Abstract—Uncertain data is quite common nowadays in a variety of modern database applications. At the same time, the join operation is one of the most important but expensive operations in SQL. However, join queries on uncertain data have not been adequately addressed thus far. In this paper, we study the SQL join operation on uncertain attributes. We observe and formalize two kinds of join operations on such data, namely v-join and d-join. They are each useful for different applications. Using probability theory, we then devise efficient query processing algorithms for these join operations. Specifically, we use probability bounds that are based on the moments of random variables to either early accept or early reject a candidate v-join result tuple. We also devise an indexing mechanism and an algorithm called Two-End Zigzag Join to further save I/O costs. For d-join, we first observe that it can be reduced to a special form of similarity join in a multidimensional space. We then design an efficient algorithm called condensed d-join and an optimal condensation scheme based on dynamic programming. Finally, we perform a comprehensive empirical study using both real datasets and synthetic datasets.

I. INTRODUCTION

Handling uncertain data is crucial for many applications today. Often, due to the inability to cleanse the data completely to its definite state, a data management system must be able to represent uncertain data to avoid losing useful information. For example, in hazardous weather monitoring [12, 20], a wide-area radar network collects a large amount of meteorological data to predict natural disasters such as tornados and severe storms. Uncertainty arises in radar scanning, integrating different radar streams, and data compression, etc. Due to the lack of complete knowledge, such uncertainty cannot be completely removed. A good system managing such data must be able to accommodate the uncertainty and make informed decisions based on probabilistic query results. Forcing uncertain attributes to be deterministic ones can cause significant information loss in query results, possibly leading to wrong decisions as to the occurrence or location of a natural disaster.

In this paper, we focus on an important kind of query, namely join queries, on uncertain attributes. We show that there are two useful types of join operations specific to uncertain attributes: value join (v-join) and distribution join (d-join). V-join is a natural extension of the join operation on deterministic data. Let us first look at an example.

Example 1 (v-join). In Figure 1, we would like to examine the temperature attributes in table R and in table S, and find pairs that are very close. Note that both temperature attributes are uncertain and contain distributions, which appear in various forms. For instance, \( N(78, 5) \) denotes a normal distribution with mean 78 and variance 5, while \( U(70, 75) \) is a uniform distribution in the range [70, 75] and “hist(…)” indicates a histogram representation whose details we omit for clarity. The query is:

\[
\text{SELECT R.ID, S.ID FROM R, S WHERE R.temperature = S.temperature} \quad 1.0, 0.8
\]

This is called probabilistic threshold join query in previous work [7]. The interpretation of the join predicate is that with probability at least 0.8, the difference between the two join attributes is no more than 1.0 degree, i.e., \(|R.temperature - S.temperature| \leq 1.0\).

Table R

<table>
<thead>
<tr>
<th>ID</th>
<th>Temperature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>N(78, 5)</td>
</tr>
<tr>
<td>2</td>
<td>U(70, 75)</td>
</tr>
<tr>
<td>3</td>
<td>N(86, 10)</td>
</tr>
<tr>
<td></td>
<td>hist(...)</td>
</tr>
</tbody>
</table>

Table S

<table>
<thead>
<tr>
<th>ID</th>
<th>Temperature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>N(85, 6)</td>
</tr>
<tr>
<td>2</td>
<td>U(92, 94)</td>
</tr>
<tr>
<td>3</td>
<td>N(77, 8)</td>
</tr>
<tr>
<td>4</td>
<td>hist(...)</td>
</tr>
</tbody>
</table>

Fig. 1 Illustrating v-join between two uncertain attributes

Our main contribution for v-join is efficient query processing. We use probability theory to either early reject or early accept candidate join result tuples. Only when we cannot make early decisions, do we resort to the old method of computing a double integral. This improves the performance significantly. Further-more, to cut down the I/O costs of v-join, we propose an indexing algorithm which uses the commonly available B+ tree indexes as the underlying tool. Based on this indexing, we further devise a TwoEndZigzagJoin algorithm, saving both I/O and CPU costs.

For uncertain attributes (either numerical or categorical), there is a special kind of join, which we call d-join. The idea of d-join is to treat probability distributions as “objects” and the join operation is based on the similarity of two distributions. We now look at some examples.

Example 2 (sensor fusion). For high availability, five sensors redundantly measure the same environmental physical property (e.g., temperature) in a sensor network deployment on Great Duck Island (off the coast of Maine) [27, 22]. Due to the harsh environment and the unreliable nature of the sensors, the readings can have large errors. A central
A database system performs a sensor fusion and uses machine learning techniques (e.g., kernel methods) [3] to obtain a temperature distribution from the five sensors. We record the temperature distributions at various times within two months in two tables (one for each month). We want to query for two time instances (one from each month) that have close temperatures.

Example 3 (data integration). Consider data integration from several sources. We need to perform schema matching and record linkage to combine different versions of the same data entity. However, due to schema and format inconsistencies, a data entity can have a lot of uncertainty. In the integrated database, we model the uncertainty with distributions (for either numerical or categorical values) [13]. If two entities have similar distributions, then they are likely to be close. It is useful to find out this information.

Example 4 (prediction queries). We use different statistical models to predict the stock prices of a large number of companies one week from now [6]. Different models gave different results and again, by using techniques such as kernel methods [3], we can get a distribution of the predicted price of each company, which is stored in relational tables. The query is to ask for pairs of two companies that are likely to have very close stock prices at that time.

In all these three examples, if we were to use v-join, even if two distributions are exactly the same, the probability that the join predicate is satisfied might still be insignificant. Here is a simple example. Suppose in Example 2, the five sensors give readings that are quite different (the difference is more than the v-join value difference parameter $\epsilon$). Thus, the integrated temperature distribution has approximately five buckets, each with the same probability (1/5). Even if we were to do a v-join on two identical distributions as such, the probability that they are within $\epsilon$ apart would be only about $\frac{1}{5}$ when both random variables fall into the same bucket. The observation here is that whether v-join is satisfied or not heavily depends on the "width" of the two distributions (i.e., the uncertainty, or, the entropy [9]). V-join does not compare the two distributions "themselves": two identical distributions may still fail to match. However, in all these examples, the fact that two distributions are close is also useful: it tends to indicate a special relationship of the two tuple entities that are being joined; i.e., their uncertain attributes are likely to be close in spite of the uncertainty. Essentially, we treat uncertainty, or, the entropy [9]). V-join does not compare the two distributions "themselves": two identical distributions may still fail to match. However, in all these examples, the fact that two distributions are close is also useful: it tends to indicate a special relationship of the two tuple entities that are being joined; i.e., their uncertain attributes are likely to be close in spite of the uncertainty. Essentially, we treat probability distributions themselves as objects and we are joining such objects. This leads to our definition of d-join in Section II.

As we do for v-join, we propose efficient algorithms for d-join. V-joins and d-joins are each useful for different applications. We conduct systematic experiments to carefully verify the efficacy of our algorithms and observe the effects of different parameter changes. In summary, the contributions of this work are:

- An efficient v-join algorithm TwoEndZigzagJoin by combing the usage of probability theory and a new indexing mechanism.
- An optimal condensation scheme for d-join based on dynamic programming.
- An efficient d-join algorithm called condensed d-join.
- A systematic empirical study on two real datasets and some synthetic datasets.

