Supporting Data Uncertainty in Array Databases

Liping Peng
School of Computer Science
University of Massachusetts Amherst, MA
lppeng@cs.umass.edu

Yanlei Diao
School of Computer Science
University of Massachusetts Amherst, MA
yanlei@cs.umass.edu

ABSTRACT

Uncertain data management has become a key issue in scientific applications. Recently, array databases have gained popularity for scientific data processing due to performance benefits. In this paper, we address uncertain data management in array databases, which may involve both *value uncertainty* within individual tuples and *position uncertainty* regarding where a tuple should belong in an array given uncertain dimension attributes. In our work, we define the formal semantics of array operations on uncertain data involving both types of uncertainty. To address the challenges raised by position uncertainty, we focus on two key array operations, *Subarray* and *Structure-Join*. We propose a number of storage and evaluation schemes for *Subarray*, with a focus on a scheme that bounds the overhead of querying by strategically placing a few replicas of the tuples with large variances. Built on this scheme, we propose two techniques for *Structure-Join* with or without using indexes. Evaluation results show that for common workloads, our best-performing techniques incur limited storage overheads and outperform baselines often by a wide margin.

1. INTRODUCTION

Uncertain data management has been studied intensively in areas such as sensor networks, information extraction, data cleaning, and business intelligence. Recently, uncertain data management has also started to play a key role in large-scale scientific applications such as severe weather monitoring [14, 25], computational astrophysics [15, 23], and asteroid threat detection [8]. In particular, recent studies [8, 22, 23] show that almost all scientific data are noisy and uncertain. Therefore, capturing uncertainty in data processing, from data input to query output, has become a key issue in scientific data management.

We next present two real-world applications that motivated our work. In the first application, *massive astrophysical surveys* such as the Sloan Digital Sky Survey (SDSS) [23] and the Large Synoptic Survey Telescope (LSST) [15] aim to generate observations of 10^8 stars and galaxies at nightly data rates of 0.5TB to 20TB. They are expected to enable real-time detection of transient events and anomalies as well as long-term tracking of objects of interest. However, the observations in digital sky surveys are inherently noisy as the objects can be too dim to be recognized in the captured images. SDSS and LSST make repeated observations of faint objects and derive continuous probability distributions for uncertain attributes, e.g., the location and luminosity of objects, in the data cooking process. For example, the Galaxy table in the SDSS astronomy archive has 297 attributes, out of which 151 attributes are uncertain and described by (multivariate) Gaussian distributions.

In the second domain of *severe weather monitoring*, the Research Center for Collaborative Adaptive Sensing of the Atmosphere (CASA) has developed distributed radar sensor networks for detecting hazardous weather events like tornados and severe storms [14]. The produced radar data are highly noisy due to environmental noise, electronic device noise, instability of transmit frequency, and quality issues of the antenna. Hence, recent work [9] has developed a data cooking process to produce continuous probability distributions for important meteorological measures such as wind velocity and reflexivity of each voxel of the air.

For supporting scientific applications, relational technology has proven useful in some applications like SDSS [23]. However, there is a recent realization that most scientific data naturally reside in multi-dimensional arrays rather than in relations. This is because most scientific data are produced to characterize physical phenomena that rely heavily on the notions of "adjacency" and "neighborhood" in a multi-dimensional space. As a result, array algebra and native array databases have recently been developed for scientific data processing [3, 8, 21]. Besides convenient expression of array operations, the new array implementation also offers significant performance benefits over relational database systems [19]. In particular, its chunk-based storage scheme enables better alignment of logical locality (i.e., objects close in the logical array) and physical locality (i.e., objects close to each other are likely to be stored in the same chunk). Since many array operations exploit logical locality of data, e.g., finding objects adjacent to a given location, the associated physical locality can lead to significant I/O savings.

The increasing popularity of array databases has significant implications on uncertain data management: Recent work on array databases [11, 13, 12] has considered the case that a tuple belongs to a specific cell of an array and some of its value attributes are uncertain, which is referred to as the "*value uncertainty*". On the other hand, a more complicated case arises when the attributes chosen to be the dimensions of an array are uncertain. For example, the x-y positions of an object in SDSS naturally serve as the dimensions of the array, but they are uncertain and characterized by a bivariate Gaussian distribution. As such, the uncertain location of an object can cause its tuple to belong to multiple cells in the array, referred to as the "*position uncertainty*". SciDB, a leading effort on array databases, has acknowledged this issue in real-world applications but leaves the solution to future work [21].

In this paper, we provide a thorough treatment to uncertain data management in array databases. We focus on continuous uncertain data because they are a natural fit for scientific data and harder to support than discrete uncertain data due to the difficulty in enumerating the possible values. In particular, we address two key questions: (i) What are the intended answers of array operations on uncertain data that may involve both position and value uncertainty? (ii) What are the storage and evaluation methods for efficiently processing array operations on continuous uncertain data? By way of addressing these, we make the following contributions:

1. We define the formal semantics of array operations on uncertain data involving both position and value uncertainty (§2). We

show that *Subarray* and *Structure-Join* are the two most important array operations that involve position uncertainty; many other array operations can be transformed into (one of) these two.

2. For *Subarray*, we provide native support for its operation on uncertain dimension attributes in the array database. We propose a number of storage and evaluation schemes to deal with position uncertainty. In particular, we focus on a novel scheme, called *store-multiple*, that bounds the overhead of querying, which can become very high given tuples with large variances, using modest replication of such tuples. In addition, *store-multiple* has the flexibility to configure the storage for best performance under different workloads due to the use of a cost model.

3. For *Structure-Join* on uncertain dimension attributes, we propose two techniques (§4): The first is to integrate existing indexes for relational databases [7, 6, 16] with the *store-multiple* scheme for array databases, and minimize the join cost by solving a set covering problem. We also propose a new subarray-based evaluation strategy for *Structure-Join*, which works without a pre-built index. This strategy employs tight conditions for running repeated subarray queries on the inner array of the join, as well as a cost model for configuring the storage for best performance.

4. We evaluate our techniques using both synthetic workloads and a case study of the Sloan Digital Sky Survey (SDSS) [23] (§5). For *Subarray*, *store-multiple* outperforms other alternatives by using a cost model to configure the storage and bounding the overhead of querying. For *Structure-Join*, the index-based join exhibits high overheads of index I/O and the set covering problem, while the subarray-based join outperforms it due to the use of tight conditions for probing the inner array and the accuracy of the cost model. Our case study shows that for realistic datasets, the storage overhead of *store-multiple* is rather limited, e.g., over 79% tuples have only 1 copy and over 92% tuples have at most 3 copies (considering that 3 is the common number for replication in big data systems). In addition, our best techniques for *Subarray* and *Structure-Join* are shown to outperform the baselines, often by a wide margin.

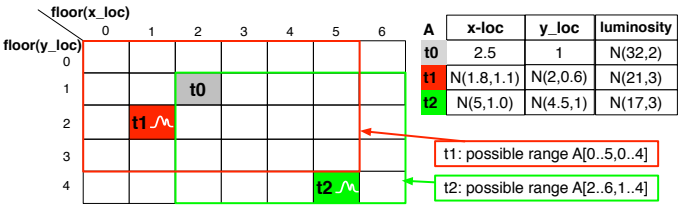
2. ARRAY MODEL AND ALGEBRA

In this section, we provide background on the array model and array algebra proposed recently [3, 20]. Furthermore, we extend the array model to accommodate uncertain data and formally define the semantics of array algebra under the uncertain data model.

2.1 Array Data Model

Background on the Array Model. An array database contains a collection of arrays. Each array is represented as $\mathbb{A}(\mathbf{D}^d; \mathbf{V}^m)$, where $\mathbf{D}^d$ denotes the d *dimension attributes* that define the array, and $\mathbf{V}^m$ denotes m *value attributes*. We sometimes also use the shorthand, $\mathbb{A}^d$, to denote a d-dimensional array. Consider an example in the Digital Sky Survey domain: $\mathbb{A}^2(x\_loc, y\_loc; luminosity, color, \ldots)$ defines a two-dimensional array using the dimension attributes (x_loc, y_loc). If a dimension attribute is discrete valued, the model requires a linear ordering of its values. If a dimension attribute is continuous valued instead, a user-defined function, e.g., $\lfloor x\_loc \rfloor$, is assumed to be available for discretizing the domain of the dimension attribute into an ordered set of values. These ordered values are used as the *index values* in a given dimension.

In an array $\mathbb{A}^d$, a unique combination of the index values of the d dimensions defines a cell, which can contain multiple tuples. Array cells are addressed by the index values of dimensions, e.g., a single cell addressed by $\mathbb{A}[\lfloor x\_loc \rfloor$=1, $\lfloor y\_loc \rfloor$=2], abbreviated as $\mathbb{A}[1, 2]$, or multiple cells by $\mathbb{A}[1 : \infty, 2 : 4]$. By default, tuples in a cell include both dimension attributes and value attributes. If a dimension attribute is discrete valued, its values in the tuples can be omitted because they are the same as the index values of the cell (which is not true for continuous attributes). Note that this model is an extension of the SciDB array model [3, 20] as it allows continuous attributes to be dimension attributes.



A	x-loc	y_loc	luminosity
t0	2.5	1	N(32,2)
t1	N(1.8,1.1)	N(2,0.6)	N(21,3)
t2	N(5,1.0)	N(4.5,1)	N(17,3)

Figure 1: Array $\mathbb{A}$ with dimension attributes, x_loc and y_loc, and the value attribute $luminosity$, all of which can be uncertain.

To draw a analogy with the relational model, we can translate an array to a relation $\mathbb{R}(D_1, \ldots, D_d, V_1, \ldots, V_m)$. That is, we treat dimension attributes as value attributes in tuples and store all the tuples in a table with no particular order. To address array cells given dimension values, we write explicit predicates like $\mathbb{A}[\lfloor x\_loc \rfloor = 1, \lfloor y\_loc \rfloor = 2]$.

An Array Model for Uncertain Data. We next extend the array model to accommodate uncertain data. When array data are uncertain, the dimension attributes can be uncertain (e.g., the x-y locations of a galaxy follow a bivariate Gaussian distribution); the value attributes can be uncertain (e.g., the luminosity of a galaxy follows a Gaussian); or both groups of attributes can be uncertain.

Uncertainty of value attributes, referred to as *value uncertainty*, is easy to support: we store a (joint) probability distribution of the uncertain value attributes, instead of fixed values, in each tuple. If there is uncertainty regarding the existence of a tuple, called *existence uncertainty*, we store the existence probability as a special value attribute in the tuple. Then we model value and existence uncertainties jointly using a mixed-type distribution [24], which states that the tuple exists with a certain probability, and if the tuple exists, its uncertain value attributes follow a joint distribution.

Uncertainty of dimension attributes is harder to support because a dimension attribute with multiple possible values can cause a tuple to belong to multiple cells in an array, referred to as *position uncertainty*. If we take the tuple's marginal distribution of each uncertain dimension attribute, we can estimate the *possible range* along that dimension where the tuple may belong. Suppose that a marginal distribution has mean μ and standard deviation σ, we can define the possible range to be $[\mu - k\sigma, \mu + k\sigma]$ with a sufficiently large k chosen based on Chebyshev's inequality or Gaussian properties, e.g., when $x \sim \mathrm{N}(\mu, \sigma)$, $\Pr(x \in [\mu - 3\sigma, \mu + 3\sigma]) > 0.99$.

In this work, we associate each tuple to a default position in the array, which is the cell indexed by the mean values of uncertain dimension attributes. (In implementation we may consider other options, which we discuss later.) Fig. 1 illustrates an array, $\mathbb{A}(x\_loc, y\_loc; luminosity)$, where continuous uncertain attributes, x_loc and y_loc, are dimension attributes, and the $floor$ function discretizes their values as index values. Tuple t_0 has fixed values for x_loc and y_loc and hence belongs to a single cell. Tuple t_1, however, has a bivariate Gaussian distribution. Therefore, although it is marked in its default cell, $\mathbb{A}[1, 2]$, with a significant probability it can reside in any cell in a possible range, $\mathbb{A}[0 : 5, 0 : 3]$, marked by the red box in the figure. Similarly, t_2 also has a possible range, $\mathbb{A}[2 : 6, 1 : 4]$, due to uncertain x_loc and y_loc.

2.2 Array Algebra

We next survey operators in array algebra and define their formal semantics under the uncertain data model. These operators were originally proposed in the Array Functional language (AFL) [18] where all attributes have deterministic values and dimension attributes must be discrete valued. In this work, we extend the seman-

tics of these operators to continuous-valued dimension attributes as well as uncertain data in both dimension and value attributes.

Value-based: The operators in the first category operate only on the value attributes of tuples. An example is *Filter*, which applies predicates to the value attributes of tuples stored in the array. Another example is *Apply*, which applies arithmetic operations to the values of tuples. A third example is *Project*, which projects out some value attributes from existing tuples. Since the above operators operate only on the value attributes of tuples, their semantics of uncertain data processing under the array model is the same as the semantics under the relational model; the semantics of the latter is already defined in previous work [24].

Structure-based: The operators in the second category operate on dimension attributes and optionally on value attributes as well. We examine several common operators:

(1) *Subarray* takes an array $\mathbb{A}$ and a condition θ on the dimension attributes, and returns a new array with the tuples that satisfy the condition θ. Revisit our example array. $Subarray(\mathbb{A}, 1.5 \leq x\_loc \leq 3.3 \text{ and } 2.1 \leq y\_loc \leq 4.8))$ will first retrieve tuples from the array block $\mathbb{A}[1:3, 2:4]$, and then filter those tuples based on the precise condition, $1.5 \leq x\_loc \leq 3.3$ and $2.1 \leq y\_loc \leq 4.8$. The output array always has the same dimensions as the input, but usually fewer cells and tuples. Subarray can be translated into selection in relational algebra, i.e., $Subarray(\mathbb{A}, \theta) \equiv \sigma_\theta(R_\mathbb{A})$, where $R_\mathbb{A}$ is the relational representation of the array.

