temp2.value) < ε AND
        temp1.day != temp2.day
```

In this query, value of the temperature table is an uncertain attribute. Assuming that the uncertain attribute is sampled from 1000 times, naively evaluating this relatively simple query will result in performing 1000 join operations on a table that could be very large.

The intuition behind the S-Join algorithm is that given a series of uncertain values, the order of samples drawn from their distributions should be similar to their expected values. Thus if the uncertain values are sorted by their expected values and then samples are drawn in that order, the samples themselves should be pseudo-sorted. We refer to the order of the tuples when sorted by expected value as the expected order. For example, assume the temperature table has the following tuples in expected order (presorted by expected value), each with a variance of 1:

ID	Day	Expected Value

1	Monday	79.0
3	Wednesday	82.0
2	Tuesday	85.0

Now imagine the following samples are drawn from the three distributions:

ID	Day	Sampled Value
1	Monday	79.5
3	Wednesday	82.4
2	Tuesday	84.9

Notice that the sampled values, drawn in the same order as the expected values, are in correct sorted order. This will not always be the case but it will be most of the time.

We can thus sort the expected values once using a fast sort algorithm, such as quicksort, draw samples, and use an insertion sort to correct the order of the samples. While an insertion sort takes time N^2 to sort a random series of values, if it can be guaranteed that any one value is only k spots away from its correct location, where k << N, the time decreases to N*k. Because sampling should not drastically alter the order of values from the order of their expected values, we feel confident in this guarantee for some small k.

The algorithm thus proceeds as follows. Perform an external sort on the input tuples according to their expected value, putting them in expected order. We assume that sampling will not change the order of a tuple more than k pages. Let's take k = 2. Now allocate 3 pages in memory for each side of the join for some number of rounds R. R is selected so that 2*3*R pages are able to fit in memory at once. Read in the first three pages of the externally sorted tuples. As each tuple is read in, draw R samples from the distribution on its join attribute. When a sampled value is drawn for round r, insert that value into r's pages using an insertion sort, maintaining sorted order in each round.

In the example above, the process entails reading in the Monday tuple from the table in expected order, drawing R samples from that tuple and populating R rounds with those R samples. Then the Wednesday tuple is read in and sampled R times. Those R samples are inserted into their respective buffers, which each already contain a sample from the Monday tuple, using an insertion sort. The process is then repeated for the Tuesday tuple.

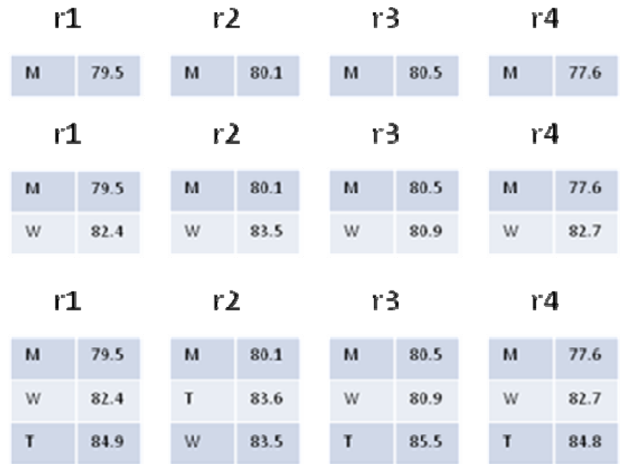

Figure 1. Sampling and sorting the first three example tuples over four rounds

After this sampling process is complete, we can perform a merge-join on the first page of every round. Because of our assumption that tuples can only deviate k=2 pages from expected order, the first of these pages will be in final sorted order while the next two will be temporarily sorted. This is because only values sampled from tuples in the first three pages can end up in the first page. Tuples whose expected order puts them in the fourth page cannot be in the first page when ordered by sample values, so the first page must be in final sorted after reading and sampling the first three pages. The second and third pages are temporarily sorted but are not in final sorted order because tuples from the fourth and fifth page, which have not yet been sampled from, may need to be interspersed within the second and third page. Now the merge process is performed over the first page in every round. After enough memory has been freed up through consuming pages while merging, a new page is read in and sampled from and another insertion sort is performed with the samples from this new page. As a result, another page is now in final sorted order and can be merged in each round. This process continues until sampling and merging is complete.

Let's assume we use 0.5 for epsilon in the original temperature query:

SELECT temp1.day, temp2.day
FROM temperature as temp1, temperature as temp2
WHERE ABS(temp1.value – temp2.value) < 0.5 AND
temp1.day != temp2.day

If the three tuples (Monday, Tuesday, and Wednesday) are the only tuples in the temperature table and only four rounds of samples are drawn as shown in Figure 1, the results of the join are:

temp1.day	temp2.day	bitVector
Tuesday	Wednesday	0100
Monday	Wednesday	0010

Analyzing the two result bit vectors as described in Section 2.2 yields Pr(T,W) = 0.25 and Pr(M,W) = 0.25.