The remainder of the paper is organized as follows. In Section II, we present the semantics formulation of join queries on uncertain data. We then discuss how to efficiently process v-join in Section III using probability theory and our indexing scheme. In Section IV we propose the condensed d-join algorithm and an optimal condensation scheme based on dynamic programming. We perform a comprehensive empirical study in Section V. Finally, we list the related work in Section VI and conclude in Section VII.

II. SEMANTICS FORMULATION

A. Preliminaries

Let us start with some needed background.

Definition 1 (variation distance) [23]. The variation distance between two probabilistic distributions $D_1$ and $D_2$ on a countable state space $S$ is given by

$$VD(D_1, D_2) = \frac{1}{2} \sum_{x \in S} |D_1(x) - D_2(x)|$$

where $D_i(x)$ denotes the probability of state $x$ in $D_i$ ($i = 1, 2$).

Variation distance is a frequently-used metric in probability theory that indicates the difference between two distributions [23]. The factor $\frac{1}{2}$ in the definition guarantees that the variation distance is between 0 and 1. There are other ways to measure the distance between two distributions, such as the KL-divergence [9]. A salient advantage of variation distance over KL-divergence is that variation distance is a true metric while KL-divergence is not (since it is not symmetric and does not satisfy the triangle inequality).

We also need the following definition for d-join.

Definition 2 (domain partition scheme). The domain partition scheme for an uncertain attribute is a many-to-one mapping of values in the domain of the uncertain attribute to a countable number of states.

Example 5 (domain partition scheme). If the domain of an uncertain attribute is all positive real numbers, then one possible domain partition scheme is based on a parameter step: we map all attribute values in the interval $[0, \text{step}]$ to state 1, all values in $[\text{step}, 2\times\text{step}]$ to state 2, and so on.

B. Join on Uncertain Attributes

We are now ready to study the join operation on uncertain attributes. Consider two relations $R$ and $S$ that have uncertain attributes $RA$ and $SB$. In $R$, each record’s $A$ attribute is a probability distribution, rather than a single value, as in deterministic databases. The distribution can be encoded in various ways, including well-known distributions (e.g., a normal distribution) and histograms. The same is true for $SB$. 

We denote a join operation between $R$ and $S$ on attributes $R.A$ and $S.B$ as $R \bowtie_{A,B}$, where $\epsilon$ and $p$ are optional parameters. There are two types of join: value join ($v$-join) and distribution join ($d$-join). A $v$-join has a join predicate that is an (approximate) equality or an inequality with some probability threshold. For example, a $v$-join predicate can be $R.A = S.B$, which means $\Pr(|R.A - S.B| < \epsilon) \geq p$. This is a probabilistic version of a band join [11]; for deterministic data, when the predicate is $|R.A - S.B| < \epsilon$, it is a band-join. Another example is $R.A < S.B$, which means $\Pr(R.A < S.B) \geq p$ ($\epsilon$ is not present here). $\epsilon$ usually denotes a small value and $p$ is a probability threshold. Note that, when it is clear from the context, we often use $R.A$ to denote the random variable that represents the $A$ attribute of a tuple in $R$, and likewise for $S.B$. A $d$-join predicate is denoted as $R.A \bowtie_{A,B}$. It is equivalent to

$$VD(R.A, S.B) \leq \epsilon,$$

where $VD(R.A, S.B)$ denotes the variation distance between a distribution in $R.A$ and a distribution in $S.B$, and $R.A$ and $S.B$ have a common set of states resulting from their domain partition schemes.

### III. Efficiently Processing V-Join

#### A. Using the First Two Moments

Our query processing techniques for $v$-join are based on the probability theory. Specifically, the $k$'th moment of a random variable $X$ is defined as $E[X^k]$. The moments are a concise way to describe the nature of the distribution of a random variable. The first moment is the expectation of the random variable while the first two moments determine the variance of the random variable: $\text{Var}[X] = E[X^2] - (E[X])^2$. In fact, all moments of a variable together uniquely define its distribution [23]. Simply computing and storing the first two moments (or equivalently, the expectation and variance) of a random variable (in our context, an uncertain attribute of a record is a random variable) incurs little overhead but, as we show, is very useful in making quick decisions during $v$-join in order to improve the speed. In some well-known distributions, such as Gaussian, the expectation and variance come for free, since they are part of the description of the distribution.

**Probabilistic Band Join.** Perhaps the most often used $v$-join is the probabilistic band join; i.e., when the join predicate is $R.A < S.B$. The basic method for evaluating this predicate is through computing a double integral of the following form:

$$p' = \int_{x} \int_{y} f_1(x)f_2(y) \, dx \, dy$$

where $x$ is a random variable in $R.A$, $y$ is a random variable in $S.B$ and $f_1(x)$ and $f_2(y)$ are the density functions of $x$ and $y$, respectively. The result $p'$ is the probability that $R.A$ and $S.B$ (of two tuples) are at most $\epsilon$ apart. Clearly, the $v$-join predicate is satisfied if and only if $p' \geq p$.

The problem with (1) is that it is very CPU expensive, as we see in the experiments. Therefore, we wish to use probability bounds to improve the speed of evaluating such a predicate. Define a random variable $X = R.A - S.B$. Then the predicate is equivalent to:

$$\Pr(|X| < \epsilon) > p$$

Let $E(X) = E(R.A) - E(S.B) = \lambda$. Then we have the following three cases, as shown in Figure 2.

**Case (a): $\lambda > \epsilon$.** We would like to see if (2) must be false, in which case we can exclude the tuple pair. From the one-sided Chebyshev inequality [23], we have:

$$\Pr(|X| < \epsilon) \leq \Pr(X < \epsilon) = \Pr(\lambda - X > \lambda - \epsilon) < \frac{\text{Var}(X)}{\text{Var}(X) + (\lambda - \epsilon)^2}$$

If

$$\frac{\text{Var}(X)}{\text{Var}(X) + (\lambda - \epsilon)^2} \leq p,$$

then (2) must be false.

**Case (b): $\lambda < -\epsilon$.** Similar to case (a), we would like to use the one-sided Chebyshev inequality to see if we can determine that (2) must be false and rule out the tuple pair:

$$\Pr(|X| < \epsilon) \leq \Pr(X > -\epsilon) = \Pr(X - \lambda > -\epsilon - \lambda) < \frac{\text{Var}(X)}{\text{Var}(X) + (\lambda + \epsilon)^2}$$

If

$$\frac{\text{Var}(X)}{\text{Var}(X) + (\lambda + \epsilon)^2} \leq p,$$

then (2) must be false.