When the dimension attributes addressed in the condition θ are uncertain, Subarray is semantically equivalent to selection on the uncertain dimension attributes in the relational setting. Hence, we have the following definition:

Definition 2.1 (Probabilistic Subarray) *Given an array $\mathbb{A}^d$, condition θ on uncertain dimension attributes, and a user-specified probability threshold $\lambda \in (0, 1)$, $Subarray(\mathbb{A}, \theta, \lambda)$ returns an array $\mathbb{B}^d$ where the cell $\mathbb{B}[i_1, \ldots, i_d]$ contains each tuple t from $\mathbb{A}[i_1, \ldots, i_d]$ that satisfies the condition θ with a probability at least λ, i.e., $\int_\theta f_t(\mathbf{x})d\mathbf{x} \geq \lambda$, where $f_t(\mathbf{x})$ is the tuple's probability density function on the the uncertain dimension attributes.*

Revisiting the above example, $Subarray(\mathbb{A}, 1.5 \leq x\_loc \leq 3.3$ and $2.1 \leq y\_loc \leq 4.8)$. When x_loc and y_loc are uncertain, we can no longer restrict the search to only the block $\mathbb{A}[1:3, 2:4]$. It is because tuples that belong to other cells, e.g., $\mathbb{A}[1, 5]$, may satisfy the $Subarray$ condition with a probability larger than λ. Based on the formal semantics, the entire array needs to be searched.

(2) *Structure-Join* (*SJoin*) in the array model takes as input an array $\mathbb{A}^d$, a second array $\mathbb{B}^d$ of the same dimensionality, and a join condition θ. $SJoin(\mathbb{A}, \mathbb{B}, \theta)$ returns an array $\mathbb{C}^{2d}$, where the cell $\mathbb{C}[i_1, \cdots, i_d, i_{d+1}, \cdots, i_{2d}]$ contains the result of θ-join between the tuples in $\mathbb{A}[i_1, \cdots, i_d]$ and the tuples in $\mathbb{B}[i_{d+1}, \cdots, i_{2d}]$. The equivalent expression in relational algebra is, $R_\mathbb{A} \bowtie_\theta R_\mathbb{B}$, where $R_\mathbb{A}$ and $R_\mathbb{B}$ are the relational representations of $\mathbb{A}$ and $\mathbb{B}$.

The join condition, θ, has a few common forms: (1) If the dimension attributes are discrete-valued, θ usually specifies equality comparison on the dimension attributes, as in the AFL proposal [18].[1] (2) If the dimension attributes are continuous-valued, equi-join is seldom used. Instead, θ takes a form of proximity join. A common form is linear proximity join, $|\mathbb{A}.d_i - \mathbb{B}.d_i| < \delta$ for each dimension attribute d_i. The join condition essentially defines a band region for each pair of join attributes. Another common form of proximity join uses Euclidean distance, $\sum_i (\mathbb{A}.d_i - \mathbb{B}.d_i)^2 < \delta^2$. As noted earlier, we focus on continuous uncertain data in this paper and hence proximity join in later technical sections.

Next we consider the case that the continuous dimension attributes of arrays $\mathbb{A}$ and $\mathbb{B}$ are uncertain. While the tuples have default positions in the array based on their mean values, they may belong to multiple cells with non-zero probabilities. In the face of position uncertainty, the join between $\mathbb{A}$ and $\mathbb{B}$ must return all pairs of tuples that satisfy the join condition θ with a significant probability. To do so, we leverage the semantics of cross-product in the above $SJoin$ definition, which involves pairing each cell in $\mathbb{A}$ with each cell in $\mathbb{B}$ and then pairing the tuples within those cells. More specifically, we define probabilistic structure-join as follows:

Definition 2.2 (Probabilistic Structure-Join) *Given arrays $\mathbb{A}^d$ and $\mathbb{B}^d$, a join condition θ, and a probability threshold λ, $SJoin(\mathbb{A}, \mathbb{B}, \theta, \lambda)$ returns an array $\mathbb{C}^{2d}$ where $\mathbb{C}[i_1, \cdots, i_d, i_{d+1}, \cdots, i_{2d}]$ contains the result of probabilistic θ-join, $\mathbb{A}[i_1, \cdots, i_d] \bowtie_{\theta,\lambda} \mathbb{B}[i_{d+1}, \cdots, i_{2d}]$ $= \{(t_1, t_2) | \, t_1 \in \mathbb{A}[i_1, \cdots, i_d], t_2 \in \mathbb{B}[i_{d+1}, \cdots, i_{2d}], \iint_\theta f_{t_1}(\mathbf{x})$ $\cdot f_{t_2}(\mathbf{y}) d\mathbf{x} d\mathbf{y} \geq \lambda\}$, where $f_{t_1}(\mathbf{x})$ and $f_{t_2}(\mathbf{y})$ are the probability density functions for t_1 and t_2, respectively.*

(3) *Regrid-Aggregation* partitions an input array into non-overlapping blocks, and for each block, applies an aggregate function to all the tuples in the block. The output array has one cell for each block which contains the aggregate value computed. Regrid can be viewed as repeated application of the Subarray operation to extract each block and then to compute the aggregate within each block.

When the dimension attributes are uncertain, one can use the *Probabilistic Subarray* operator to extract the tuples that belong to each block with non-zero probabilities (usually more than those that physically exist in the block). Note that even if a tuple belongs to a block with a small probability, if its aggregate attribute has a large value, it can still contribute a modest value, which is the product of its attribute value and existence probability, to the aggregate. Hence, the probability threshold for tuple existence in *Subarray* should be set to 0 in theory, or a small value in practice.

(4) *GroupBy-Aggregation* takes three arguments including an input array $\mathbb{A}^d$, a list of grouping dimensions $\mathbb{G}^{d_1}$, where $d_1 \leq d$, and an aggregate function. Again, it can be viewed as repeated application of *Subarray* to construct array blocks corresponding to the groups and then computing the aggregate within each block.

As shown in the above discussion, *Subarray* and *Structure-Join* are the two most important primitives in array algebra. Hence, we focus on efficient implementation of them under data uncertainty in subsequent sections.

We finally show two example queries written for the Sloan Digital Sky Survey (SDSS) [23], where the attributes, $rowc$ and $colc$, define a two dimensional array called Galaxy. These queries are written by following the convention of the AFL language, but with syntactic differences due to reasons such as the support of continuous dimension attributes. Query Q_1 computes the average brightness in a subarray region. Query Q_2 finds the regions in the sky space with the observation density greater than a threshold τ, where the observation density for a region is defined to be the sum of the number of observations within δ distance in either direction of each point in the region. To do so, the query first performs a self join of the Galaxy array based on the array structure, then groups the join output into 100 by 100 blocks of $G_1.rowc$ and $G_1.colc$, and finally counts the number of observations per block for density filtering.

Q1: AVG(SUBARRAY(Galaxy, $x_1 < rowc < x_2$ and $y_1 < colc < y_2$), $brightness$)

[1] In this case, the output array, $\mathbb{C} = SJoin(\mathbb{A}, \mathbb{B}, \theta)$, can be simplified to have the same dimensionality as $\mathbb{A}$ and $\mathbb{B}$, where each cell $\mathbb{C}[i_1, \ldots, i_d]$ contains the result of $\mathbb{A}[i_1, \ldots, i_d] \bowtie_\theta \mathbb{B}[i_1, \ldots, i_d]$. This definition is consistent with equi-join in relational algebra where only one copy of the common join attributes is retained.

Q2:FILTER(GROUPBY(SJOIN(Galaxy G_1, Galaxy G_2, $|G_1.rowc - G_2.rowc| < \delta$ and $|G_1.colc - G_2.colc| < \delta$), $\lfloor G_1.rowc/100 \rfloor$ and $\lfloor G_1.colc/100 \rfloor$, COUNT cnt), $cnt > \tau$)

3. NATIVE SUPPORT FOR SUBARRAY

In this section, we focus on the structural operation, *Subarray*. As defined previously, it defines a region based on the dimension attributes and returns tuples whose existence probability in the region exceeds a threshold. Since *Subarray* is equivalent to selection in relational algebra, there are two options for implementation:

The first option is to translate *Subarray* to selection in the relational setting. When the dimension attributes are uncertain, to avoid scanning all tuples in the database, existing work has built various indexes based on statistical quantities such as quantiles [4, 5] and moments [16] of tuple distributions. However, these indexes may not be effective when the filtering power is low and can trigger many random I/O's since they are often secondary indexes.

The second option is to build native support of *Subarray* in array databases where logical locality and physical locality are better-aligned. For instance, *Subarray* that exploits logical locality of data, e.g., a few adjacent array cells, may need to retrieve only a few relevant physical storage units called chunks. This effect of exploiting physical locality is similar to using a clustered primary index in relational databases, but without having to build the index.

Hence, in this work we focus on native support of array operations on uncertain data. Building such native support, however, is challenging due to the issue of *position uncertainty*: when the dimension attributes are uncertain, each tuple can belong to multiple cells with non-zero probabilities. The key question we address is how to reduce the complexity associated with such position uncertainty and maximize the benefits of locality in array databases.

3.1 Storage and Evaluation Schemes

Given a tuple and a probability distribution on its dimension attributes, the array cells that have non-zero probabilities to contain the tuple form the "*possible range*" of the tuple, as defined in Section 2.1. We next propose a few storage schemes that guarantee that the tuple can be observed when any cell of its possible range is covered in the query region.

Store-All: One solution is to store a copy of the tuple in each cell of the tuple's possible range. Fig. 2(a) depicts the storage of two tuples, t_1 and t_2, where t_1 is replicated in its possible range $\mathbb{A}$[0:5, 0:3] (including the red and yellow cells), and t_2 is replicated in $\mathbb{A}$[2:6, 1:4] (the green and yellow cells), with the overlap region marked in yellow. A query region, $\mathbb{A}$[2:2, 3:3], is marked by a solid blue box in Fig. 2(d). A major advantage of this scheme is that we can execute the query region directly on the array, without any missing results. The disadvantages include possibly excessive storage overheads and high I/O costs in querying because each logical cell may need many physical chunks to store the replicated tuples.

A similar storage scheme is *store-all with pointers*: store a tuple in its default position, i.e., the cell where the mean values of its dimension attributes reside, and add a pointer to this tuple in all other cells in the tuple's possible range. Thus we avoid repeated storage of all attributes of a tuple. However, the numerous pointers can still incur high storage overheads, as well as frequent random I/O's at query time as a result of chasing pointers from a given query region to fetch other relevant tuples stored outside the region.

Store-Mean: To avoid excessive storage overheads discussed above, we next consider storing a tuple only once based on the mean values of its dimension attributes. However, directly running *Subarray* on such storage will lead to missed results: tuples whose mean values are outside the query region but whose possible ranges overlap with the region will be missed. To avoid the problem, the query region must be expanded. One way to do so is to augment each cell with upper and lower bounds for each dimension, indicating the distance to travel along each dimension in order to find all tuples that could belong to that cell—we call these bounds the *upper and lower fences* for expanding the query region from this cell. This way, the storage overhead is limited to two integers per dimension per cell.

Fig. 2(b) shows the storage layout for tuples t_1 and t_2. Consider the cell $\mathbb{A}[2, 3]$. The fences for the x dimension, $(-1, 3)$, means that at query time, from this cell we need to walk one step to the left and three steps to the right, while the fences for the y dimension, $(-1, 1)$, indicates walking one step up and one step down in the array. After walking on both dimensions, we will reach cell $\mathbb{A}[1, 2]$ to retrieve tuple t_1 and cell $\mathbb{A}[5, 4]$ to retrieve t_2.

To generate fences, whenever a new tuple is inserted into a cell C in the array based on its mean value, we identify every cell in the tuple's possible range, compute its distance from the cell C, then expand its fences if they do not cover the computed distance. At query time, for each cell contained in the query region, we expand it using the upper and lower fences, and take the union of all these expansions to produce a complete *expanded query region*. Fig. 2(e) depicts the user-specified query region in a solid blue box and the expanded query region using a dashed blue box.

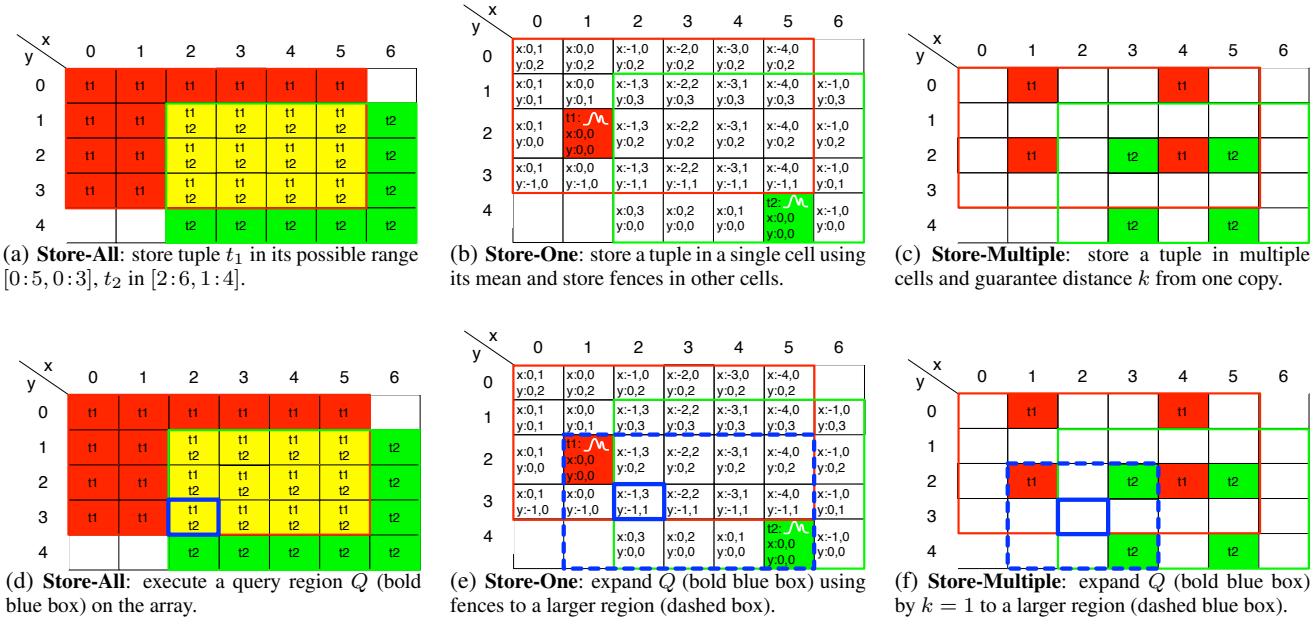
The advantage of this strategy is significant reduction of auxiliary information stored in each cell, i.e., two fences for each dimension, as opposed to *store-all* and *store-all with pointers*. However, a potential issue is that the expanded query region can grow very large, containing both relevant and irrelevant tuples, and incur both high I/O cost for fetching all the tuples and high CPU cost for validating them using the precise *Subarray* condition.