The S-Join algorithm provides a significant speedup over naively performing all joins independently because only one external sort must be performed (on the expected values of the original tuples). The only subsequent sorts are in-memory insertion sorts on tuples that are already nearly sorted. Additionally, by performing multiple rounds together, unnecessary disk accesses are eliminated. If 200 rounds can be performed at once out of a total of 1000 rounds, the externally sorted tuples need to be read from disk only 5 times as opposed to 1000 times.

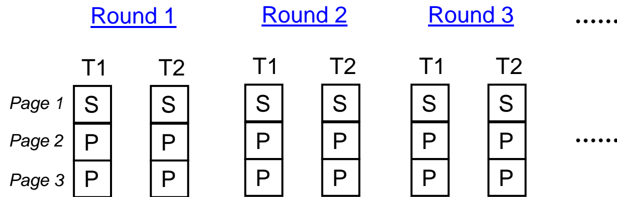

Figure 2. Illustrating sampling-based join algorithm. Multiple rounds are carried out in parallel. Each round has 3 pages in buffer for each side of the join (T1 and T2). Pages for round 1 are all sorted. For other rounds, initially the 3 pages of a relation are pseudo-sorted. As insertion sort is performed, the 1st page is sorted while the other two are still not in their final sorted form, since tuples from following pages may need to be mixed in.

(1) Externally sort the input tuples according to their expected value

(2) Allocate 3 pages in buffer for each side of the JOIN for each round (for a number of rounds that can fit in the buffer space).

(3) For each round:

(4) Obtain fresh samples for the 3 pages in buffer (for each side of JOIN) and arrange them in *pseudo-sorted* order determined by (1).

(5) Perform an *insertion sort* on the pseudo-sorted 3 pages (for each side of JOIN). After doing this, according to our assumption, the first of the 3 pages will be *exactly* sorted, and the other two pages will remain pseudo-sorted.

(6) End

(7) For each round, repeat the following until JOIN finishes (all rounds *in parallel*, e.g., in a *lockstep* or in *round-robin* fashion),

(8) Do the "merge" step of the JOIN on the *exactly* sorted pages.

(9) Replaced consumed pages with new ones

(10) Obtain a set of fresh samples for the new pages and perform *insertion sort* to adjust the order. This converts another page (the oldest among the three) from *pseudo-sorted* to *sorted*.

(11) End

Figure 3. The S-Join Algorithm

4. EMPIRICAL STUDY

I have implemented the S-Join algorithm in Java. The implementation takes in two representations of database tables and performs a join on them, using a configurable number of rounds. The number of rounds to perform at once is also configurable, as is the maximum displacement of a tuple after sampling. I also implemented two naïve join algorithms. The first algorithm performs the join on every round independently. The second algorithm externally sorts the expected values as in the S-Join algorithm and then resorts the pseudo-sorted samples with an insertion sort. However, sampling and merging of rounds is done in series so the input tuples must be read from disk every round. The experiments were run on a Debian Linux workstation with an AMD Athlon-64 2 Ghz processor, 512 MB memory and a Samsung HD160JJ disk. Additional results can be found in [1].

In this section, I use a real world application, multi-sensor tracking [2], to empirically study the following issues:

- Using our proposed correlation model and sampling algorithms, how does the query result compare with a model that assumes independence on the attribute values of different tuples?
- How is the performance of our S-Join algorithm? How does it compare with a naïve algorithm that performs a multi-round join operation?

4.1 Problem and Setup

A common vexing problem in multi-sensor tracking is to devise a mapping between the tracks of one sensor and the tracks of another sensor, assuming both sensors are tracking the same objects [2]. Once a mapping has been verified, tracks from different sensors on the same object can be fused together to form a single object track. This problem, known as the track-to-track correlation problem, is well-suited for our study because of the nature of errors in sensor tracks. Track errors are often the result of bias in the sensor that maintains the track. Thus, it can be expected that errors in tracks that originate from the same sensor will be correlated in some fashion. I generate synthetic datasets for the tracks of different sensors and model correlation of errors for tracks originating from the same sensor with a simple graphical structure (cliques of size two). Each track itself also has a random error independent of other tracks.