**Case (c): $-\epsilon < \lambda < \epsilon$.** In contrast to the previous two cases, we would like to see if (2) must be true and hence the tuple pair satisfies the $v$-join condition. From one-sided Chebyshev inequality, we have:

$$\Pr(X > \epsilon) = \Pr(X > \lambda + \epsilon - \lambda) < \frac{\text{Var}(X)}{\text{Var}(X) + (\epsilon - \lambda)^2}$$

$$\Pr(X < -\epsilon) = \Pr(X > -\epsilon) = \frac{\text{Var}(X)}{\text{Var}(X) + (\epsilon + \lambda)^2}$$

$$\Pr(|X| > \epsilon) = 1 - \Pr(X > \epsilon \text{ or } X < -\epsilon) > 1 - \frac{\text{Var}(X)}{\text{Var}(X) + (\epsilon - \lambda)^2} - \frac{\text{Var}(X)}{\text{Var}(X) + (\epsilon + \lambda)^2}$$

The last inequality is due to the union bound [23]. Clearly, if

$$1 - \frac{\text{Var}(X)}{\text{Var}(X) + (\epsilon - \lambda)^2} - \frac{\text{Var}(X)}{\text{Var}(X) + (\epsilon + \lambda)^2} \geq p$$

then (2) must be true. Now suppose we are only given the moments of the two fields being joined. For clarity, we write $A$ for $R.A$ and $B$ for $S.B$. For the above methods to work, we need to express $\lambda$ and $\text{Var}(X)$ using the moments of $A$ and $B$. From the linearity of expectation, $\lambda = E(X) = E(A) - E(B)$. With the typical assumption that $A$ and $B$ are independent, we have

$$\text{Var}(X) = \text{Var}(A) + \text{Var}(B) = E(A^2) - E^2(A) + E(B^2) - E^2(B).$$

Therefore, only using the first two moments of $A$ and $B$, we can quickly exclude the $(A, B)$ pair from the join result (Case a and b) or include it in the result (Case c) if the conditions in those cases are met. If the pair is neither excluded nor included, we need to resort to the “old-fashioned” way of computing the actual probability that $|A - B| < \epsilon$ through a double integral (or summation if they are discrete) as in (1) and see if it is greater than $p$. We can save a great deal of
computational cost by using moments and probabilistic bounds to make quick judgments.

**Other Inequality V-Joins.** Thus far we have only considered probabilistic band join; we now turn to other inequality v-joins. We only demonstrate \( R.A \prec S.B \) and we can apply similar techniques to other inequalities. Again we define a random variable \( X = R.A - S.B \). Let \( E(X) = \lambda \). We now examine two cases:

**Case (a):** \( \lambda < 0 \). Then,

\[
Pr(R.A \geq S.B) = Pr(X \geq 0) = Pr(X - \lambda \geq -\lambda) < \frac{Var(X)}{Var(X) + \lambda^2}
\]

The last inequality follows from one-sided Chebyshev inequality. If \( \frac{Var(X)}{Var(X) + \lambda^2} \leq 1 - p \) it must be true that \( Pr(R.A < S.B) \geq p \) and the predicate is satisfied.

**Case (b):** \( \lambda \geq 0 \). Then we see if we can exclude the tuple pair:

\[
Pr(R.A \prec S.B) = Pr(X < 0) = Pr(X - \lambda > \lambda) < \frac{Var(X)}{Var(X) + \lambda^2}
\]

If \( \frac{Var(X)}{Var(X) + \lambda^2} \leq p \) it must be true that \( Pr(R.A \prec S.B) < p \) and the tuple pair is excluded from the result. Details such as obtaining \( \lambda \) and \( Var(Y) \) from the moments of \( R.A \) and \( S.B \) are the same as in the discussions for probabilistic band join.

**B. Using Higher Moments for Processing V-Join**

When higher moments exist, for probabilistic band join, we can have a different way of early rejecting or early accepting candidate result tuples. The idea is to define \( Y = X^2 \), where, as before, \( X = A - B \). Then,

\[
Pr(|A - B| < \epsilon) = Pr(|X| < \epsilon) = Pr(X^2 < \epsilon^2) = Pr(Y < \epsilon^2)
\]

Now let \( E(Y) = \lambda^2 \). Then we consider two cases:

**Case (a):** \( \lambda^2 > \epsilon^2 \). Then

\[
Pr(Y < \epsilon^2) = Pr(Y^2 - \lambda^2 > \lambda^2 - \epsilon^2) < \frac{Var(Y)}{Var(Y) + (\lambda^2 - \epsilon^2)^2}
\]

If \( \frac{Var(Y)}{Var(Y) + (\lambda^2 - \epsilon^2)^2} \leq p \), then our target inequality (2) in Section III-A must be false (i.e., early reject).

**Case (b):** \( \lambda^2 \leq \epsilon^2 \). Then

\[
Pr(Y > \epsilon^2) = Pr(Y - \lambda^2 > \lambda^2 - \epsilon^2) < \frac{Var(Y)}{Var(Y) + (\lambda^2 - \epsilon^2)^2}
\]

\[
Pr(Y < \epsilon^2) > 1 - \frac{Var(Y)}{Var(Y) + (\lambda^2 - \epsilon^2)^2} = \frac{(\lambda^2 - \epsilon^2)^2}{Var(Y) + (\lambda^2 - \epsilon^2)^2}
\]

If the right hand side of the above inequality is at least \( p \), then (2) must be true (i.e., early accept).

What remains is to express \( \lambda^2 \) and \( Var(Y) \) using the moments of \( A \) and \( B \). From the linearity of expectation, we have

\[
\lambda^2 = E(Y^2) = E(A^2 + B^2 - 2E(A)E(B)) = E(A^2) + E(B^2) - 2E(A)E(B)
\]

\[
Var(Y) = E(Y^2) - (\lambda^2)^2
\]

From these three equalities we can see that both \( \lambda^2 \) and \( Var(Y) \) can be obtained from the four moments of \( A \) and \( B \). If there are four moments available for \( A \) and \( B \), one can simply compare the method above with the one in Section III-A, and early reject or early accept when either method does so. Let us look at an example.

**Example 4 (early decisions).** Continuing with the schemas in Example 1, let us assume that the v-join predicate is \( R.temperature = S.temperature \) and table \( R \) has a record with temperature attribute \( N(70, 4) \) (i.e., Normal distribution with mean 70 and variance 4) while table \( S \) has an \( N(75, 4) \).

Using the method in Section III-A, we have \( \lambda = 70 - 75 = -5 < \epsilon \). Thus, we are in case (b). Now

\[
\frac{Var(Y)}{Var(Y) + (\lambda^2 - \epsilon^2)^2} = \frac{4 + 4}{(4 + 4 + (-5)^2)} = \frac{8}{17} \leq p = 0.6
\]

We conclude that this tuple pair can be rejected.

On the other hand, we try the method in this sub-section. We can easily compute the four moments for both distributions (details omitted in the interest of space). We then have \( \lambda^2 = 33 \) and \( Var(Y) = 2017 \). Since \( \lambda^2 > \epsilon^2 = 4 \), we are in case (a). But this time

\[
\frac{Var(Y)}{Var(Y) + (\lambda^2 - \epsilon^2)^2} = \frac{2017}{2017 + (33 - 4)^2} = 0.7 > p = 0.6
\]

Thus we cannot early reject the tuple pair using this method.