Store-Multiple: Finally, we propose a scheme that employs limited replication of tuples and guarantees that from any cell in a tuple's possible range, the query needs to be expanded by at most k cells (steps) along each dimension to find a copy of the tuple. We call k the **step size**, and use it to control both query expansion, by $2k$ on each dimension, and the degree of replication in storage, roughly inversely proportional to k. Interestingly, *Store-multiple* subsumes both *store-all* and *store-mean*: it becomes *store-all* when $k = 0$, and approximates *store-mean* (without fences) when k is big enough to cover the largest possible range among all tuples.

More importantly, *store-multiple* overcomes the shortcomings of the previous two schemes: First, it bounds the query expansion by k cells along each dimension. Such controlled expansion is particularly helpful when some tuples have large variances and hence large possible ranges. In other schemes, tuples of large variances will cause them to be replicated in numerous cells (*store-all*) or cause the query region to be expanded based on the largest tuple variance in a wide neighborhood (*store-mean*). Second, *store-multiple* offers the flexibility to configure the parameter k for different workloads to achieve best performance, as we shall show shortly.

Fig. 2(c) shows such storage with k=1, where tuple t_1 is stored in four cells and t_2 in another four cells. We can verify that for each cell in t_1's possible region (the red rectangle), we need to walk only one step in both dimensions to find a copy of t_1. The same guarantee holds for t_2. Fig. 2(f) shows a query region matching the cell $\mathbb{A}[2, 3]$, marked by the solid blue box, and the expanded region $\mathbb{A}$[1:3,2:4] using $k = 1$, marked by the dashed blue box.

The evaluation of *Subarray* under *store-multiple* includes two steps: (1) I/O step: The original query region is expanded by k cells along both directions on each dimension; all tuples in the expanded query region are read from disk. (2) CPU step: The exact existence probability in the query region is computed for each retrieved tuple based on its distribution and compared with the probability thresh-



(a) **Store-All**: store tuple t_1 in its possible range $[0:5, 0:3]$, t_2 in $[2:6, 1:4]$.

(b) **Store-One**: store a tuple in a single cell using its mean and store fences in other cells.

(c) **Store-Multiple**: store a tuple in multiple cells and guarantee distance k from one copy.

(d) **Store-All**: execute a query region Q (bold blue box) on the array.

(e) **Store-One**: expand Q (bold blue box) using fences to a larger region (dashed box).

(f) **Store-Multiple**: expand Q (bold blue box) by $k = 1$ to a larger region (dashed blue box).

Figure 2: Alternative storage and evaluation strategies for tuples with uncertain dimension attributes.

old. Since computing the exact existence probability for continuous random variables requires expensive integration, we optimize it by first running fast filters [16] with negligible costs to prune tuples of low existence probabilities, and building an in-memory hash table to avoid doing integration for different copies of the same tuple.

Two questions remain for *store-multiple*: First, the way to store tuples while guaranteeing the step size k in query expansion is not unique, leading to different degrees of replication of a tuple. How do we find the best layout of tuples under the step size k configuration? Second, given a dataset and typical query workloads, how do we choose the best configuration of k for optimal performance? We address these two issues in §3.2 and §3.3, respectively.

3.2 Tuple Layout under Store-Multiple

Consider the tuple layout in a d-dimensional array $\mathbb{A}^d$ stored using *store-multiple* with a step size configuration $\langle k_1, k_2, \cdots, k_d \rangle$. This means that from any cell in the tuple's possible range, walking k_i cells in both directions on the i-th dimension, for $1 \leq i \leq d$, guarantees to find a copy of the tuple. Finding the best way to store tuple copies amounts to a coverage problem, as we define below.

Definition 3.1 (Covering Cell) *Given a d-dimensional array $\mathbb{A}^d$ under store-multiple with a step size configuration $\langle k_1, k_2, \cdots, k_d \rangle$, the covering range of the walk from a cell $\mathbb{A}[x_1, x_2, \cdots, x_d]$ is $\mathbb{A}[x_1 - k_1 : x_1 + k_1, \cdots, x_d - k_d : x_d + k_d]$. We also say each cell in $\mathbb{A}[x_1 - k_1 : x_1 + k_1, \cdots, x_d - k_d : x_d + k_d]$ is "covered" by the cell $\mathbb{A}[x_1, x_2, \cdots, x_d]$.*

Definition 3.2 (Covering Set) *A set of cells $\boldsymbol{S}$ is covered by a set of cells $\boldsymbol{C}$ if and only if each cell in $\boldsymbol{S}$ is covered by at least one cell in $\boldsymbol{C}$. $\boldsymbol{C}$ is called the covering set of $\boldsymbol{S}$.*

Definition 3.3 (Problem of Tuple Copy Layout) *Given a tuple t, find the minimum covering set $\boldsymbol{C}$ of its possible range $\boldsymbol{S} = \mathbb{A}[l_1 : u_1, l_2 : u_2, \cdots, l_d : u_d]$ so that placing one copy of the tuple in each cell of $\boldsymbol{C}$ guarantees the correctness of query expansion using the step size $\langle k_1, k_2, \cdots, k_d \rangle$.*

We address the problem by first showing the lower bound of the size of a covering set, as shown in the following proposition.

Proposition 3.1 *Given an array $\mathbb{A}^d$ under store-multiple with a step size configuration $\langle k_1, k_2, \cdots, k_d \rangle$, if a tuple's possible range is $\boldsymbol{S} = \mathbb{A}[l_1 : u_1, l_2 : u_2, \cdots, l_d : u_d]$, the number of cells needed to cover $\boldsymbol{S}$ is at least $\prod_{i=1}^{d} (\lfloor (u_i - l_i)/(2k_i + 1) \rfloor + 1)$.*

PROOF. We can pick a subset of cells from the region $\boldsymbol{S} = \mathbb{A}[l_1 : u_1, l_2 : u_2, \cdots, l_d : u_d]$ as follows: $\boldsymbol{S'} = \{\mathbb{A}[x_1, x_2, \cdots, x_d] | \forall i \in \{1, 2, \ldots, d\}, x_i = l_i + p_i(2k_i + 1)$ and $l_i \leq x_i \leq u_i$, where $p_i \in \{0\} \cup \mathbb{N}\}$. Obviously, the size of the set of picked cells $|\boldsymbol{S'}|$ is $\prod_{i=1}^{d} (\lfloor (u_i - l_i)/(2k_i + 1) \rfloor + 1)$. Based on Definition 3.2, if we can prove that at least $|\boldsymbol{S'}|$ cells are already needed just to cover $\boldsymbol{S'}$ which is a subset of $\boldsymbol{S}$, it is also true that at least $|\boldsymbol{S'}|$ cells are needed to cover $\boldsymbol{S}$.

Let us assume a cell $\mathbb{A}[x_1, x_2, \cdots, x_d] \in \boldsymbol{S'}$ is covered by (the walk from) a cell $\mathbb{A}[y_1, y_2, \cdots, y_d]$. This means $y_i - k_i \leq x_i \leq y_i + k_i$ on any dimension i. For any cell $\mathbb{A}[x'_1, x'_2, \cdots, x'_d] \in \boldsymbol{S'} - \{\mathbb{A}[x_1, x_2, \cdots, x_d]\}$, there exists a dimension j such that $x_j \neq x'_j$. Without loss of generality, assume $x'_j = x_j + p_j(2k_j + 1)$ where $p_j \in \mathbb{N}$. Then $x'_j \geq y_j - k_j + p_j(2k_j + 1) > y_j + k_j$, which means $\mathbb{A}[x'_1, x'_2, \cdots, x'_d]$ does not fall in the covering range of $\mathbb{A}[y_1, y_2, \cdots, y_d]$. Therefore, no two cells in $\boldsymbol{S'}$ can be covered by the same cell. In other words, at least $|\boldsymbol{S'}|$ cells are needed in order to cover $\boldsymbol{S'}$. Then to cover $\boldsymbol{S}$, a superset of $\boldsymbol{S'}$, at least $|\boldsymbol{S'}| = \prod_{i=1}^{d} (\lfloor (u_i - l_i)/(2k_i + 1) \rfloor + 1)$ cells are needed as well. □

Given the lower bound on the size, we next consider how to distribute the covering set, i.e., the cells with tuple copies, to achieve this lower bound. To maximize the union of the covering ranges of those tuple copies, we can store them in evenly-spaced cells $\mathbb{A}[x_1, x_2, \cdots, x_d]$, where $x_i = l_i + k_i + p_i(2k_i + 1)$, for $p_i = 0, 1, \ldots, \lfloor (u_i - l_i)/(2k_i + 1) \rfloor$ and $i = 1, \ldots, d$. Basically, on the i-th dimension the first copy is stored at $l_i + k_i$ and the other copies are stored $2k_i$ cells away from each other. Fig. 3(a) shows such distribution of tuple copies in a two-dimensional array when $k_1 = k_2 = 2$. The tuple's possible range consists of all the cells within the solid boundary, requiring at least 9 copies to be placed. When each of the black cells stores one tuple copy, we use exactly 9 copies. However, three copies are stored outside the tuple's possible range, which increases the chance of reading irrelevant copies when a query region falls outside the tuple's possible range.

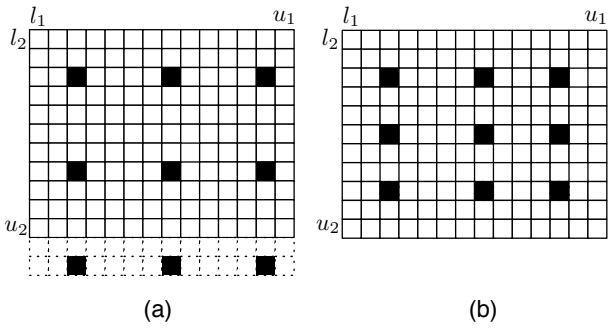


Figure 3: Distribution of tuple copies in a 2-dimensional array using Store-Multiple (step sizes $k_1 = k_2 = 2$)

symbol	description
T	number of tuples
b	number of bytes per tuple
pr_i	length of a tuple's possible range on the i-th dimension
d	dimensionality of an array
c	chunk size (the I/O unit) in bytes
s_i	length of each cell on the i-th dimension
n_i	number of cells on the i-th dimension
q_i	query region size on the i-th dimension
k_i	step size on the i-th dimension

Table 1: Notation in modeling and analysis.

It is thus desirable to store all copies of a tuple inside its possible range. Our solution is that when a tuple needs only one copy on the i-th dimension, we store it at the center of its possible range, i.e., $\lfloor (l_i + u_i)/2 \rfloor$; when it needs more than one copy, we store the first copy at $l_i + k_i$, the last copy at $u_i - k_i$, and the others (if any) are evenly spaced in between, as shown in Fig. 3(b). Thus we still use the minimum number of copies to cover the tuple's possible range.

3.3 Cost Model of Subarray under Store-Multiple

We next propose a cost model for *Subarray* under the *store-multiple* scheme and use the model to find the optimal step size configuration. The symbols used in the model are summarized in Table 1. Like in SciDB [3], a cell is a logical unit in an array while a chunk is a physical storage unit as well as the I/O unit; tuples in a logical cell can be stored in one or multiple chunks. For *Subarray* evaluation under *store-multiple*, as explained in §3.1, the I/O cost consists of the seek and transfer time of chunks in the expanded query region, while the CPU cost is the product of the number of tuples to be validated and the validation cost per tuple. For simplicity, we assume that the centers of tuples' possible ranges are evenly spread over the whole array. We also begin by assuming that all tuples' possible ranges have the same size, pr_i, on the i-th dimension. Our model can be extended to support possible ranges of variable sizes, as we explain at the end of the section.

I/O Cost: To capture I/O cost, we focus on a key factor, the number of chunks in the expanded query region.

Let us first compute the number of cells with which a tuple's possible range overlaps on the i-th dimension. Obviously this depends on the alignment of the possible range and the cells along this dimension, as shown in Fig. 4. We can chop the possible range into $\lceil pr_i/s_i \rceil$ segments, where the first $\lceil pr_i/s_i \rceil - 1$ segments have length s_i and the last segment has length $r = pr_i - (\lceil pr_i/s_i \rceil - 1)\, s_i$. Depending on the starting position of the possible range in the first cell, it can overlap with different numbers of cells: when the starting position is in $[0, s_i - r]$, it overlaps with $\lceil pr_i/s_i \rceil$ cells; when the starting position is in $(s_i - r, s_i)$, it overlaps with $\lceil pr_i/s_i \rceil + 1$ cells. Then the expected number of cells the possible range overlaps with is

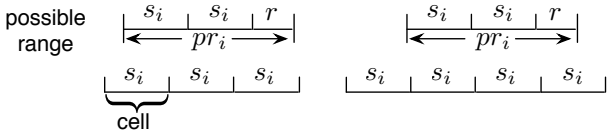


Figure 4: Illustration of the number of cells that a possible range overlaps with

$$\frac{s_i - r}{s_i} \left\lceil \frac{pr_i}{s_i} \right\rceil + \frac{r}{s_i} \left(\left\lceil \frac{pr_i}{s_i} \right\rceil + 1 \right) = \frac{pr_i}{s_i} + 1 \qquad (1)$$

Calculated in a similar way, the number of cells that overlap with the query region Q on the i-th dimension is $q_i/s_i + 1$, and the number for the expanded query region $\tilde{Q}$ is $q_i/s_i + 1 + 2k_i$.