4.2 Correctness

As an example, Figure 4 shows the positions in *X* and *Y* of six objects, each being tracked by two sensors. Tracks 1-6 belong to sensor 1 and are illustrated with dots in solid-lined circles. Tracks A-F belong to sensor 2 and are illustrated with dots in dashed-lined circles. The dots in the center depict the reported position of each track, the *original* value in our model. The circles around those dots are drawn one standard deviation away from the center and serve to demarcate the *correction* value in our model. The actual position of a track is the original plus the correction. The errors of the tracks from the same sensor are correlated due to the common sensor error. Individual tracks also have a random error.

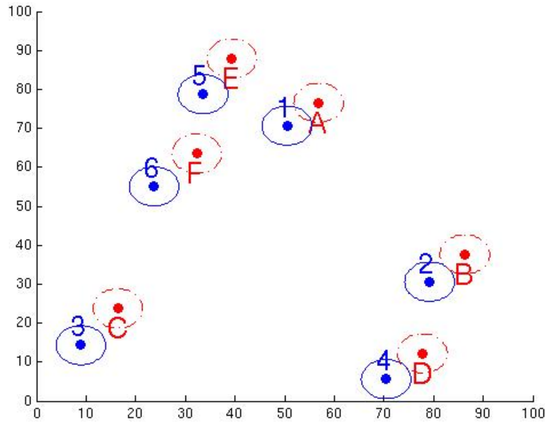

Figure 4. Example track from two sensors

Let us define T_1 as the set of all tracks belonging to sensor 1 and T_2 as the set of all tracks belonging to sensor 2. A mapping M is defined as a set of pairs $(t_1 \in T_1, t_2 \in T_2)$ such that each t_1 and t_2 appear exactly once. The probability of a mapping M being valid is the probability that *distance* $(t_1, t_2) < \varepsilon$ for all pairs (t_1, t_2) in M. Under an (erroneous) independence assumption,

$$\Pr(M) = \prod \Pr(t_1 \approx t_2) \cdot$$

The query necessary to create this mapping is:

SELECT sensor1.trackID, sensor2.trackID
FROM sensor1, sensor2
WHERE SQRT(POWER(sensor1.x – sensor2.x, 2) +
POWER(sensor1.y – sensor2.y, 2)) < ε

In this case, the result of note is not any individual result tuple but a joint distribution on many result tuples. In the semantics defined in Section 2.2, each result tuple will be accompanied by a k-bit array where k is the number of rounds of sampling. The value of bit i for tuple t will be 1 if that tuple is a member of the result set in the i'th round of sampling. Thus I can compute the probability of any mapping M by doing a bit-wise *AND* of all the bit arrays of result tuples in M, counting the number of *1*s, and dividing by k.

To the naked eye, it looks obvious that track *1* should be paired with track *A*, track *2* with track *B*, and so on. However, our experiments show that if independence between track errors is assumed, the probability of this mapping being valid quickly approaches zero as the number of tracks increases.

I drew 10,000 rounds of samples for each track using the correlated model and the uncorrelated model. The uncorrelated model has the same variance as the marginal distribution of the correlated model. I tallied up the number of rounds that yielded sampled track states such that *distance* $(t_1, t_2) < \varepsilon = 5$ for all pairs (t_1, t_2) in our hypothesis M under each model. I then divided that tally by the number of rounds to yield *Pr(M)*. Figure 5 shows the average *Pr(M)* over 10 distinct trials under both models as the number of tracks increases. Even though the error circles seem to overlap quite a bit, the direction of each error is unknown, and the chance that solid-circled tracks have negative error in X and Y while dashed-circle tracks have positive error in X and Y so that the actual track positions are the same for all pairs is quite small (approximately 0.1 for each track). Thus, if the correlation between errors of a given sensor's tracks is ignored, the probability that all six pairs of tracks are caused by the same six objects is 0.1^6. In contrast, if most of the track error is attributed to correlated sensor error, the effects of adding more tracks on *Pr(M)* is mitigated. It is clear from the figure that if correlations in error are ignored, simple queries yield highly erroneous results even when the number of correlated tuples is fairly small.

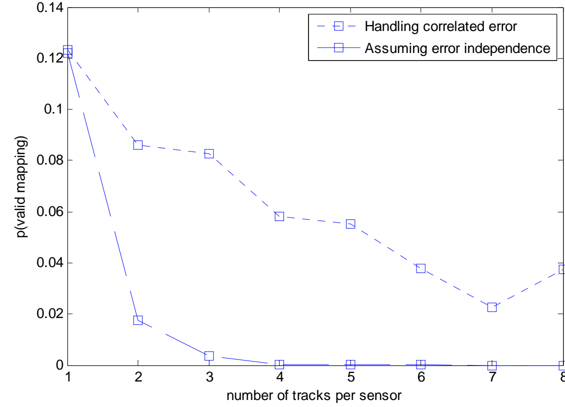

Figure 5. Probability of a valid mapping of tracks in Figure 3 under correlated and uncorrelated model.