In Example 4, the method in Section III-A actually works better, in addition to the fact that it has less overhead by only requiring the first two moments. Indeed, we are able to come up with an example where using four moments is better: let one distribution be \( \{0, 0.3\}, \{1, 0.35\}, \{2, 0.35\} \) and the other distribution be \( \{0, 0.3\}, \{1, 0.35\}, \{2, 0.35\} \) where the first number in each bracket is a value and the second number is its probability. One can verify that in this rather contrived example, if \( \epsilon = 0.1 \) and \( p = 0.6 \), using four moments will be able to early reject it while the other method cannot. Nevertheless, with all the real datasets and generated synthetic datasets in our empirical study (Section V), using the method in Section III-A is more effective.

**C. Indexing and Two-End Zigzag Join**

If there is no way to index \( A \) or \( B \) columns, a v-join query has to perform a block nested loop join, which can potentially cause a big I/O cost, even though individual pair-wise comparisons can take advantage of the results in Section III-A. The natural question is whether we can somehow order or index an uncertain attribute for v-join. Note that the indexing methods proposed in previous work [8, 1] only handle probabilistic threshold queries over an interval, but not joins.

We now describe how we index an uncertain attribute \( B \). We build a B+ tree index using the expectations of each field in \( B \) as the index key. But at the leaf level, each index record (called leaf entry) contains five fields: \(<e, var, var', var_r, RID>\), where \( e \) is the expectation of the entry (i.e., the index key, as in internal nodes), \( var \) is its variance, \( var_r \) is the maximum variance amongst this leaf entry and all the leaf entries to the left of it (i.e., with keys no more than \( e \)), and
var, is the maximum variance amongst this leaf entry and all the leaf entries to the right (i.e., with keys at least e). As usual, RID is the record ID that points to the base table. An internal node only has index keys (e) and pointers.

In Figure 3, we show the algorithm JoinOneRecord that only joins one record of the outer table with the inner table through its index. In step (3), starting from l, we follow the leaf pointers to the left, and stop at an index entry < e, var, varl, varr, RID> that satisfies both conditions e < e0 – e and
\[
\frac{\text{var}_l + \text{var}_r}{\text{var}_l + \text{var}_r + (e_0 - e)^2} > p^{-1}.
\]
but the next index entry on the left does not satisfy both conditions. Likewise, in step (4), we iterate through the index entries to the right. Thereafter, in steps (5) to (8), we check through the tuples in between the two found in steps (3) and (4) as we do before in Section III-A. Thus, in addition to saving CPU costs by using probability theory, we also reduce I/O costs by this index.

Fig. 3 Algorithm JoinOneRecord

Theorem 1. The JoinOneRecord algorithm in Figure 3 gives the correct result.

Proof. We only need to show that any leaf entry i outside the range from i1 to i2 will not match with r. In step (3), as we traverse the leaf entries to the left, by the construction of the index, the e value of the index entries either drops or stays the same. Now, because the index entry i1 satisfies e < e0 – e, it must also be the case for all index entries to the left of i1. Thus, from step (3), it must be the case that for the index entry on the left of i1,
\[
\frac{\text{var}_l + \text{var}_r}{\text{var}_l + \text{var}_r + (e_0 - e)^2} \leq p.
\]

Observe that
\[
\frac{\text{var}_l + \text{var}_r}{\text{var}_l + \text{var}_r + (e_0 - e)^2} = 1 + \frac{(e_0 - e)^2}{\text{var}_l + \text{var}_r + (e_0 - e)^2}
\]
will increase if e decreases or if varl decreases. Therefore, if e decreases or if varl decreases, so does \(\frac{\text{var}_l + \text{var}_r}{\text{var}_l + \text{var}_r + (e_0 - e)^2}\).

Combining this result with inequality (3), and with the fact that all entries to the left of i1 must have e < e0 – e and varl ≤ var, we know that for all index entries to the left of i1, e0 – e ≥ e and
\[
\frac{\text{var}_l + \text{var}_r}{\text{var}_l + \text{var}_r + (e_0 - e)^2} \leq p.
\]
Thus, they all fall in case (a) of Section III-A and they all satisfy the early rejection condition.

With similar reasoning, we can show that all index entries to the right of i2 fall in case (b) of Section III-A and they are all early rejected as well. We conclude that the algorithm in Figure 3 gives the correct result.

The JoinOneRecord algorithm immediately suggests an index nested-loop join algorithm by iterating through all records in the outer table. However, this still involves a lot of traversals through all levels of the index (one search per record in the outer table). We can use an index for the outer table as well, in order to better utilize sequential block access and caching. Let T1 and T2’s indexes be Idx1 and Idx2, respectively.

Line (3) of Algorithm JoinOneRecord finds the left end i1 and line (4) finds the right end i2 of an interval of index entries in Idx2 that we need to examine (in lines 5 to 8). Thus, for a record r in the outer table, let \(I(r)\) denote the interval from i1 to i2, and let \(L(r)\) and \(R(r)\) denote the left end (i1) and right end (i2) of this interval, respectively.

Similarly, consider a leaf block b of Idx1, which contains a set of records. There is also an interval of leaf entries in Idx2, denoted as \(I(b)\), which is the smallest interval that contains the \(I(r)\)'s for every record r in b. Let \(L(b)\) and \(R(b)\) denote the two ends of \(I(b)\).

Input: Outer table T1, index Idx1, inner table T2, index Idx2.
Output: Result of v-join.

(1) Let p be a pointer to a leaf block of Idx2. Set p = NULL.
(2) for each leaf block b of Idx1, from left to right, do
(3) if p is inside \(I(b)\) then
(4) for each page p from p to L(b) do
(5) Join all records in b and p' as in lines 6, 7 of Fig. 3
(6) end for
(7) for each page p' from p to R(b) do
(8) Join all records in b and p' as in lines 6, 7 of Fig. 3
(9) end for
(10) p = p'
(11) else // p is outside I(b)
(12) Use the 1st record in b to probe Idx2 as in line 2 of Fig 3
(13) p = current leaf block of Idx2
(14) Do lines 4 to 9
(15) end if
(16) end for

Fig. 4 Algorithm ZigzagJoin
We now have an intermediate algorithm, called ZigzagJoin, as shown in Figure 4. The basic idea is to access the outer table through its index leaf nodes, which are sorted blocks. Then for the inner table, we “zigzag” through a bigger interval of index entries for the whole block of outer table. There are a lot of overlaps among these intervals; thus we obtain great caching behavior and do not have to go through the upper levels of the index if we can avoid it.

In line 2, the algorithm iterates through the sorted blocks of the outer table through its index leaves. We maintain a pointer $p$ that points to the current leaf block of the inner table index that we access (line 1). In line 3, it is easy to check whether the current leaf block $p$ is inside the interval $I(b)$ associated with an outer block $b$. We just go through each record $r$ in $b$ and check whether $p$ is inside $I(r)$ (lines 3 and 4 of JoinOneRecord). Likewise, we can check if we reach $L(b)$ in line 4 and $R(b)$ in line 7. Only in the event that $p$ is outside $I(b)$ (line 11), do we use the upper levels of the index to locate a leaf block that is inside $I(b)$ (line 12). For clarity, we do not show the details at line 4 (or line 7) about what we do if there is a “gap” between $p$ and $I(b)$, i.e., what if $p'$ is not in $I(r)$, for any $r$ in $b$. The solution is similar to line 12; i.e., we use the next record in $b$ (on the left) to probe $Idx_2$ and find the next matching range within $I(b)$ to continue the loop.