We next model the number of chunks in the expanded query region $\tilde{Q}$. It is the product of the number of cells in $\tilde{Q}$ and the average number of chunks per cell. Combining Equation (1) with Proposition 3.1, we write $u_i - l_i + 1 = pr_i/s_i + 1$, and derive the number of copies per tuple as:

$$\prod_{i=1}^{d} \left(\left\lfloor \frac{pr_i/s_i}{2k_i + 1} \right\rfloor + 1 \right). \qquad (2)$$

The average number of chunks per cell is the total number of tuple copies in the array divided first by the number of cells in the array and then by the number of tuples a chunk can hold, i.e., $\lfloor c/b \rfloor$:

$$T \prod_{i=1}^{d} \left(\left\lfloor \frac{pr_i/s_i}{2k_i + 1} \right\rfloor + 1 \right) \Big/ \prod_{i=1}^{d} n_i \Big/ \lfloor c/b \rfloor . \qquad (3)$$

Multiplying this with the number of cells in $\tilde{Q}$, $\prod_{i=1}^{d}(q_i/s_i + 1 + 2k_i)$, we get the number of chunks in $\tilde{Q}$:

$$T \prod_{i=1}^{d} \left(\left\lfloor \frac{pr_i/s_i}{2k_i + 1} \right\rfloor + 1 \right) \Big/ \prod_{i=1}^{d} n_i \Big/ \lfloor c/b \rfloor \cdot \prod_{i=1}^{d} \left(\frac{q_i}{s_i} + 1 + 2k_i \right) \qquad (4)$$

CPU Cost: To capture CPU cost, we model the number of tuples to be validated. Given an expanded query region $\tilde{Q}$, a tuple is retrieved for validation as long as it has one copy stored in $\tilde{Q}$. Fig. 5 shows an example tuple whose possible range is the red box. The tuple has a copy in $\tilde{Q}$ but its center of the possible range, marked by a black dot, lies outside $\tilde{Q}$. To handle such tuples, let us define the *validation region*, V, to be the set of cells where the centers of the possible ranges of to-be-validated tuples reside, and model the size of V first. Now consider the i-th dimension of the array: (1) When k_i is large enough that every tuple only needs one copy to cover its possible range, V is simply the expanded query region $\tilde{Q}$, of size $q_i/s_i + 1 + 2k_i$. (2) When k_i is smaller so that all tuples have more than one copy, tuples that have at least one copy stored in $\tilde{Q}$ need to be validated. It can be derived that in this case V on the i-th dimension is of size $q_i/s_i + 1 + pr_i/s_i$. Summarizing the above two cases and multiplying the size of V with the number of tuples per cell, we have the number of tuples to be validated as:

$$T \Big/ \prod_{i=1}^{d} n_i \cdot \prod_{i=1}^{d} \left(\frac{q_i}{s_i} + 1 + z_i \right), \qquad (5)$$

where $z_i = 2k_i$ when $pr_i/s_i < 2k_i + 1$ and $z_i = pr_i/s_i$ otherwise.

Finally we combine the I/O and CPU costs by plugging in unit cost measurements, including the seek and transfer time per chunk and per tuple validation time.

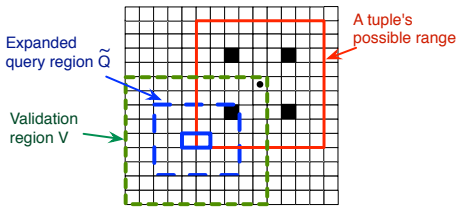


Figure 5: Illustration of the validation region.

A Generalized Model. We next relax the assumption that all tuples have the same possible range size. When we have different possible range sizes, we can group tuples based on the possible range size. The runtime of a query will be a weighted sum of runtime over each group of tuples, where the number of tuples per group serves as the weight. In practice, we can build statistics of the possible range size when a batch of tuples comes in. For instance, SDSS [23] updates the scanned image of the sky on a nightly basis and can build the statistics as a nightly observation is being produced. If domain knowledge reveals the distribution of the possible range size does not change drastically from day to day, we can also re-use statistics collected in the past.

Implementation. Given the cost model and basic statistics of tuples' possible range sizes and query sizes, we can extend the data loading routine of an array database, such as that of SciDB, as follows: given a wide range of step size configurations, we estimate query costs, including both CPU and I/O costs, from different step size configurations, and choose the configuration that offers the best estimated performance. We evaluate the effectiveness of our model for doing so in §5. Once the step size is configured, as we scan each tuple from the original data file, its possible ranges can be obtained and its copies can be distributed as shown by Fig. 3(b).

Our implementation of *store-multiple* can be directly used to implement *store-all* by setting $k_i = 0$ $(i = 1, \ldots, d)$. It can also be revised to implement *store-mean* with only modest changes: we can store one copy at the tuple's mean position and other copies as before. In this way, we introduce at most one more copy per tuple. The mean copies of all tuples are stored in a separate file. This revised implementation can also support evaluation algorithms that use only the mean copy of each tuple, ignoring other copies.

4. SUPPORT FOR STRUCTURE JOIN

In this section, we focus on another common array operator, *Structure-Join* under position uncertainty. The default evaluation strategy, as stated in Definition 2.2, creates all pairs of tuples from the two input arrays and evaluates an integral for each pair of tuples, which is prohibitively expensive. To improve performance, we propose to integrate existing indexes for relational databases [7, 6, 16] with the *store-multiple* scheme for array databases and derive an index-based technique (in §4.1). We further propose a new subarray-based evaluation strategy, as well as model-based optimization to achieve the best performance of this strategy (in §4.2).

In our discussion below, we focus on linear proximity join, that is, $SJoin(\mathbb{A}^d, \mathbb{B}^d, \theta, \lambda)$ where $\theta = \bigwedge_{i=1}^{d} |\mathbb{A}.d_i - \mathbb{B}.d_i| < \delta$, which is the most common in scientific applications. Non-linear proximity join based on Euclidean distance, e.g., $\sum_i (\mathbb{A}.d_i - \mathbb{B}.d_i)^2 < \delta^2$, can be first relaxed to linear proximity join, and then followed by additional filtering using exact integration based on θ. Moreover, we define the selectivity of a probabilistic $SJoin$ to be $\frac{|\mathbb{A}| \cdot |\mathbb{B}|}{|\mathbb{C}|}$, where $|\mathbb{A}|$, $|\mathbb{B}|$, $|\mathbb{C}|$ are tuple counts of inputs $\mathbb{A}$ and $\mathbb{B}$, and the output $\mathbb{C}$.

4.1 Index-based Join

Recent work has proposed new indexes on continuous uncertain data in relational databases to improve query performance [7, 6, 16]. A natural way to use indexes in *Structure-Join* is to perform

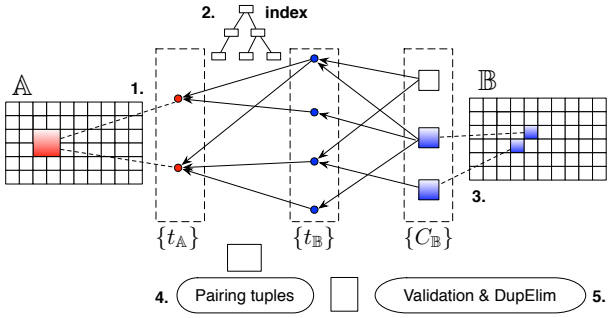


Figure 6: Illustration of index-based join.

index nested loops join (INLJ): the outer relation is scanned once; for each tuple in the outer relation, the index on the inner relation is probed to find the candidates.

While index nested loops join (INLJ) is a standard join method, integrating it with the *store-multiple* scheme proposed previously for handling uncertain data raises a new question: Since tuples may have multiple copies stored in the array database, how can we minimize the chance of producing duplicate results from multiple copies of the same tuple, hence the associated I/O and CPU costs?

Our main idea is as follows: Consider $SJoin(\mathbb{A}^d, \mathbb{B}^d, \theta, \lambda)$ where both $\mathbb{A}$ (the outer) and $\mathbb{B}$ (the inner) are stored using *store-multiple*. Assume that there is a pre-built index on the continuous uncertain join attribute(s) in $\mathbb{B}$, and the leaf nodes in the index store the possible range of each tuple. As shown in Fig. 6, the array $\mathbb{A}$ is scanned once, by reading one block (with one or multiple cells) at a time. Each tuple, $t_{\mathbb{A}}$, in the current block of $\mathbb{A}$ triggers an index lookup to find its candidates from $\mathbb{B}$, $\{t_{\mathbb{B}}\}$, as depicted by the left mapping structure in Fig. 6. Each candidate tuple $t_{\mathbb{B}}$ may be stored in multiple cells, $\{C_{\mathbb{B}}\}$, as depicted by the right mapping structure; these $C_{\mathbb{B}}$ cells can be computed from $t_{\mathbb{B}}$'s possible range stored in the index. After processing all tuples in the current block of $\mathbb{A}$, two mapping structures are complete. We want to read a subset of the $\mathbb{B}$ cells to minimize I/O while guaranteeing *coverage*: every candidate $\mathbb{B}$ tuple resides in at least one of those cells. Then the $\mathbb{A}$ tuples and their candidates $\mathbb{B}$ tuples are paired and validated if the index returns a superset of matches.

Given the two mapping structures, deciding which $\mathbb{B}$ cells to read to minimize I/O while guaranteeing coverage amounts to a set covering problem (SCP). Denote the union of candidates for all probing tuples in the current block as U. Each cell $C_{\mathbb{B}}$ from $\mathbb{B}$ stores a subset of U. Using the number of chunks as the weight of $C_{\mathbb{B}}$, denoted by $\omega(C_{\mathbb{B}})$, our problem is a variant of the classic SCP:

$$\begin{aligned} \text{minimize} \quad & \sum_{C_{\mathbb{B}} \in \mathbb{B}} \omega(C_{\mathbb{B}}) \cdot I_{C_{\mathbb{B}}} \\ \text{subject to} \quad & \sum_{C_{\mathbb{B}} : t_{\mathbb{B}} \in C_{\mathbb{B}}} I_{C_{\mathbb{B}}} \geq 1 \qquad \text{for all } t_{\mathbb{B}} \in U \\ & I_{C_{\mathbb{B}}} \in \{0, 1\} \qquad \text{for all } C_{\mathbb{B}} \in \mathbb{B} \end{aligned}$$

This is an NP-hard problem but can be solved efficiently by many techniques, e.g., the integer linear program and greedy algorithm.

This algorithm, which we call index-based join (IBJ), is affected by several factors in performance. We analyze them briefly below and evaluate them empirically in §5.

(1) *Selectivity*: The performance of IBJ highly depends on the filtering power of the index for a given workload. For example, when the selectivity is close to 1, almost all tuple pairs from the two input arrays exist in the output, so the index has little filtering power. This will cause the mapping structures to grow so large that the memory for the $\mathbb{A}$ block and mappings is consumed fast and may not even be enough to hold the mappings for a single tuple

$t_{\mathbb{A}}$. On the other hand, when the chosen index has great filtering power for a selective *Structure-Join*, the mapping structure is quite small and solving SCP is also quick. Then the overheads of probing indexes, building mappings, and solving SCP are outweighed by the savings of CPU and I/O costs.

(2) *Memory allocation*: The IBJ algorithm uses several data structures. The memory is shared among the read block of $\mathbb{A}$, the two mapping structures, the index, and the cache of $\mathbb{B}$ cells (or chunks). Since the sizes of both mappings increase with the size of the $\mathbb{A}$ block, these three data structures share a memory quota and then the block size can be automatically determined: the algorithm keeps adding a new $\mathbb{A}$ cell until this memory quota is used up. The index and the cache of $\mathbb{B}$ have their own memory quotas. These three parts compete for memory: if more memory is given to the $\mathbb{A}$ block and the mappings, fewer blocks are needed and SCP is more effective in reducing the I/O of reading $\mathbb{B}$; the more memory is given to the index, the lower I/O cost in index lookups; and the more is given to cache $\mathbb{B}$ cells, the lower I/O cost in reading $\mathbb{B}$ cells.

(3) *Storage schemes*: Although IBJ generally works for input arrays with any step size configuration, its performance is better shown when the outer array uses a large step size so that each tuple has fewer copies and thus the chance of duplicate index lookups for the same tuple is reduced. It turns out IBJ also prefers a large step size for the inner to minimize I/O cost. We provide a detailed explanation in evaluation in §5.

4.2 Subarray-based Join

The index-based join requires pre-built indexes, which may not always be available, and can consume excessive memory due to the use of the tuple-level mapping. We next present a new evaluation strategy of *Structure-Join*, called subarray-based join (SBJ).

Similar to block nested loops joins, *Structure-Join* can be naturally transformed into iterative *Subarray* operations on the inner array, for each block of the outer array. Assume that the smaller array, $\mathbb{A}$, is the outer of the join. For each cell $C_{\mathbb{A}}$, we do the following: (1) Load it into memory, form a subarray condition $\theta_{C_{\mathbb{A}}}$ on the inner array $\mathbb{B}$ by considering both the original join predicate and the storage scheme of $\mathbb{A}$, and run the *Subarray* query on $\mathbb{B}$. (2) Pair tuples in $C_{\mathbb{A}}$ with those tuples retrieved by *Subarray* on $\mathbb{B}$. (3) The final phase is validation that computes the exact probability for each tuple pair $(t_{\mathbb{A}}, t_{\mathbb{B}})$ to satisfy the join condition and compares it with the probability threshold λ. We show how to determine the subarray condition $\theta_{C_{\mathbb{A}}}$ in §4.2.1 and present a complete algorithm and its cost model in §4.2.2.