4.3 Performance

In order to process queries of the form:

SELECT A.ID, B.ID
FROM A,B
WHERE ABS(A.value – B.value) < ε

as in the previous example, it becomes necessary to process JOINs efficiently over multiple rounds of sampled data. Imagine hundreds of sensors each tracking thousands of objects. In this case, efficiently processing the JOIN is of the utmost importance. I implemented our JOIN algorithm presented in Section 4 and tested it against two standard naïve JOIN algorithms. The first naïve algorithm (1) performs a sort-merge join on each round of samples independently. It reads in all the data, samples from the data, and performs an in-memory quick-sort if space allows or an external merge-sort otherwise. It then does the standard merge on the sorted samples. The second slightly-less naïve algorithm (2) performs one external sort on the original values (expected values). It then reads the values, samples, resorts with insertion sort, and merges one round at a time. This algorithm should take the same amount of time to sample and sort as our algorithm but must read in the sorted expected values in each of the n rounds. In contrast, if x rounds can fit in memory with each round occupying $k+1$ pages, our join algorithm only needs to read the data n/x times. They all undertook 1000 rounds of sampling and joined tables of equal size.

Figure 6(a) shows the average runtime of 4 trials for sample-based merge join and the naive merge-join algorithms presented above on tables of various sizes. The results of the experiments show that our algorithm has roughly linear performance until the cost of doing one external sort outweighs the cost of doing 1000 rounds of linear traversals. In contrast, the naive sort-merge-join algorithm (1) is slower than our algorithm even when the entire sort can be done in memory. The dramatic bump in run-time of (1) occurs around table size = 100,000 when the internal sort becomes an external sort. After this point, the cost of repeated sorts becomes overbearing. Algorithm (2) performs better for smaller datasets mainly because it requires less overhead than our algorithm which must keep track of multiple rounds of sampling at once. However, once the entire table can no longer fit in memory, it must be reread from disk during every round, and the cost of the disk-reads slows it down considerably. Figure 6(b) shows a log plot of the same data to better illustrate performance on the smaller datasets.

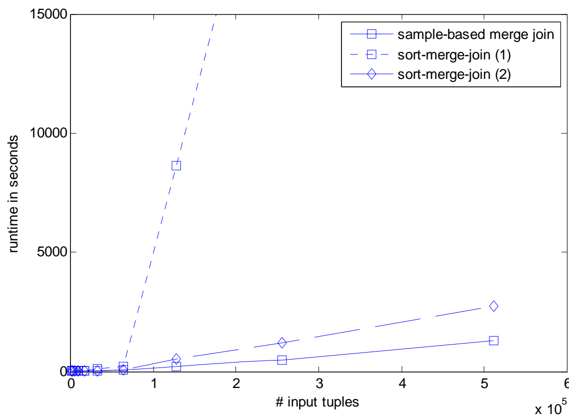

Figure 6(a). Runtimes of the three join algorithms.

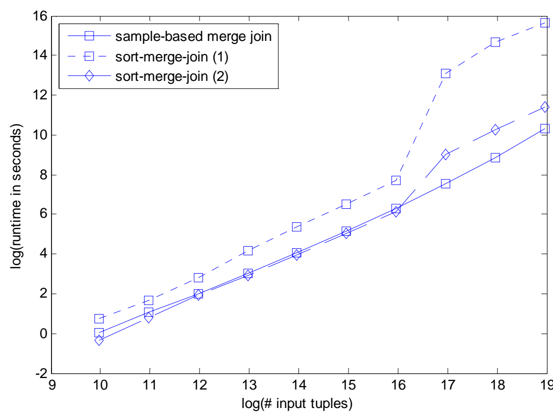

Figure 6(b). Log runtimes of the three join algorithms.

5.

Un n many cases, correlations between the uncertainties must be dealt with as well. Tingjian Ge and I have developed sampling-based semantics for database systems to handle correlated uncertainty. We have also developed an efficient join algorithm that performs well when the join attribute consists of uncertain values that must be sampled and joined multiple times. I have implemented this algorithm and ran extensive experiments on the implementation.

6. REFERENCES

[1] Tingjian Ge, David Grabiner, and Stan Zdonik: A Treatment of Correlated Attribute Uncertainty in Array Database Systems

[2] Y. Bar-Shalom, D. William Dale Blair. *Multitarget-Multi-sensor Tracking: Applications and Advances*, Vol. III, Artech House, Boston, London, 2001.