![Fig. 5 Illustrating the TwoEndZigzagJoin algorithm](image)

What if there is a record in the inner table that has a large variance and therefore the var$_1$ or var$_2$ of other leaves has to be large as well? From lines 3 and 4 of JoinOneRecord, that implies we need to read and examine many more pages which slows down the join. This motivates us to start two concurrent instances of ZigzagJoin that work together. The idea is that whichever instance goes faster will bring the v-join closer to finish, as explained next.

Our final algorithm, namely TwoEndZigzagJoin, is a simple extension of ZigzagJoin. It starts two instances of ZigzagJoin, called $Z_1$ and $Z_2$. $Z_1$ uses $T_1$ as the outer table, while $Z_2$ uses $T_2$ as the outer table and scan the leaf blocks of $Idx_2$ from right to left instead (in line 2 of Fig 4).

In line 2 of Fig. 4, $Z_1$ has a cursor, denoted as $b_1$, which is a pointer to a leaf block of $Idx_1$, moving from left to right; similarly, $Z_2$ has a cursor $b_2$ in $Idx_2$, moving from right to left. For $Z_1$, denote the leaf block pointer in $Idx_2$ as $p_2$ (line 1 of Fig. 4); denote $Z_2$’s leaf block pointer in $Idx_1$ as $p_1$. Figure 5 illustrates the TwoEndZigzagJoin.

$Z_1$ and $Z_2$ are run concurrently in lockstep. We now describe the scheduling. First let $Z_1$ advance one block in $b_1$ (i.e., finish lines 3 to 15 in Fig. 4 for this block); let the number of blocks that $p_1$ has gone through (in $Idx_1$) be $n_1$. Now, use $n_2$ as a budget for $Z_2$.

That is to say, run $Z_2$ and let it advance one block in $b_2$; let the number of blocks that $p_1$ has gone through (in $Idx_1$) be $n_1$. If $n_1 < n_2$, we repeat the one-block execution of $Z_2$ with a new budget $n_2 = n_2 - n_1$. Otherwise ($n_1 \geq n_2$), we repeat the one-block execution of $Z_1$ with a new budget $n_1 = n_1 - n_2$.

Since $p_2$ (from $Z_1$) moves in a zigzag manner with the overall direction from left to right in $Idx_2$, and $b_2$ (from $Z_2$) moves from right to left in $Idx_2$, they can meet at some point. Observe that we have the invariant that $Z_2$ must have finished all join results that involve any block to the right of $b_2$. Thus, $p_2$ never needs to move beyond $b_2$, but just skips $b_2$ and all blocks to its right. By the same token, $p_1$ never needs to move beyond $b_1$, but just skips $b_1$ and all blocks to its left. The TwoEndZigzagJoin algorithm finishes when either $Z_1$ or $Z_2$ finishes.

We now show an example of the scheduling of the join algorithm.

**Example 5 (TwoEndZigzagJoin).** Let us look at a possible schedule:

One block of $b_1$ (Z1)

- $p_2$: 10 blocks
  - budget for $Z_2$: $n_2 = 10$

One block of $b_2$ (Z2)

- $p_1$: 3 blocks
  - budget for $Z_2$: $n_2 = 10 - 3 = 7$

One block of $b_2$ (Z2)

- $p_1$: 4 blocks
  - budget for $Z_2$: $n_2 = 7 - 4 = 3$

One block of $b_1$ (Z1)

- $p_2$: 10 blocks
  - budget for $Z_2$: $n_2 = 10 - 1 = 9$

One block of $b_2$ (Z2)

The schedule above starts with $Z_1$ advancing its $b_1$ cursor by one block. This block joins with 10 blocks at $p_2$ in $Idx_2$ (use Fig. 5 for reference). Now, $Z_2$ has a budget of $n_2 = 10$ due to this. It is $Z_2$’s turn to advance one block at $b_2$, which only touches 3 blocks at the other side ($p_1$). Therefore, $Z_2$ has not used up its budget, and it is still $Z_2$’s turn with a reduced budget of $n_2 = 10 - 3 = 7$. The dynamic scheduling proceeds like this.

Another possible solution to the large variance problem is to first separate large variance entries into an “outlier page” and to handle them alone. If we need to pick out so many entries that they do not all fit in the buffer, then the outliers themselves can be organized as a B+ tree. We omit the details in the interest of space. It is also straightforward to extend the v-join algorithm using indexes to other inequality v-joins as described in Section III-A.
IV. Efficiently Processing D-Join

A. The Condensed D-Join Algorithm

In this section, we examine how we can process d-join queries efficiently. We can perform a d-join on two uncertain attributes if their domain partition schemes (Section II) result in a common set of states. Let the size of the state space \( S \) resulting from the domain partition schemes be \( n \). Then the “features” of an uncertain distribution with respect to \( S \) can be described as \((p_1, p_2, ..., p_n)\), meaning that the uncertain field has probability \( p_i \) to be in state \( s_i \). By taking this vector, we can map an uncertain distribution to a point in the \( n \)-dimensional space. It is then easy to verify that the variation distance between two distributions exactly maps to half of the L-1 interval \([0, 1]\). The condensation scheme of the attribute and where
defined below.

**Definition 2 (condensation scheme).** A condensation scheme for an uncertain attribute is an onto function \( f : S \rightarrow S' \), where \( S \) is the original state space determined by the domain partition scheme of the attribute and \( S' \) is a state space with a smaller cardinality, i.e., \(|S'| < |S| \). \( S' \) is called the condensed state space.

**Input:** Two uncertain attributes \( R.A \) and \( S.B \) whose domain partition schemes have the same states; and value \( \epsilon \).

**Output:** Pairs of \( R.A \) and \( S.B \) that satisfy \( R.A \sim S.B \).

1. **Precomputation step:** Determine the best condensation scheme for either \( R.A \) or \( S.B \) using the algorithm in Section 4.2.
2. In the condensed state space determined in (1), we get the new distributions for all fields in \( R.A \) and \( S.B \). If a new state is the merge of a number of previous states, then its probability is the sum of the original states.
3. **Phase 1:** Use any existing similarity join algorithm to compute the join result based on the smaller state space, using the L-1 distance metric and the distance parameter \( 2\epsilon \).
4. **Phase 2:** Among the qualified tuple pairs selected in (3), further refine the selections by computing the \( VD \) over the original state space.

**Fig. 6 Condensed d-join algorithm**

Step (1) of the condensed d-join algorithm is to determine the optimal condensation scheme according to any one side of the join and is typically precomputed. The condensation algorithm combines a number of neighboring states into one and sums up their probabilities. We thus get the new probability distributions in step (2). Phase 1 of the condensed d-join is performed in step (3), where we essentially reduce the d-join problem to the similarity join of multidimensional points (with the parameter value \( 2\epsilon \)). Because of the reduced dimensionality, it is much faster. Over the qualified tuple pairs, we perform the second phase, which is a post-processing as shown in step (4). As we can see, the goal of the two phase approach is to avoid the slow performance caused by high dimensionality. The quality of the condensation scheme is of a critical role here since it impacts the number of false positives that must be filtered out in the post-processing phase. We study the optimal condensation scheme in details in Section IV-B.