4.2.1 Subarray Condition for Each Outer Cell

The subarray condition $\theta_{C_{\mathbb{A}}}$ for each outer cell $C_{\mathbb{A}}$ must produce all join results while being as tight as possible to ensure good performance. Below we propose several necessary conditions for linear proximity join that guarantee no missing result in the join output.

Given a tuple $t_{\mathbb{A}}$, let $(l_{t_{\mathbb{A}}.d_i}, u_{t_{\mathbb{A}}.d_i})$ denote the lower and upper bounds of its possible range on the i-th dimension. Similarly, we have $(l_{t_{\mathbb{B}}.d_i}, u_{t_{\mathbb{B}}.d_i})$ for tuple $t_{\mathbb{B}}$. Then we have:

Proposition 4.1 *For any tuple pair $(t_{\mathbb{A}}, t_{\mathbb{B}})$ returned by $SJoin(\mathbb{A}^d, \mathbb{B}^d, \bigwedge_{i=1}^{d} |\mathbb{A}.d_i - \mathbb{B}.d_i| < \delta, \lambda)$, the intervals $(l_{t_{\mathbb{A}}.d_i} - \delta,\ u_{t_{\mathbb{A}}.d_i} + \delta)$ and $(l_{t_{\mathbb{B}}.d_i}, u_{t_{\mathbb{B}}.d_i})$ overlap on any dimension i $(i = 1, \ldots, d)$.*

PROOF. We prove by contradiction. Consider a tuple pair $(t_{\mathbb{A}}, t_{\mathbb{B}})$ returned by $SJoin$. Assume that there exists a dimension d_i where $(l_{t_{\mathbb{A}}.d_i} - \delta, u_{t_{\mathbb{A}}.d_i} + \delta)$ and $(l_{t_{\mathbb{B}}.d_i}, u_{t_{\mathbb{B}}.d_i})$ do not overlap, i.e., $l_{t_{\mathbb{A}}.d_i} - \delta > u_{t_{\mathbb{B}}.d_i}$ or $u_{t_{\mathbb{A}}.d_i} + \delta < l_{t_{\mathbb{B}}.d_i}$. Without loss of generality, let us assume $l_{t_{\mathbb{A}}.d_i} - \delta > u_{t_{\mathbb{B}}.d_i}$. Below we focus on computing probability $p = \iint_{\theta} f_{t_{\mathbb{A}}}(\mathbf{x}) f_{t_{\mathbb{B}}}(\mathbf{y}) \mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y}$ where the integration domain θ is $\{(\mathbf{x}, \mathbf{y}) | \bigwedge_{i=1}^{d} |x_i - y_i| < \delta\}$.

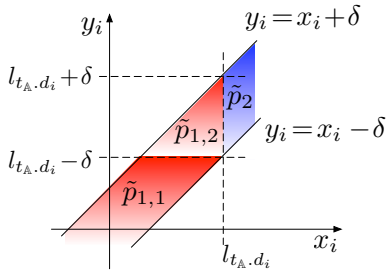


Figure 7: Illurstraion of $\tilde{p} = \tilde{p}_{1,1} + \tilde{p}_{1,2} + \tilde{p}_2$.

We start with finding an upper bound of p. Relaxing the join condition by only considering dimension d_i, we have

$$p < \iint\limits_{|x_i - y_i| < \delta} f_{t_{\mathbb{A}}}(\mathbf{x}) f_{t_{\mathbb{B}}}(\mathbf{y}) \mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y} = \iint\limits_{|x_i - y_i| < \delta} f_{t_{\mathbb{A}}.d_i}(x_i) f_{t_{\mathbb{B}}.d_i}(y_i) \mathrm{d}x_i \mathrm{d}y_i.$$

It means the probability for $(t_{\mathbb{A}}, t_{\mathbb{B}})$ to satisfy the join predicate is upper bounded by the probability for their values on dimension d_i to satisfy the join predicate on dimension d_i, denoted as $\tilde{p}$. The integration domain is colored in Fig. 7 and partitioned into three parts. Denote the probability mass of each partition as $\tilde{p}_{1,1}$, $\tilde{p}_{1,2}$ and $\tilde{p}_2$ respectively. Below we derive the upper bound for each of them by applying the assumption.

$$\begin{aligned}
\tilde{p}_{1,1} &= \int_{-\infty}^{l_{t_{\mathbb{A}}.d_i} - \delta} f_{t_{\mathbb{B}}.d_i}(y_i) \left(\int_{y_i - \delta}^{y_i + \delta} f_{t_{\mathbb{A}}.d_i}(x_i) \mathrm{d}x_i \right) \mathrm{d}y_i \\
&< \int_{-\infty}^{l_{t_{\mathbb{A}}.d_i} - \delta} f_{t_{\mathbb{B}}.d_i}(y_i) \left(\int_{-\infty}^{l_{t_{\mathbb{A}}.d_i}} f_{t_{\mathbb{A}}.d_i}(x_i) \mathrm{d}x_i \right) \mathrm{d}y_i \\
&= \frac{\epsilon}{2} \int_{-\infty}^{l_{t_{\mathbb{A}}.d_i} - \delta} f_{t_{\mathbb{B}}.d_i}(y_i) \mathrm{d}y_i < \frac{\epsilon}{2} \\
\tilde{p}_{1,2} &= \int_{l_{t_{\mathbb{A}}.d_i} - \delta}^{l_{t_{\mathbb{A}}.d_i} + \delta} f_{t_{\mathbb{B}}.d_i}(y_i) \left(\int_{y_i - \delta}^{l_{t_{\mathbb{A}}.d_i}} f_{t_{\mathbb{A}}.d_i}(x_i) \mathrm{d}x_i \right) \mathrm{d}y_i \\
&< \int_{u_{t_{\mathbb{B}}.d_i}}^{+\infty} f_{t_{\mathbb{B}}.d_i}(y_i) \left(\int_{-\infty}^{l_{t_{\mathbb{A}}.d_i}} f_{t_{\mathbb{A}}.d_i}(x_i) \mathrm{d}x_i \right) \mathrm{d}y_i \\
&= \frac{\epsilon}{2} \int_{u_{t_{\mathbb{B}}.d_i}}^{+\infty} f_{t_{\mathbb{B}}.d_i}(y_i) \mathrm{d}y_i = \frac{\epsilon}{2} \cdot \frac{\epsilon}{2} = \frac{\epsilon^2}{4} \\
\tilde{p}_2 &= \int_{l_{t_{\mathbb{A}}.d_i}}^{+\infty} f_{t_{\mathbb{A}}.d_i}(x_i) \left(\int_{x_i - \delta}^{x_i + \delta} f_{t_{\mathbb{B}}.d_i}(y_i) \right) \mathrm{d}x_i \\
&< \int_{l_{t_{\mathbb{A}}.d_i}}^{+\infty} f_{t_{\mathbb{A}}.d_i}(x_i) \left(\int_{u_{t_{\mathbb{B}}.d_i}}^{+\infty} f_{t_{\mathbb{B}}.d_i}(y_i) \right) \mathrm{d}x_i \\
&= \frac{\epsilon}{2} \int_{l_{t_{\mathbb{A}}.d_i}}^{+\infty} f_{t_{\mathbb{A}}.d_i}(x_i) \mathrm{d}x_i < \frac{\epsilon}{2}
\end{aligned}$$

Finally we have $p < \tilde{p} = \tilde{p}_{1,1} + \tilde{p}_{1,2} + \tilde{p}_2 < \epsilon + \epsilon^2/4 \ll \lambda$, which means $(t_{\mathbb{A}}, t_{\mathbb{B}})$ can never be in the join result. Then we reach a contradiction and thus the assumption is wrong. □

The proposition states a way to find a superset of the join answers: for each tuple $t_{\mathbb{A}}$ from $\mathbb{A}$, expand its possible range by δ on each dimension, denoted by $I_{t_{\mathbb{A}}}$, then pair $t_{\mathbb{A}}$ with all tuples $t_{\mathbb{B}}$ from $\mathbb{B}$ whose possible ranges overlap with $I_{t_{\mathbb{A}}}$.

When $\mathbb{A}$ is stored using *store-mean*, we use the above result to form a subarray condition on $\mathbb{B}$, for each cell $C_{\mathbb{A}} \in \mathbb{A}$. The next proposition shows how to do so, i.e., by relaxing the condition using the minimum lower bound and maximum upper bound of possible ranges of all tuples in $C_{\mathbb{A}}$.

Proposition 4.2 (Subarray for Store-mean) *Consider* $SJoin(\mathbb{A}^d, \mathbb{B}^d, \bigwedge_{i=1}^{d}|\mathbb{A}.d_i - \mathbb{B}.d_i| < \delta, \lambda)$ *when* $\mathbb{A}$ *is under store-mean. For a cell* $C_{\mathbb{A}}$*, a subarray condition* $\theta_{C_{\mathbb{A}}}$ *that returns all join results is:*

$$\bigwedge_{i=1}^{d} \min_{t_{\mathbb{A}} \in C_{\mathbb{A}}} l_{t_{\mathbb{A}}.d_i} - \delta < \mathbb{B}.d_i < \max_{t_{\mathbb{A}} \in C_{\mathbb{A}}} u_{t_{\mathbb{A}}.d_i} + \delta.$$

When $\mathbb{A}$ is stored using *store-multiple*, we do not need to relax the join condition as aggressively, e.g., to accommodate large possible ranges of some tuples. Instead, we can bound the relaxation using the step size of $\mathbb{A}$ and δ. Given the step size $\langle k_1, k_2, \cdots, k_d \rangle$ of array $\mathbb{A}$, we define some notation:

- Let the value range of cell $C_{\mathbb{A}}$ on dimension d_i be $(l_{C_{\mathbb{A}}.d_i}, u_{C_{\mathbb{A}}.d_i})$.
- For any cell $C_{\mathbb{A}} = \mathbb{A}[x_1, \ldots, x_d]$, two cells bound the expansion from $C_{\mathbb{A}}$ by the step size of $\mathbb{A}$, denoted as $C_{\mathbb{A}}^- = \mathbb{A}[x_1 - k_1, \ldots, x_d - k_d]$ and $C_{\mathbb{A}}^+ = \mathbb{A}[x_1 + k_1, \ldots, x_d + k_d]$.

Proposition 4.3 (Subarray for Store-multiple) *Consider* $SJoin(\mathbb{A}^d, \mathbb{B}^d, \bigwedge_{i=1}^{d}|\mathbb{A}.d_i - \mathbb{B}.d_i| < \delta, \lambda)$ *when* $\mathbb{A}$ *is under store-multiple. For cell* $C_{\mathbb{A}}$*, a subarray condition* $\theta_{C_{\mathbb{A}}}$ *that returns all join results is:*

$$\bigwedge_{i=1}^{d} l_{C_{\mathbb{A}}^-.d_i} - \delta < \mathbb{B}.d_i < u_{C_{\mathbb{A}}^+.d_i} + \delta.$$

PROOF. Let $S_{t_{\mathbb{A}}}$ denote the set of cells that store a copy of $t_{\mathbb{A}}$, i.e., $S_{t_{\mathbb{A}}} = \{C_{\mathbb{A}} | t_{\mathbb{A}} \in C_{\mathbb{A}}\}$. Below we first prove that $(l_{t_{\mathbb{A}}.d_i}, u_{t_{\mathbb{A}}.d_i}) \subseteq \bigcup_{C_{\mathbb{A}} \in S_{t_{\mathbb{A}}}} \left(l_{C_{\mathbb{A}}^-.d_i}, u_{C_{\mathbb{A}}^+.d_i}\right)$: When $t_{\mathbb{A}}$ only needs one copy to cover its possible range on dimension d_i, assume the copy is stored at $C_{\mathbb{A}}$, then $(l_{t_{\mathbb{A}}.d_i}, u_{t_{\mathbb{A}}.d_i}) \subseteq \left(l_{C_{\mathbb{A}}^-.d_i}, u_{C_{\mathbb{A}}^+.d_i}\right)$ because otherwise it needs at least two copies; When $t_{\mathbb{A}}$ has more than one copies on dimension d_i, according to §3.2, the first copy and the last copy are stored k_i cells away from the lower and upper bounds of $t_{\mathbb{A}}$'s possible range respectively, depicted by Fig. 3(b). So $l_{t_{\mathbb{A}}.d_i} = \min_{C_{\mathbb{A}} \in S_{t_{\mathbb{A}}}} l_{C_{\mathbb{A}}^-.d_i}$ and $u_{t_{\mathbb{A}}.d_i} = \max_{C_{\mathbb{A}} \in S_{t_{\mathbb{A}}}} u_{C_{\mathbb{A}}^+.d_i}$, which means $(l_{t_{\mathbb{A}}.d_i}, u_{t_{\mathbb{A}}.d_i}) = \bigcup_{C_{\mathbb{A}} \in S_{t_{\mathbb{A}}}} \left(l_{C_{\mathbb{A}}^-.d_i}, u_{C_{\mathbb{A}}^+.d_i}\right)$. Combining the two cases, we have $(l_{t_{\mathbb{A}}.d_i}, u_{t_{\mathbb{A}}.d_i}) \subseteq \bigcup_{C_{\mathbb{A}} \in S_{t_{\mathbb{A}}}} \left(l_{C_{\mathbb{A}}^-.d_i}, u_{C_{\mathbb{A}}^+.d_i}\right)$. Then for any tuple $t_{\mathbb{B}}$, if its possible range $(l_{t_{\mathbb{B}}.d_i}, u_{t_{\mathbb{B}}.d_i})$ overlaps with $(l_{t_{\mathbb{A}}.d_i} - \delta, u_{t_{\mathbb{A}}.d_i} + \delta)$, which is a necessary condition for $t_{\mathbb{B}}$ being a true match of $t_{\mathbb{A}}$ according to Proposition 4.1, it must also overlap with $\bigcup_{C_{\mathbb{A}} \in S_{t_{\mathbb{A}}}} \left(l_{C_{\mathbb{A}}^-.d_i} - \delta, u_{C_{\mathbb{A}}^+.d_i} + \delta\right)$. This means that $t_{\mathbb{B}}$ will be returned by at least one of the subarray queries formed for all cells in $S_{t_{\mathbb{A}}}$, say $Subarray(\mathbb{B}, \theta_{C_{\mathbb{A}_0}}, \lambda)$. In this way, we guarantee that no result is missing. □

This proposition states that for each cell $C_{\mathbb{A}}$, the subarray condition on the inner array $\mathbb{B}$ can be formed by expanding $C_{\mathbb{A}}$ first by the step size of $\mathbb{A}$ and then by δ.