We show the correctness of the condensed d-join algorithm.

**Lemma 1.** Over the original state space determined by the domain partition scheme, let distribution \( D_1 \) come from \( R.A \) and \( D_2 \) come from \( S.B \). After the condensation, let the distributions (in step 2) be \( D_1' \) and \( D_2' \). Then \( VD(D_1', D_2') \leq VD(D_1, D_2) \).

**Proof.** Suppose the condensation scheme merges \( k \) states \( s_1, s_2, ..., s_k \) into a single state \( s' \). Let \( D_1 \) have probabilities \( p_{11}, p_{12}, ..., p_{1k} \) and \( D_2 \) have probabilities \( p_{21}, p_{22}, ..., p_{2k} \) in those states, respectively. Then step (2) of the algorithm indicates that \( D_1' \) has probability \( p_{11'} = p_{11} + p_{12} + ... + p_{1k} \) in state \( s' \) while \( D_2' \) has probability \( p_{21'} = p_{21} + p_{22} + ... + p_{2k} \) in state \( s' \). It holds that
\[
|p_{11'} - p_{21'}| = |(p_{11} - p_{21}) + (p_{12} - p_{22}) + ... + (p_{1k} - p_{2k})| \\
\leq |p_{11} - p_{21}| + |p_{12} - p_{22}| + ... + |p_{1k} - p_{2k}|
\]

Thus, iterating this over all states of \( D_1' \) and \( D_2' \), summing up the inequalities as produced above, and finally dividing both sides of the resulting inequality by \( 2 \), we get \( VD(D_1', D_2') \leq VD(D_1, D_2) \), which directly follows from the definition of \( VD \).

**Theorem 2.** Condensed d-join algorithm gives the correct result.

**Proof.** From Lemma 1, we know that if in the original state space, the \( VD \) between two distributions is less than \( \epsilon \), then it must also be true after the condensation. Thus, phase 1 of the d-join (step 3) will not miss any result tuples that should be returned. Finally, the second phase of the algorithm filters out all false positives.

B. The Optimal Condensation Scheme

Consider an uncertain attribute and a condensation scheme that reduces the number of its states from \( n \) to \( k \) \((k < n)\). The question now is how we should merge the states in the original state space \( S \). Let us look at a motivating example. Figure 7 shows the distributions of an uncertain attribute. The (yellow) solid vertical lines describe the domain partition scheme: an interval between two neighboring lines is a state. Suppose the (blue) dotted lines indicate a potential condensation scheme: there are three condensed states: the interval \([v_1, v_2]\) is the first condensed state, \([v_2, v_3]\) is the
second, and \([v_3, v_4]\) is the third. It appears to be a fair condensation scheme as each condensed state contains about the same number of the original states. However, let us suppose that 1000 distributions fall in the middle range, i.e., between \(v_2\) and \(v_3\) while there is only one distribution in the first and third condensed states, respectively.

![Fig. 7 Illustrating the necessity of a good condensation scheme](image)

Then this condensation scheme loses a lot of information: all of the 1000 distributions have the same distribution \((0, 1, 0)\) in the condensed state space (each number is the probability of one state). In other words, it is not discriminative. In the condensed space, if one of the 1000 distributions matches with a distribution in another column for d-join, so will all other 999 distributions. We therefore need a principled algorithm to make the condensation scheme more discriminative.

But how to make it discriminative? The idea is to make the new distributions after applying the condensation scheme as faithful as possible to the original ones. The faithfulness is again measured by variation distance. Clearly condensation would lose some information about the distributions. Thus we would prefer a scheme that would result in new distributions that have the minimum distance from the original ones, which would prefer a scheme that would result in new distributions that would lose some information about the distributions. Thus we again measured by variation distance. Clearly condensation scheme as each condensed state contains about the 1000 distributions fall in the middle range, i.e., \([v_2, v_3]\) while there is only one distribution in the first and third condensed states, respectively. The point \(d_1'\) corresponds to the conversion of the new distribution back to its original state space, which we describe in the previous paragraph. Not knowing where \(d_1\) is a priori, by minimizing the distance between \(d_1\) and \(d_1'\), the distance between \(d_1\) and \(d_2\) is optimally approximated by the distance between \(d_1'\) and \(d_2\). Moreover, this optimization problem is over all distributions in an uncertain column.

We first formalize the problem. An uncertain column has \(N\) probability distributions. The column follows a domain partition scheme that consists of \(n\) states in some serial order (e.g., \(n\) small buckets in value order). The goal of our condensation scheme is to merge some neighboring states in order to reduce them to \(k\) states \((k < n)\). The scheme is chosen in such a way that the variation distance between the new distribution and the original one, summing over all records of the column, is minimized.

Let us denote the optimal (i.e., minimum) sum value (over the whole column) of the variation distances between the new distributions and the original ones as \(D(k, n)\), where \(k\) is the target number of condensed states and \(n\) is the original number of states. We then have the following recursion:

\[
D(k, n) = \min \{ D(k - 1, i - 1) + \sum_{i=1}^{n} C(i, n) \} 
\]

where \(C(i, n)\) is the “cost” of merging states from \(i\) to \(n\) into a single new state for the distribution in record \(r\). This cost is just the part of the variation distance between the two distributions at states from \(i\) to \(n\). More precisely (recall Figure 8),

\[
C(i, n) = \frac{1}{2} \sum_{j=1}^{n} |p_j(i) - a_j(i)|, \text{ where } a_j(i) = \frac{1}{n-i+1} \sum_{j=i}^{n} p_j
\]

Here \(p_j(i)\) is the probability of the \(j\)’th state in the \(r\)’th distribution. We also note the boundary condition that

\[
D(1, i) = \sum_{i=1}^{n} C(i, 1), \text{ for } 1 \leq i \leq n-k+1
\]

![Fig. 9 A dynamic programming algorithm to get the D function (a) and a column scan to obtain the aggregated C values (b).](image)

We then can have an efficient dynamic programming algorithm for this problem, as illustrated in Figure 9(a). The figure shows a “D table” for values of the D function in Equation (1). The row numbers of the D table (1 to \(k\)) are the first parameter of the function while the column numbers (1 to \(n\)) are the second parameter. Our target value is \(D(k, n)\), which is indicated by the red “?” at the bottom right corner of the D table. From the recursion in Equation (1), the target value can be obtained from the values in the row above, assuming we already have all the C values. The whole process can be recursively applied for each cell in the table. We therefore
have a top-down procedure to fill in the shaded region in Figure 9(a) row by row, starting from the boundary condition as described in Equation (3).

In the above algorithm, we assume that we have all the C values. We now describe how to obtain them. We simply do a scan of the whole uncertain column and compute the aggregation of the C values as described in Equation (2). Figure 9(b) illustrates the C table. It is not hard to verify from Equations (1) and (3) that we only need to fill in the shaded region of the C table (C is a two-dimensional array). As we scan the column and get each distribution, we obtain the C values of the shaded region using Equation (2). Because in Equations (1) and (3) we require a sum of the C values over all distributions, we do the aggregation (sum) for each cell of the shaded region of the C table as we scan each distribution of the column one by one. Eventually, when we finish scanning the uncertain column, each cell of the C table contains a sum value. Combining the above two algorithms (i.e., getting the C table followed by getting the D table), we have an efficient method to obtain the optimal condensation scheme.

V. EXPERIMENTS

In this section, we perform a systematic empirical study using two real datasets and some synthetic datasets. We aim to answer the following questions:

- How much performance improvement can we get by using probability bounds for v-join?
- How much do we gain by using our indexing techniques and the TwoEndZigzagJoin presented in Section III-C?
- How effective is the condensed d-join algorithm?
- How does our condensation scheme compare with the simple equal partitioning of the domain?

A. Setup and Datasets

We perform the study using the following datasets:

- A real-world scientific dataset: the Sloan Digital Sky Survey (SDSS) data release 6 [30]. SDSS is one of the most ambitious and influential surveys in the history of astronomy. It covers more than a quarter of the sky and contains more than 930,000 galaxies and more than 120,000 quasars.
- A real-world dataset collected by the CarTel project team [15]. It consists of measurements of actual traffic delays on roads in the greater Boston area performed by the CarTel vehicular testbed [21], a set of 28 taxis equipped with various sensors and a wireless network.
- Some synthetic datasets, with the final one generated using the R-statistical package [29]. With synthetic datasets we can control the various parameters of the data and study their impact on results.

We implemented all the algorithms presented in this paper to study the results. All the experiments were conducted on a 2.4GHz Intel Core 2 Quad CPU machine with 3GB memory.

B. Results for V-Join

First we study v-join using the SDSS dataset and a synthetic dataset. Given several observations of the sky, called runs, astronomers often want to cross-match all the observations of each object from all runs that observed that object [14]. These objects are matched according to their positional attributes - right ascension (ra) and declination (dec) in the J2000 coordinate system. We issue a query Q1 as follows:

```
SELECT o1.id, o2.id
FROM observation1 AS o1, observation2 AS o2
WHERE o1.ra = o2.ra AND o1.dec = o2.dec
```

Q1 selects matching objects from two observation tables o1 and o2 satisfying the condition that their ra values differ by no more than 2 with probability at least 0.9 and their dec values differ by no more than 1 with probability at least 0.9. The dataset contains ra_error and dec_error which are error bars indicating that ra has a normal distribution with σ value ra_error (and likewise for dec).

Figure 10(a) shows the execution times of Q1 using different query processing methods with various sizes of the relations. Specifically, we compare five methods: (1) “manual” v-join in which we need to compute a double integral (Section III-A), (2) using probability bounds (Section III-A), (3) using four moments (Section III-B), (4) using our indexing technique and TwoEndZigzagJoin algorithm in Section III-C, and (5) using static page-level x-bounds as proposed in [7]. For distinction, we call our method (2) dynamic bounds in the figure. For the manual v-join, we have the following optimization: We cut off the tails of a normal distribution at μ ± 3σ and ignore a tuple-pair if no overlap between their intervals. We can see that even with this optimization, manual v-join is still significantly slower than using our dynamic probability bounds. By contrast, using static bounds is not effective due to the variation of distributions. We discuss in details the comparison with [7] in the related work section. Moreover, in all our experiments on v-join, the performance of using four moments turns out to be only slightly better than a manual join; simply using two moments gets much better results. We thus exclude the line for four moments in the figures for clarity. Using indexes, on the other hand, improves the performance considerably since the I/O cost drops significantly.

Furthermore, amongst the tuple pairs in method (1), we examine, in method (2), the number of those that are early accepted or rejected and the number of those that have to be manually determined. This result is shown in Figure 10(b). The majority are early accepted or rejected. We then vary the parameter ε in the first join predicate of Q1 from 1 to 5. Figure 10(c) shows that the execution time increases for all methods as ε increases because more tuple pairs will need to be considered.

We now generate and query synthetic datasets. For each side of v-join, first we generate 500K normal distributions with μ in the range [0, 2000] but vary σ with average values 5, 20, and 40, respectively. We actually choose a value uniformly at random from the interval [σ/2, 3σ/2]. For instance, when σ = 20, we choose a value from [10, 30] as the standard deviation of a distribution. We use ε = 2 and probability threshold 0.9. We compare manual v-join vs. using...
our method. This is shown in Figure 10(d). Clearly, as we increase \( \sigma \), the number of tuple pairs for which we compute a double integral decreases, since fewer tuples fall within the \( 3\sigma_1+3\sigma_2+\varepsilon \) range of a distribution. But within this variant set of tuples, we only examine the fractions of each category in our method (2). One can observe from Figure 10(d) that the fractions of both early rejection and early acceptance drop noticeably. This can be explained from the three inequality conditions of cases (a), (b) and (c) in Section III-A. When the variance increases, they are less likely to be satisfied. In Figure 10(e), on the other hand, we choose the parameter \( \sigma \) to be 20, but vary the range in which the \( \mu \) of the distributions are drawn. The ranges we look at are \([0, 1000]\), \([0, 2000]\), and \([0, 3000]\), respectively. There is only a slight variation in the fractions. This is because when the range increases, the distributions are sparser, but the fractions of each category of method (2) should not change much. The result verifies our expectation.

We finally experiment with more arbitrary shapes of distributions by generating random histogram distributions. Each histogram distribution has five bins, with the bin probabilities chosen uniformly at random, and then the whole distribution is normalized. The result is in Figure 10(f). We can see that the trend in the results is very close to that of the normal distributions shown in Figure 10(a). The only difference is that the execution times are longer because the variances of the distributions are generally bigger, and because the histograms take more space.

### C. Results for D-Join

We now use the traffic delay dataset to study d-join. The dataset contains samples of actual traffic delays on various road segments. We define

\[
\text{congestion\_score} = \frac{\text{speed\_limit}}{\text{length\_delay}}
\]

the denominator is the actual travel speed and the numerator is the speed limit of the road segment. Thus, a higher congestion score implies a more congested road segment. Each road contains one or more measurement record. Each record is considered uncertain and the congestion score of a road is probabilistic [21]. A road generally contains hundreds or
thousands of measurements. We create 50 bins and then bin the samples to collect the statistics of the frequencies of the bins and obtain a discrete distribution for each road segment. We then issue the following query Q2:

```
SELECT r1.segment_id, r2.segment_id
FROM all_roads r1, all_roads r2
WHERE r1.congestion ~ r2.congestion
AND r1.segment_id <> r2.segment_id
```

Q2 is a d-join (self-join) on the congestion score and selects all pairs of road segments that have close congestion score distributions (with $VD$ no more than 0.1). A congestion score distribution of a road segment shows its travel pattern. For example, a main road in a downtown busy area tends to have a narrow distribution at the high end of the score spectrum while a road in a quiet residential area has a narrow distribution at the low score end. Yet some other roads have relatively wide distributions spanning both high and low scores. Such roads include the ones connecting business and residential areas which are congested during rush hours. The result of Q2 essentially gives a graph (road segments are vertices and an edge exists between two vertices if they are in the result). The connected components of the graph show a clustering of the roads, which can be very useful for traffic pattern analysis and urban planning: adjusting traffic light cycles, adding public transportation lines, and widening or adding roads.