4.2.2 Algorithm and Cost Model

The detailed algorithm of subarray-based join (SBJ) is illustrated in Fig. 8. It processes one block of the outer at a time (marked as Step 1 in Fig. 8, with a red block followed by a green block of $\mathbb{A}$). For each cell $C_{\mathbb{A}}$ in the current block, the algorithm forms a *Subarray* query and runs it on the inner array $\mathbb{B}$ (Step 2). We call the $\mathbb{B}$ cells returned by the *Subarray* query for each $C_{\mathbb{A}}$ the **candidate cells** of $C_{\mathbb{A}}$. Since the candidate cells of different outer cells may overlap, as an optimization to save I/O, the algorithm maintains the

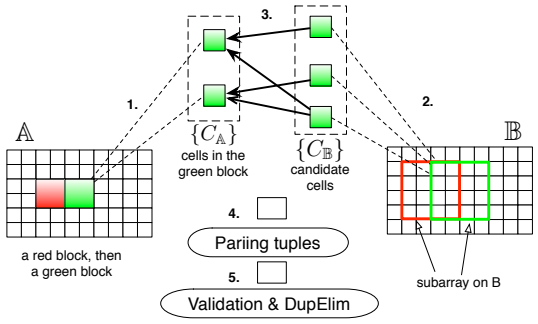


Figure 8: Illustration of subarray-based joins

union of the candidate cells of all outer cells in the current block, in $\{C_{\mathbb{B}}\}$ in Fig. 8. To avoid nonviable pairs of tuples, the algorithm maintains a hash map that maps a cell $C_{\mathbb{B}}$ to only those $\mathbb{A}$ cells whose candidate cells include $C_{\mathbb{B}}$, i.e., the mapping structure in Fig. 8 (Step 3). Then the algorithm reads relevant cells of $\mathbb{B}$ and pairs tuples accordingly (Step 4). It is optional to apply quick filters to the paired tuples to reduce later CPU cost. It finally does validation using the join condition and removes duplicates (Step 5).

As shown in Fig. 8, the memory is shared by (1) the read block of $\mathbb{A}$, (2) the cell-level mapping structure, which is much more compact than the tuple-level mapping in the index-based join, and (3) the cache of $\mathbb{B}$ cells (or chunks). Since first two items grow together, they share a memory quota. Then the block size is determined by loading more cells from $\mathbb{A}$ until the memory quota is used up. The rest of the memory is given to the cache of $\mathbb{B}$. With constrained memory, items (1) and (2) should be given as much memory as possible. Intuitively, the more memory is given to (1) and (2), the fewer batches and fewer duplicate reads of the same $\mathbb{B}$ cell due to the overlapping subarray regions on $\mathbb{B}$.

Next we build a cost model for SBJ under the *store-multiple* scheme which can be used to find the optimal step size during data loading given basic data statistics. We use the symbols in Table 1 with subscripts to distinguish inner and outer arrays.

I/O cost: We model the numbers of $\mathbb{A}$ and $\mathbb{B}$ chunks read in I/O and later translate them to seek and transfer times. First consider the outer array $\mathbb{A}$, which is read exactly once. Its number of chunks, denoted by $||\mathbb{A}||$, is the total number of tuple copies, denoted by $|\mathbb{A}|$, divided by the number of tuple copies per chunk. Based on Eq. (2) in §3.3, we have:

$$|\mathbb{A}| = T_{\mathbb{A}} \prod_{i=1}^{d} \left(\left\lfloor \frac{pr_{\mathbb{A},i}/s_{\mathbb{A},i}}{2k_{\mathbb{A},i}+1} \right\rfloor + 1 \right), \; ||\mathbb{A}|| = |\mathbb{A}| / \lfloor c/b_{\mathbb{A}} \rfloor .$$

Now consider the inner array $\mathbb{B}$. Each cell in $\mathbb{B}$ may be read multiple times as it can exist in the results of *Subarray* queries formed from different $\mathbb{A}$ blocks. Hence, the I/O cost for reading $\mathbb{B}$ is the product of (1) the number of $\mathbb{A}$ blocks, $\alpha_{R_{\mathbb{A}}}$, (2) the number of $\mathbb{B}$ cells to read per $\mathbb{A}$ block, denoted by $\beta_{R_{\mathbb{A}}}$, and (3) the number of chunks per $\mathbb{B}$ cell, $||C_{\mathbb{B}}||$. Below we model each of them in order.

We first model $\alpha_{R_{\mathbb{A}}}$. Assume that a memory quota of K chunks is given to the $\mathbb{A}$ block and its mapping. Then the number of cells in each $\mathbb{A}$ block, $n_{R_{\mathbb{A}}}$, is $K/(||C_{\mathbb{A}}|| + ||\mathcal{M}_{C_{\mathbb{A}}}||)$, where $||C_{\mathbb{A}}||$ is the number of chunks per $\mathbb{A}$ cell and $||\mathcal{M}_{C_{\mathbb{A}}}||$ is the number of chunks for the mapping entries per $\mathbb{A}$ cell. It is easy to see that

$$||C_{\mathbb{A}}|| = ||\mathbb{A}|| \Big/ \prod_{i=1}^{d} n_{\mathbb{A},i}.$$

According to Proposition 4.3, the subarray condition formed for cell $C_{\mathbb{A}}$ expands $C_{\mathbb{A}}$ by $\mathbb{A}$'s step size and then by δ, so the length of the *Subarray* query on dimension d_i is $(1 + 2k_{\mathbb{A},i})s_{\mathbb{A},i} + 2\delta$. It amounts to $\left((1 + 2k_{\mathbb{A},i})s_{\mathbb{A},i} + 2\delta\right) / s_{\mathbb{B},i} + 1$ cells in the $\mathbb{B}$ array

according to Eq. (1). When running this query on $\mathbb{B}$, the number of candidate cells of $C_{\mathbb{A}}$ is:

$$\beta_{C_{\mathbb{A}}} = \prod_{i=1}^{d} \left(\frac{(1 + 2k_{\mathbb{A},i})s_{\mathbb{A},i} + 2\delta}{s_{\mathbb{B},i}} + 1 + 2k_{\mathbb{B},i} \right) \quad (6)$$

Assuming that each mapping entry has b_{map} bytes, we have:

$$||\mathcal{M}_{C_{\mathbb{A}}}|| = \beta_{C_{\mathbb{A}}} \cdot \frac{b_{map}}{c}.$$

We then get the number of $\mathbb{A}$ blocks as the total number of cells divided by the number of cells in each $R_{\mathbb{A}}$ block:

$$\alpha_{R_{\mathbb{A}}} = \frac{\prod_{i=1}^{d} n_{\mathbb{A},i}}{n_{R_{\mathbb{A}}}} = \frac{(||C_{\mathbb{A}}|| + ||\mathcal{M}_{C_{\mathbb{A}}}||) \prod_{i=1}^{d} n_{\mathbb{A},i}}{K}$$

We next model the second factor, $\beta_{R_{\mathbb{A}}}$. For the current read block $R_{\mathbb{A}}$, we take the union of $\mathbb{B}$ cells returned by the *Subarray* query formed for each $\mathbb{A}$ cell. This union is equivalent to the set of $\mathbb{B}$ cells returned by a single *Subarray* query formed for the entire read block $R_{\mathbb{A}}$. Hence, similar to Eq. (6), we can get $\beta_{R_{\mathbb{A}}}$ as follows:

$$\beta_{R_{\mathbb{A}}} = \prod_{i=1}^{d} \left(\frac{(n_{R_{\mathbb{A}}}^{\frac{1}{d}} + 2k_{\mathbb{A},i})s_{\mathbb{A},i} + 2\delta}{s_{\mathbb{B},i}} + 1 + 2k_{\mathbb{B},i} \right).$$

We can get the last factor $||C_{\mathbb{B}}||$ in the same way as $||C_{\mathbb{A}}||$.

CPU cost: The CPU cost is the product of the number of tuple pairs to be validated, which we will model below, and the validation cost per tuple pair. According to our algorithm, tuples in each cell $C_{\mathbb{A}}$ are paired with the tuples in $C_{\mathbb{A}}$'s candidate cells and all such tuple pairs need to be validated. Therefore, the number of tuple pairs is the product of (1) the number of tuple copies in $\mathbb{A}$, (2) the number of candidate cells per $\mathbb{A}$ cell, and (3) the number of tuple copies per $\mathbb{B}$ cell. Using Eq. (6), we compute the product as:

$$|\mathbb{A}| \cdot \prod_{i=1}^{d} \left(\frac{(1 + 2k_{\mathbb{A},i})s_{\mathbb{A},i} + 2\delta}{s_{\mathbb{B},i}} + 1 + 2k_{\mathbb{B},i} \right) \cdot \frac{|\mathbb{B}|}{\prod_{i=1}^{d} n_{\mathbb{B},i}}$$

Finally, the combined IO and CPU model allows us to find the optimal step sizes for both inner and outer arrays if proximity join is the key workload. Basic statistics needed are the distribution of tuples' possible ranges and the common distance values in proximity join. Collecting such statistics is a common task of the query optimizer and we can leverage a large body of work on relational DBMSs in our setting.

5. EXPERIMENTS

In this section, we evaluate our techniques for *Subarray* and *Structure-Join* intensively using both a wide range of synthetic workloads with controlled properties and a case study of the Sloan Digital Sky Survey (SDSS) [23].

5.1 Evaluation of Subarray

Experimental Setup. We first generate synthetic workloads using parameters in Table 2. We consider 2D arrays where each dimension includes 100 cells of size 1. The dataset contains 500,000 tuples. In a tuple, each dimension attribute is described by a Gaussian distribution, (μ, σ); without loss of generality, we consider the tuple's possible range on this dimension to be $\mu \pm 3\sigma$. In the array, the μ values determine where the centers of tuples' possible ranges are located. We generate these values using the distribution, D_μ, which is set to a uniform distribution over the domain (0,100) or a Gaussian distribution with more tuples clustered at the center. The σ values determine how wide tuples' possible ranges are: they are generated from the distribution D_σ, which is a Zipfian distribution

	Parameter	Default Value	Other Values
Data	D_μ, distribution of μ	$\mathcal{U}(0, 100)$	$\mathcal{N}(50, 50/3)$
	D_σ, distribution of σ	$zipf(1)$	$zipf(0)$ - $zipf(3)$
Query	q, query range / domain size	10%	30%, 50%
	λ, probability threshold	0.9	0.01

Table 2: Parameters in $Subarray$ experiments.

on a set of values $\{1/12, 2/12, \ldots, 400/12\}$ with a configurable skewness parameter α. This way, tuples' possible ranges can vary from a single cell, to multiple cells, to the entire array, and with α=0 they are all equally likely whereas a larger α produces more tuples with smaller possible ranges. We generate a dataset for each combination of D_μ and D_σ configurations.

To generate queries, we vary the query size, q, between 10%, 30% and 50% of the domain for each dimension. The probability threshold, λ, prunes tuples based on the existence probability in the query region. In the common case, the user wants only the tuples with high existence probabilities; we use λ=0.9 to represent this workload. However, if the query includes an aggregate after $Subarray$, a tuple that has a low existence probability, p, in the query region but a large value, v, of the aggregate attribute may contribute a modest value, $p \cdot v$, to the aggregate. Hence, knowing the domain of v, it is only safe to prune tuples with low existence probabilities; we use λ=0.01 to represent this workload.

Expt 1: Cost Breakdown. Our *store-multiple* scheme has a configurable parameter, *step size* k, which determines both the degree of replication and query expansion. We start by showing how the $Subarray$ processing cost changes as k varies. We first consider the default workload, (D_μ=$\mathcal{U}$, D_σ=$zipf(1)$; q=10%, λ=0.9), or abbreviated as ($\mathcal{U}, zipf(1); 10\%, 0.9$). Fig. 9(a) shows two trends:

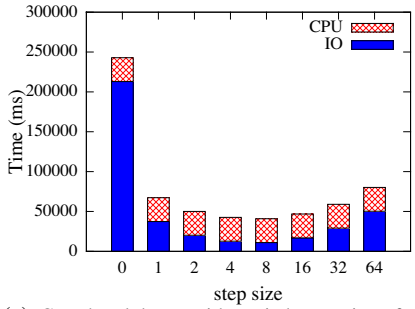
(1) *The I/O cost first decreases and then increases with the step size.* I/O is determined by both the number of cells in the expanded query region and the number of chunks per cell. When k is small, which means more aggressive replication of tuples, the expanded query region is small, but the number of chunks per cell is large and has a stronger impact on I/O. As k grows larger, fewer tuples are replicated, so each cell is smaller. But the expanded query region becomes very wide and affects I/O cost more. So the optimal I/O cost appears in the middle of the spectrum of k.

(2) *The CPU cost does not change with the step size when the probability threshold λ is high.* The CPU cost depends on the number of tuples that passed the quick filter and need to be validated using expensive integration. Fig. 9(c) shows the number of tuples that pass the filter. When λ is sufficiently high, ≥ 0.1 in this figure, the filter can drop most irrelevant tuples, so the number of tuples after filtering does not change with the step size, or the number of tuples retrieved from storage. We also examined the filter's effect using datasets beyond the default setting of the parameters in Table 2. Our observation of the filtering power is consistent.