We compare three methods of processing Q2. The baseline method uses an index nested loop join, where the index is an R*-tree [2]. As described in Section IV-A, this is a similarity join using L-1 distance. Our distributions have 50 bins and hence a distribution maps to a 50-dimensional point. The first phase is the index nested loop join and the second phase is a post-processing that filters out false positives, since the hypercube we use to query the R*-tree may contain points that are outside of our bound. The second method is to first use our condensation scheme obtained as in Section IV-B to condense the distributions into four-dimensional points, in a preprocessing step. Then phase 1 is to use an index nested loop join on the four-dimensional points, while phase 2’s post-processing is to check the exact L-1 distance of the 50-dimensional points. The third method is identical to the second, except that we partition the 50 bins equally into four larger bins.

Figure 10(g) compares the execution times of the two phases. Due to “the curse of dimensionality”, phase 1 of the original base method is much slower than the other two methods. The phase 2 cost of our method is very close to that of the original method because lower dimensionality does not necessarily imply more false positives. The reason for this is that as dimensionality increases, the portion of area inside the hypercube where false positives reside also increases (at one extreme, the one dimensional “cube” is just a line segment and contains no false positive area). Equal partitioning, on the other hand, performs poorly in phase 2 because it disregards individual distributions and many distributions become identical after the condensation, thus producing a lot of false positives.

We then vary the parameters $k$ (the number of bins after condensation) and $\varepsilon$ (the $VD$ bound), with results in Figures 10(h) and 10(i). The execution time is the sum of both phases. As $k$ increases, the difference between the optimal condensation and the equal partition becomes smaller, causing their performance to approach each other. Moreover, as $\varepsilon$ increases, more tuple pairs are involved, increasing the costs for all methods. In Figure 10(j), we further compare the number of tuple pairs that are post-processed in phase 2 for all three methods. Also shown in the figure is the actual number of pairs satisfying the join. We can see that our condensation scheme results in slightly fewer false positives than the baseline method whereas equal partitioning produces significantly more false positives. We then examine the efficiency of our condensation algorithm. The result in Figure 10(k) shows that the dynamic programming algorithm is quite efficient and the condensation time is linear with table size.

Finally, we use the R-statistical package [29] to generate 500K points from a seed normal distribution $D_0$. We then treat each of these points as the center (i.e., $\mu$) of a normal distribution, whose $\sigma$ is chosen randomly from [25, 75]. These distributions are used for d-join. In Figure 10(l), we vary the $\sigma$ parameter of the seed distribution $D_0$ from 50 up to 500. We then measure the speed-up of condensed d-joins from the baseline algorithm which has no condensation. The speed-up our condensation scheme is invariably between 5 and 7, while the speed-up of equal partitioning increases with $\sigma$. This is because as $\sigma$ of $D_0$ increases, the distributions to be joined are more evenly spaced in the whole range, making equal partitioning closer to the optimal (achieved by our condensation scheme).

VI. RELATED WORK

There has been substantial work on uncertain data management lately (e.g., [10, 28, 24, 16, 17]). Cheng et al. [7] proposed probabilistic threshold join, which is the same as our v-join semantics. [7] proposes the concept of $x$-bounds, where $x$ is a constant probability number between 0 and 1. The idea is to have a series of $x$-bounds for some data structure (e.g., each data page), for various $x$ values between 0 and 1, and then try to prune the whole data structure during query processing to save costs. The dilemma here is that if the data structure is coarse, and if the probability distributions in the data structure vary in their value range, then the $x$-bounds are loose for each individual distribution and the pruning is not effective (e.g., when a page has relatively diverse distributions); but if the data structure is fine-grained (e.g., per distribution), then this approach has a considerable overhead in computing, storing, and loading those $x$-bounds for query processing.

Our work, on the other hand, enables the system to quickly compute probabilistic bounds dynamically, tailored to the query parameters (e.g., $\varepsilon$ and threshold $p$), based only on the first two moments of a distribution using probability theory. We propose an indexing scheme and a TwoEndZigzagJoin algorithm to further cut down I/O costs as well. Moreover, unlike [7], we also propose and study different semantics of
join (i.e., d-join) on both numerical and categorical uncertain data, which is useful for many applications.

Singh et al. in [26] propose ideas to index uncertain categorical data. Their semantics for join queries are also based on threshold and are similar to ours in that regard. Nonetheless, their work specifically targets categorical data. In contrast, the bulk of our work applies to numerical data, although d-join also applies to categorical data. Bohm et al. propose an index structure called Gauss-Tree that puts means and standard deviations of Gaussian distributions in an R-tree index [5]. The goal of their work is different from ours; they try to identify uncertain objects whose multidimensional features are modeled as Gaussian distributions, which is why they use R-trees. By contrast, we use a B+ tree based structure to facilitate value match. There is also a line of work that addresses indexing numerical uncertain data [8, 1]. However, the special query type for which those indexing techniques work is to find the set of all distributions whose random variables (i.e., attributes) fall in an interval with probability at least \( p \). There is no easy method to make such an indexing technique to work for the types of join queries that we study, which is the aim of our work.

Similarity join on data points in multidimensional space is well studied (e.g., [4, 25, 19]). The connection between this line of work and ours is due to the fact that we can reduce a d-join to a similarity join. However, when the dimensionality is high, with any existing technique, there is invariably a significant performance penalty. Our design of the condensed d-join and the optimal condensation scheme are a novel contribution. Dimensionality reduction is also studied for indexing time series databases (e.g., [18]). However, a salient difference in discrete probability distributions than time series features is the constraint that probability values are between 0 and 1 and sum to 1. We take advantage of this and devise a simple, efficient, and optimal condensation algorithm. Finally, band-join on deterministic data is also studied in the literature (e.g., [11]). V-join can deal with a variant of band-join on uncertain data, where the old techniques cannot be applied.

VII. CONCLUSIONS AND FUTURE WORK

We organize and classify the join semantics into v-join and d-join for uncertain data. Our main contribution is to propose algorithms that can efficiently process each type of join. For v-join, we devise the TwoEndZigzagJoin algorithm which is a combination of using probability theory and our novel indexing mechanism. This saves both CPU and I/O costs. We reduce a d-join to a similarity join and devise the condensed d-join algorithm and an optimal condensation scheme based on dynamic programming. Finally, our systematic empirical study based on real and synthetic datasets shows us the benefit of our various techniques. An open problem for future work is the join order selection and optimization for multi-relation joins.

ACKNOWLEDGMENT

This work was supported in part by the NSF, under the grant IIS-1017452. The author would like to thank Eli Upfal for helpful conversations, and the anonymous referees for their comments and suggestions.

REFERENCES


ACKNOWLEDGMENT

This work was supported in part by the NSF, under the grant IIS-1017452. The author would like to thank Eli Upfal for helpful conversations, and the anonymous referees for their comments and suggestions.

REFERENCES