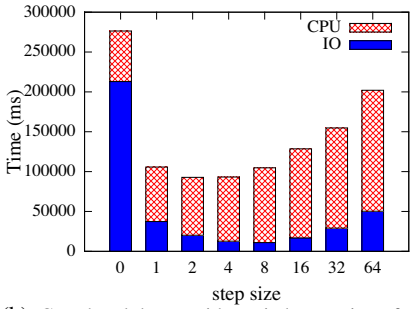
We next consider a new workload with $\lambda = 0.01$ and show the CPU and I/O costs in Fig. 9(b). The I/O cost is the same as Fig. 9(a) because it is not effected by λ. The CPU cost now grows with k because (1) a larger step size means a larger expanded query region and more tuples retrieved from storage, and (2) the filter is not able to drop many tuples given such a small threshold, illustrated by the blue line in Fig. 9(c). Combing both CPU and I/O costs, the optimal step size also appears in the middle of its spectrum.

Expt 2: Optimal Step Size. To show the challenge of finding the optimal step size, we next study how the optimal step size changes with the workload parameters in Table 2.

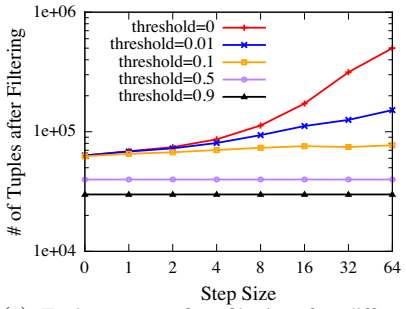
We first examine the effect of the skewness of the distribution of σ. Fig. 9(d) shows three lines for different datasets, $zipf(0)$, $zipf(1)$ and $zipf(2)$, when $\lambda = 0.9$. We can observe that *the optimal step size decreases with the skewness of the distribution of*



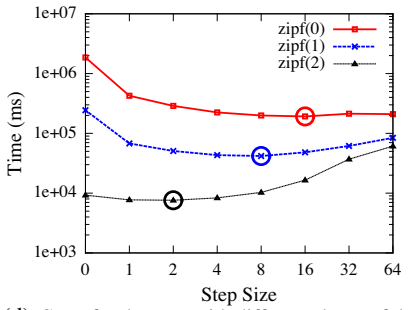
(a) Cost breakdown with varied step sizes for workload $(\mathcal{U}, zipf(1); 10\%, 0.9)$



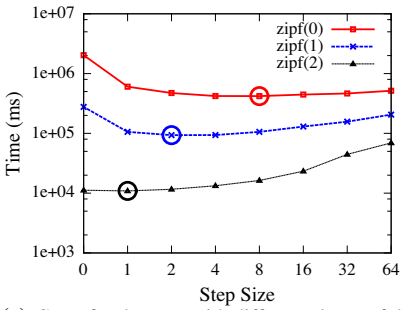
(b) Cost breakdown with varied step sizes for workload $(\mathcal{U}, zipf(1); 10\%, 0.01)$



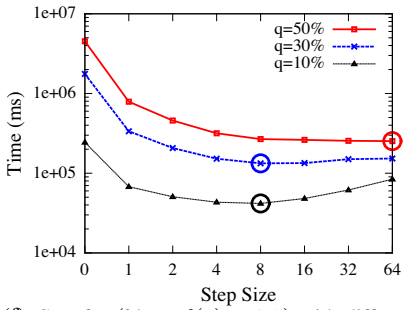
(c) Tuple counts after filtering for different threshold λ values $(\mathcal{U}, zipf(1); 10\%, \lambda)$



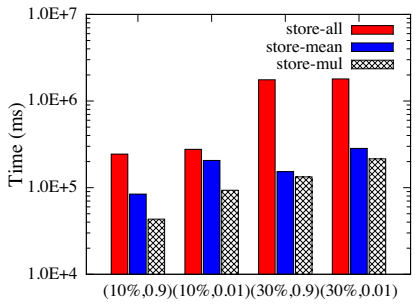
(d) Costs for datasets with different skews of the σ distribution $(\mathcal{U}, zipf(\cdot); 10\%, 0.9)$



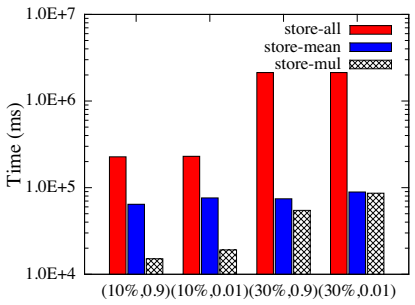
(e) Costs for datasets with different skews of the σ distribution $(\mathcal{U}, zipf(\cdot); 10\%, 0.01)$



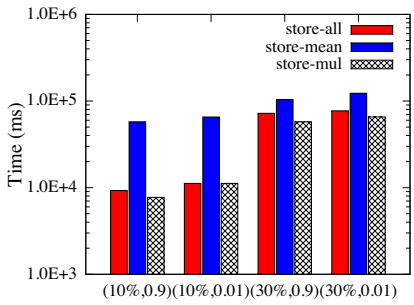
(f) Cost for $(\mathcal{U}, zipf(1); q, 0.9)$ with different query sizes



(g) Compare *store-all*, *store-mean*, and *store-multiple* for dataset $(\mathcal{U}, zipf(1))$



(h) Compare *store-all*, *store-mean*, and *store-multiple* for dataset $(\mathcal{N}, zipf(1))$



(i) Compare *store-all*, *store-mean*, and *store-multiple* for dataset $(\mathcal{U}, zipf(2))$

Figure 9: $Subarray$ **on two-dimensional synthetic datasets.**

σ. Recall that the CPU cost is the same for all step size configurations when λ is sufficiently high, and the I/O cost is minimized at a particular step size, before which the number of chunks per cell affects the I/O more than the expanded query size but after which it is the converse. Given increased skewness α, more tuples have small possible ranges, so the limited tuple replication keeps the number of chunks per cell small, hence its impact on I/O. We observe the same trend where we change the probability threshold, λ, to 0.01, as shown by Fig. 9(e).

We next demonstrate the effect of probability threshold λ. Comparing lines of the same color in Fig. 9(d) and Fig. 9(e), it can be seen that *the optimal step size shifts left when λ is very small*. The reason is that I/O cost does not change for different λ values, but the CPU cost grows with the step size when λ is very small because the filters are not as effective, as shown by Fig. 9(c).

We finally study the effect of the query region size. Fig. 9(f) shows that *the optimal step size increases with the query region size*. Recall that the I/O cost is affected by both the number of chunks per cell, which only depends on the step size k, and the expanded query size, which depends on both k and the query size q. The bigger q, the less sensitive the expanded query size is to the change of k. Therefore when q is big, a bigger step size leads to less number of chunks per cell without being penalized much by the bigger expanded query region it brings, hence less I/O. CPU is the same for all values of k when $\lambda = 0.9$. Therefore, the optimal step size is bigger than that for a small query. We made the same observation when $\lambda = 0.01$, with the optimal step size under each query size shifting left, as explained earlier.

Expt 3: Model Accuracy. We now use the cost model in §3.3 to determine the step size to be used when loading data into an array. We assume that the user can provide basic statistics including the σ distribution in the data and common $Subarray$ sizes. We denote the optimal step size $\mathbf{k}^*$, and the step size returned by our model $\tilde{\mathbf{k}}$. We measure the performance lose of our model, $\left(Cost(\tilde{\mathbf{k}}) - Cost(\mathbf{k}^*)\right)/Cost(\mathbf{k}^*)$. When tuples' mean values, μ, are normally distributed around the center of the array, the center of the query region matters as the data density varies. For such datasets, we pick 9 query centers, with one at (50,50) and others evenly scattered over the array, and report on the average of 9 runs.

The results for different datasets for query size $q = 10\%$ and $q = 50\%$ are shown in Table 3. In most cases, our model returns the optimal step size. Even when it does not, the performance lose is within 3.5% when the tuples' μ values are uniformly distributed. When the μ values are normally distributed, instead, the model assumes an even distribution of tuple's mean, and hence can be off sometimes, but the overall performance loss is within 11%. But we can still use the model as a heuristic because we observe the following when looking at the result for each query center: (1) The model is likely to give an overestimated step size for the sparse region,

D_μ	D_σ	q	λ	optimal step size	model step size	perf. lose
$\mathcal{U}(0,100)$	$zipf(0)$	10%	0.9	[16,16]	[16,16]	**0%**
			0.01	[8,8]	[8,8]	**0%**
		50%	0.9	[64,64]	[64,64]	**0%**
			0.01	[64,64]	[64,64]	**0%**
	$zipf(1)$	10%	0.9	[8,8]	[4,4]	**3.5%**
			0.01	[2,2]	[2,2]	**0%**
		50%	0.9	[64,64]	[64,64]	**0%**
			0.01	[8,8]	[8,8]	**0%**
	$zipf(2)$	10%	0.9	[2,2]	[1,1]	**1.51%**
			0.01	[1,1]	[0,0]	**3.41%**
		50%	0.9	[2,2]	[1,1]	**3.37%**
			0.01	[2,2]	[1,1]	**1.06%**
$\mathcal{N}(50,50/3)$	$zipf(0)$	10%	0.9	[16,16]	[16,16]	**0%**
			0.01	[8,8]	[8,8]	**0%**
		50%	0.9	[64,64]	[64,64]	**0%**
			0.01	[64,64]	[64,64]	**0%**
	$zipf(1)$	10%	0.9	[8,8]	[4,4]	**9.89%**
			0.01	[8,8]	[4,4]	**0.96%**
		50%	0.9	[64,64]	[64,64]	**0%**
			0.01	[32,32]	[16,16]	**0.79%**
	$zipf(2)$	10%	0.9	[1,1]	[1,1]	**0%**
			0.01	[1,1]	[1,1]	**0%**
		50%	0.9	[4,4]	[1,1]	**10.88%**
			0.01	[4,4]	[1,1]	**9.03%**

Table 3: Subarray model accuracy.

because decreasing the step size in the sparse region only further fills the current chunk rather than requesting more chunks but can result in a smaller expanded query region. (2) The model is likely to return an underestimated step size for the dense region. This is because, although increasing the step size enlarges the expanded query region, it drastically decreases the number of chunks per cell due to the dense inherence.

Expt 4: Comparison among Schemes. We now use the step size returned by the model to configure *store-multiple* and compare it to *store-all* and *store-mean* with fences for $Subarray$ evaluation. The results are shown in a log scale in Fig. 9(g)-9(i), each for a unique data workload. Each plot shows 4 queries, one for each combination of a small or big query region and a high or low threshold.

In all cases, *store-multiple* works the best. In comparison, *store-all* often incurs tremendous storage overheads and I/O costs in querying, as shown in Fig. 9(g) and 9(h). The only exception is when most tuples' possible ranges are small enough to fit in a single cell, as shown for the dataset in Fig. 9(i): *store-all* then incurs only a small storage overhead for the few tuples that have large variances. Moreover, *store-multiple* outperforms *store-mean* considerably when the query region q is relatively small, e.g., q=10%, which is the common case, due to a more constrained expanded query region. When q grows larger, e.g., q=30%, the difference between *store-multiple* and *store-mean* is reduced because the optimal step size also tends to be larger, meaning that infrequent replication of tuples works just fine if q is large. As such, most tuples have only one copy under *store-multiple*, hence similar to *store-mean*.

5.2 Evaluation of Structure-Join

Experimental Setup. We generate synthetic workloads for *Structure-Join* as before with a few changes: For efficiency, we first consider 1D arrays of 1000 cells and datasets of 100,000 tuples. We will show results on two-dimensional arrays and larger datasets in the case study. We consider $SJoin\,(\mathbb{A}_1, \mathbb{A}_2, |\mathbb{A}_1.x - \mathbb{A}_2.x| < \delta, \lambda)$, where $\mathbb{A}_1$ and $\mathbb{A}_2$ are loaded from the same dataset, and we fix δ to 1% of the domain for proximity join. The memory size is about 176 chunks, i.e., 30% of the data size. We use a state-of-the-art index on continuous uncertain data [16] whenever possible, e.g., in the Index-based Join and as an in-memory filter in Subarray-based Join (detailed later). This index returns only true matches on 1D arrays, so validation is not needed.

Expt 5: Index-Based Join (IBJ). We first study the Index-based Join, which is sensitive to: (1) the selectivity, which we control using the probability threshold λ, (2) the storage scheme, and (3) the memory allocation scheme.

We first use λ=0.9 or 0.01 to represent high or low selectivity. IBJ failed on all datasets when λ=0.01 regardless of memory allocation and storage schemes, because a single tuple can have so many matches that its mapping entries do not fit in memory. Hence, we consider only λ=0.9 below.

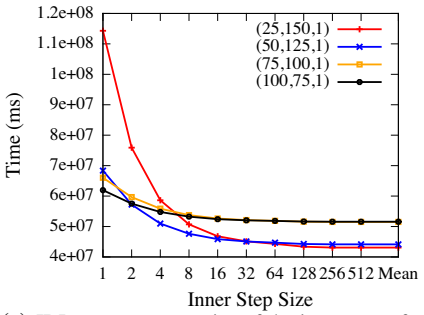
We next consider the effect of the storage scheme. As stated in §3.3, when array $\mathbb{A}$ is under *store-multiple*, it can still be processed as *store-mean*. Our question is whether IBJ works better by considering multiple tuple copies or only the copy at mean (ignoring other copies). We observe that for the outer array, it is always better to use only the copy at mean because it avoids duplicate index lookups for the same outer tuple. For the inner array, Fig. 10(a) shows the performance of *store-multiple* with different k values and that of *store-mean* on $zipf(1)$ dataset, where each line denotes a different memory allocation scheme (discussed shortly). As is seen, IBJ has the best performance when the inner array uses *store-mean* under all memory allocation schemes. The reason is that after the index lookups return all candidate tuples, fetching all of them from the inner array requires more I/O when nonviable tuples have a copy stored in the to-read cells (chunks), which is less likely to happen with a lower degree of replication. The observation also holds for $zipf(0)$ and $zipf(2)$. Hence, for better performance, IBJ should consider only the tuple copy at the mean in the underlying storage.

Finally, we examine memory allocation among: i) the outer block and associated mappings; ii) the cache of index nodes; and iii) the cache of inner cells (or chunks). As in relational DBMS's, the third factor is least important, so we fix it to 1 chunk to focus on others. As Fig. 6 shows, when IBJ runs on *store-mean* or *store-multiple* with large step sizes (≥ 32), the I/O is dominated by the index I/O, not the inner array I/O. Hence, the more memory is given to the index, the better performance we obtain.

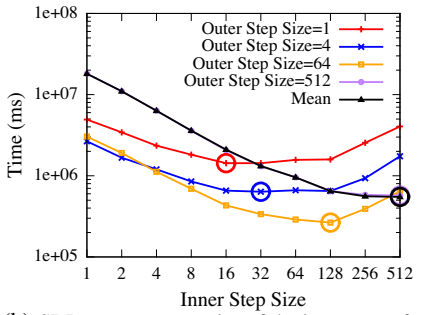
Expt 6: Subarray-Based Join (SBJ). The performance of SBJ is affected by the I/O cost for running repeated $Subarray$ queries on the inner array, and the CPU cost for validating paired tuples. To reduce CPU cost, as we pair tuples from an outer cell and its candidate cell, we implement a filter [16] to prune nonviable pairs quickly. SBJ enjoys a memory allocation scheme of giving most memory to the outer block and its mapping, which is used below.

We first demonstrate that SBJ's performance is sensitive to the storage scheme. Fig. 10(b) shows various combinations of the outer step size, k_{out}, and inner step size, k_{in}, with $\lambda = 0.9$. The x-axis shows different values of k_{in}. Each line represents a fixed value of k_{out}, with the corresponding optimal inner step size circled. There are two main trends: (1) For a fixed k_{out}, the optimal inner step size k^*_{in} is in the middle of its spectrum. As explained in Expt 1, the inner I/O first decreases and then increases with its step size. (2) Once k_{in} is fixed, the optimal k^*_{out} also occurs in the middle, because it achieves the best tradeoff between the pairing and filtering costs for the same outer tuple, which decreases with k_{out}, and the number of candidate cells to consider, which increases with k_{out}. The conclusion holds when $\lambda = 0.01$.

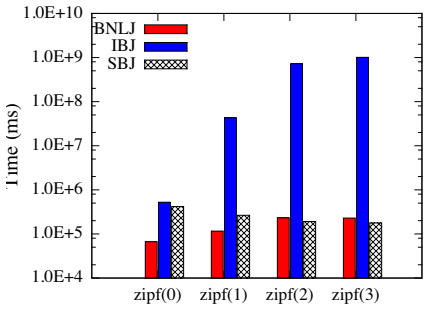
Next we show that the cost model in §4.2.2 can predicate the performance of SBJ so that given basic statistics, we can use it to choose the optimal step size configuration during data loading (if SJoin is known to the key workload). We again use the performance loss to evaluate the model accuracy. The results are shown in Table 4, where $\langle k_{out}\rangle$; $\langle k_{in}\rangle$ denotes the outer and inner step sizes. The model returns the optimal step sizes in all cases tested.



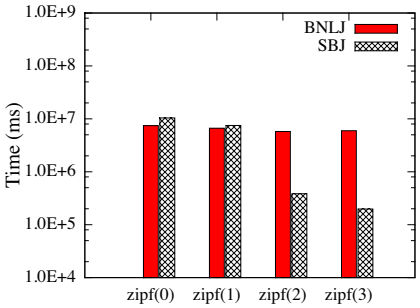
(a) IBJ cost w.r.t. step size of the inner array for ($zipf(1)$; 1%, 0.9)



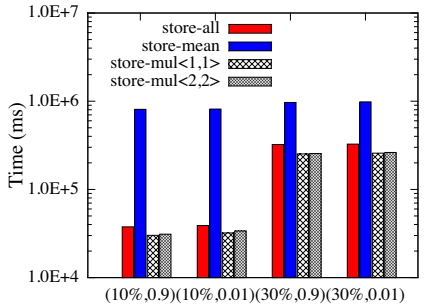
(b) SBJ cost w.r.t. step size of the inner array for ($zipf(1)$; 1%, 0.9)



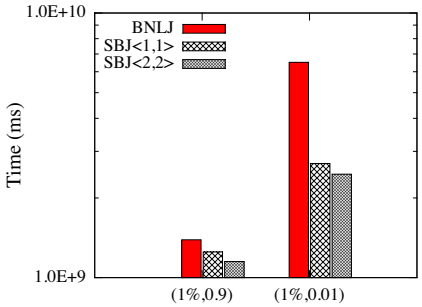
(c) Compare BNLJ, IBJ and SBJ for data and query workload (D_σ; 1%, 0.9)



(d) Compare BNLJ, IBJ and SBJ for data and query workload (D_σ; 1%, 0.01)



(e) SDSS Case study: *store-all* v.s. *store-mean* v.s. *store-multiple* for Subarray



(f) SDSS Case study: BNLJ v.s. SBJ for Structure-Join

Figure 10: Structure-Join results and the Case Study.

D_μ	D_σ	λ	optimal step size	model step size	perf. loss
$\mathcal{U}(0,1000)$	$zipf(0)$	0.9	$\langle 512\rangle;\langle 512\rangle$	$\langle 512\rangle;\langle 512\rangle$	**0%**
		0.01	$\langle 512\rangle;\langle 512\rangle$	$\langle 512\rangle;\langle 512\rangle$	**0%**
	$zipf(1)$	0.9	$\langle 64\rangle;\langle 128\rangle$	$\langle 64\rangle;\langle 128\rangle$	**0%**
		0.01	$\langle 512\rangle;\langle 512\rangle$	$\langle 512\rangle;\langle 512\rangle$	**0%**
	$zipf(2)$	0.9	$\langle 4\rangle;\langle 4\rangle$	$\langle 4\rangle;\langle 4\rangle$	**0%**
		0.01	$\langle 4\rangle;\langle 4\rangle$	$\langle 4\rangle;\langle 4\rangle$	**0%**

Table 4: SBJ Model Accuracy when $\delta = 1\%$

Expt 7: Comparison of Join Algorithms. We now use the step size returned by the model to configure SBJ, and compare it to IBJ and block nested loops joins (BNLJ) where both inner and outer arrays are stored using *store-mean*. The result is shown in Fig. 10(c) and Fig. 10(d) with one group of bars per dataset. As is mentioned in Expt 4, IBJ failed when $\lambda = 0.01$ due to the huge tuple-level mapping used. When $\lambda = 0.9$, IBJ still works poorly due to the tremendous index I/Os. For $zipf(0)$ and $zipf(1)$, BNLJ works better than SBJ because when many tuples have large possible ranges, the optimal step size configuration for SBJ is large (as shown in Table 4) which makes the expanded query region close to the entire inner array. Since SBJ consumes memory for maintaining the cell-level mapping, it requires more outer blocks than BNLJ and hence more repeated inner I/Os. But for $zipf(2)$ and $zipf(3)$ with fewer tuples of large variances, which represent more common workloads as show in our case study, SBJ outperforms BNLJ, e.g., 20% better when λ=0.9 and 95% better when λ=0.01. This is because SBJ does not incur much storage overhead and can effectively limit the number of inner cells to be loaded.

5.3 A Case Study using SDSS

We finally evaluate our techniques using the SDSS [23] dataset, which uses two dimension attributes $rowc$ and $colc$ and includes 1,893,685 tuples. Most tuples have small possible ranges; the Zipfian distribution fitted over the possible range sizes have skewness 2.4 for both $rowc$ and $colc$. The subarray model in §3.3 suggests $\langle 1, 1\rangle$ as the step size configuration regardless of the query size, while the SBJ model in §4.2.2 suggests $\langle 2, 2\rangle$ when $\delta = 1\%$ of the domain size is taken as a common workload for proximity join. When the two models do not return the same step size, the user should choose the right model by considering the importance (e.g., frequency and cost) of each type of query.

Regarding storage overheads, for the step size configuration $\langle 1, 1\rangle$, 79.28% of tuples have only one copy and 92.36% of tuples have at most three copies, and the numbers for $\langle 2, 2\rangle$ are 90.84% and 98.82%. As such, *store-multiple* incurs only a modest storage overhead when most tuples have concentrated distributions.

The *Subarray* performance is shown in Fig. 10(e). Although the two models return different step sizes, when the query is 10% of the domain size for each dimension, they both improve *store-mean* with fences by over 95% and *store-all* by over 13%. When the query is 30%, the numbers are 73% and 19%, respectively.

For *Structure-Join*, we compare SBJ only to the baseline BNLJ, since IBJ is shown to work poorly in §5.2. For multi-dimensional join, the filter [16] returns a superset of true matches. Therefore, different from the experiments in §5.2, the validation which involves an expensive integral per tuple pair also contributes to the CPU cost. The result is shown in Fig. 10(f). When input arrays are configured by the SBJ model, SBJ achieves 17.3% improvement over BNLJ when $\lambda = 0.9$ and 62.2% improvement when $\lambda = 0.01$. When input arrays are configured by the subarray model, the performance gains are 10.1% and 58.5%, respectively.

6. RELATED WORK

Most relevant techniques have been discussed in earlier sections. Below, we survey several broader areas.

Probabilistic processing under the array model. Recent work [12] observes that correlations in array data are mostly restricted to local areas and proposes a unified model for modeling both correlated data and physical storage. Monte Carlo processing has also been studied for join and sampling for uncertain array data [11]. As stated earlier, this line of work focuses on only value uncertainty in array data but not position uncertainty, i.e., it does consider the fact that uncertain attributes can be used as dimension attributes.

Indexing uncertain data. Recent work has addressed indexing uncertain data. Some existing indexes in relational probabilistic databases [7, 6, 10, 16] can be used in our index-based join or serve as filters to reduce validation cost in subarray evaluation and subarray-based join. Other indexes [17, 2, 1] are designed for nearest-neighbor queries, hence not directly applicable to our work.

7. CONCLUSIONS

In this paper, we addressed uncertain data management in array databases, which may involve both *value uncertainty* and *position uncertainty*. To support array operations under position uncertainty, we proposed a number of storage and evaluation schemes for *Subarray*, in particular, the *store-multiple* scheme, and building on that, the index-based join and subarray-based join for *Structure-Join*. Evaluation results show that for *Subarray*, *store-multiple* outperforms other alternatives by using a cost model to configure the storage and bounding the overhead of querying. For *Structure-Join*, the subarray-based join outperforms the index-based join by configuring the storage for the workload and avoiding many overheads in processing. Our case study using SDSS shows that for realistic datasets, the storage overhead of the *store-multiple* scheme is very limited and our best techniques for *Subarray* and *Structure-Join* outperform the baselines often by a wide margin.

In future work, we plan to extend our implementation to a broader set of array operations and integrate our techniques, which are fundamentally based on tuple-level replication, in big data systems that inherently maintain replicas for reasons like fault tolerance.

8. REFERENCES

[1] P. K. Agarwal, et al. Nearest-neighbor searching under uncertainty II. In *PODS*, 2013.

[2] P. K. Agarwal, et al. Nearest-neighbor searching under uncertainty. In *PODS*, 2012.

[3] P. G. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *SIGMOD*, 2010.

[4] R. Cheng, et al. Evaluating probabilistic queries over imprecise data. In *SIGMOD*, 2003.

[5] R. Cheng, et al. U-DBMS: a database system for managing constantly-evolving data. In *VLDB*, 2005.

[6] R. Cheng, et al. Efficient join processing over uncertain data. In *CIKM*, 2006.

[7] R. Cheng, et al. Efficient indexing methods for probabilistic threshold queries over uncertain data. In *VLDB*, 2004.

[8] P. Cudré-Mauroux, et al. A demonstration of scidb: A science-oriented dbms. *PVLDB*, 2009.

[9] Y. Diao, et al. Capturing data uncertainty in high-volume stream processing. In *CIDR*, 2009.

[10] T. Ge. Join queries on uncertain data: Semantics and efficient processing. In *ICDE*, 2011.

[11] T. Ge, et al. Monte carlo query processing of uncertain multidimensional array data. In *ICDE*, 2011.

[12] T. Ge and S. B. Zdonik. A*-tree: A structure for storage and modeling of uncertain multidimensional arrays. *PVLDB*, 2010.

[13] H. Kimura, et al. UPI: A primary index for uncertain databases. *PVLDB*, 2010.

[14] J. F. Kurose, et al. An end-user-responsive sensor network architecture for hazardous weather detection, prediction and response. In *AINTEC*, 2006.

[15] Large synoptic survey telescope: the widest, fastest, deepest eye of the new digital age. `http://www.lsst.org/`.

[16] L. Peng, et al. Optimizing probabilistic query processing on continuous uncertain data. *PVLDB*, 2011.

[17] B. E. Ruttenberg and A. K. Singh. Indexing the earth movers distance using normal distributions. In *VLDB*, 2012.

[18] SciDB. SciDB array functional language 11.06. `http://trac.scidb.org/wiki/Docs/Release_11.06/`.

[19] M. Stonebraker, et al. One size fits all? part 2: Benchmarking studies. In *CIDR*, 2007.

[20] M. Stonebraker, et al. Requirements for science data bases and scidb. In *CIDR*, 2009.

[21] M. Stonebraker, et al. The architecture of scidb. In *SSDBM*, 2011.

[22] D. Suciu, et al. Embracing uncertainty in large-scale computational astrophysics. In *MUD*, 2009.

[23] A. S. Szalay, et al. Designing and mining multi-terabyte astronomy archives: The sloan digital sky survey. In *SIGMOD*, 2000.

[24] T. T. L. Tran, et al. CLARO: modeling and processing uncertain data streams. *VLDB J.*, 21(5):651–676, 2012.

[25] T. T. L. Tran, et al. PODS: a new model and processing algorithms for uncertain data streams. In *SIGMOD*, 2010.