Supporting Data Uncertainty in Array Databases

Liping Peng
University of Massachusetts Amherst
lppeng@cs.umass.edu

Yanlei Diao
University of Massachusetts Amherst
yanlei@cs.umass.edu

ABSTRACT

Uncertain data management has become crucial to 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. Our work defines the formal semantics of array operations under both value and position uncertainty. To address the new challenge raised by position uncertainty, we propose a suite of storage and evaluation strategies for array operations, with a focus on a new scheme that bounds the overhead of querying by strategically treating tuples with large variances via replication in storage. Results from real datasets show that for common workloads, our best-performing techniques outperform alternative methods based on state-of-the-art indexes by 1.7x to 4.3x for the *Subarray* operation and 1 to 2 orders of magnitude for *Structure-Join*, at only a small storage cost.

1. INTRODUCTION

Uncertain data management has been studied intensively in areas such as sensor networks, information extraction, data cleaning, and business intelligence. Recently, it has also started to play a key role in large-scale scientific applications such as severe weather monitoring [27], computational astrophysics [28, 40], and asteroid threat detection [15]. In particular, recent studies [15, 39, 40, 29] 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.

There is a recent realization that most scientific data naturally reside in multi-dimensional arrays rather than relations, 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. Hence, array databases [15, 9, 38, 18], as well as relational databases with arrays as a first-class citizen [25, 6, 46], have been developed for scientific data processing. In particular, the new chunk-based storage scheme proposed in array databases enables better alignment of logical locality and physical locality (i.e., objects close to each other in the logical array are likely to be stored in the same chunk). Since many array operations exploit logical locality of data (e.g., finding objects close to a location), the associated physical locality can lead to significant I/O savings, which is a major reason that array databases can offer significant performance gains over relational databases [36].

The increasing popularity of array data management has significant implications on uncertain data management: Recent work on multidimensional arrays [22, 21] 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 the Sloan Digital Sky Survey (SDSS) [40] 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 [38]. Existing indexes for uncertain data can be built on array databases but can still incur high I/O cost, as we will show later in this paper.

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. We assume that tuples are loaded into an array database in a batched, append-only fashion, which is common in scientific applications [9, 38], and each tuple has obtained a (joint) distribution for uncertain attributes through a scientific cooking process, as described above. We then 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 efficient array operations on continuous uncertain data?

Given position uncertainty, naive solutions would replicate a tuple in every possible location in the array, or store the tuple once in a default location but to answer a query, search as widely as the entire array to retrieve all the tuples that satisfy the query with a high probability. These solutions incur both high I/O cost to read numerous tuples, and high CPU cost to validate retrieved tuples by computing their probabilities. Hence, the challenge in addressing position uncertainty lies in finding a storage and evaluation strategy that minimizes both I/O and CPU costs while returning all tuples that satisfy the query with a required probability. We address these challenges by strategically treating tuples with uncertain dimension attributes via (limited) replication in storage, which allows us

SIGMOD'15 May 31 - June 04, 2015, Melbourne, VIC, Australia

http://dx.doi.org/10.1145/2723372.2723738.

to fully exploit the locality benefit of array databases and bound the overhead of querying. More specifically, our contributions include:

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 arrays with uncertain dimension attributes (§3). 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 by strategically placing a few replicas for the tuples with large variances, which would otherwise make the query region grow very large. We also augment *store-multiple* with a detailed cost model and use it to configure storage for best performance under various workloads.

3. For *Structure-Join*, we propose a new evaluation strategy, called the subarray-based join (SBJ), which works without a prebuilt index and employs tight conditions for running repeated subarray queries on the inner array of the join, as well as a detailed cost model for configuring the storage for best performance.

4. We evaluate our techniques using both synthetic workloads and the Sloan Digital Sky Survey (SDSS) [40] (§5). For *Subarray*, *store-multiple* outperforms other alternatives due to the bounded overhead of querying and optimized storage based on the cost model. For *Structure-Join*, our SBJ outperforms existing join methods due to the tight conditions for probing the inner array and optimization based on the cost model. Our case study shows that for SDSS datasets, the storage overhead of *store-multiple* is rather small: **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 today's big data systems). In addition, our best techniques outperform those based on state-of-the-art indexes by 1.7x to 4.3x for *Subarray* and 1 to 2 orders of magnitude for *Structure-Join*.

2. ARRAY MODEL AND ALGEBRA

In this section, we provide background on the array model and array algebra proposed recently [9, 37]. 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: $\mathbb{A}^2(x\_loc, y\_loc; luminosity, color)$ defines a two-dimensional array with two dimension attributes (x_loc, y_loc) and value attributes $(luminosity, color)$. 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 mapping function `M` (e.g., the floor function) is assumed available for discretizing the domain into an ordered set of values. These ordered values are used as the index values in a given dimension, where the number of index values is determined by the domain size and the user function `M`.

In an array $\mathbb{A}^d$, a unique combination of the index values of the d dimensions defines a *cell*. Array cells are addressed by the index values of dimensions, e.g., a single cell addressed by $\mathbb{A}[1, 2]$, or multiple cells by $\mathbb{A}[2:6, 1:4]$. Since multiple values of a dimension attribute can be mapped to the same index value, a cell can contain multiple tuples. Tuples include the value attributes and in the continuous case, the dimension attributes as well since the attribute values offer differ from the index values. To draw an analogy with the relational model, we can translate $\mathbb{A}^d$ to a relation $\mathbb{R}(D_1, \ldots, D_d, V_1, \ldots, V_m)$ by treating dimension attributes as regular value attributes and storing tuples in no particular order.

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.

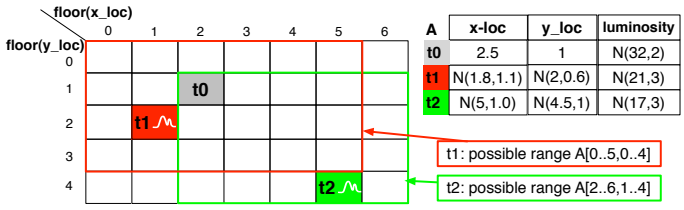
Uncertainty of dimension attributes is harder to support because a dimension attribute with multiple possible values can cause a tuple to belong to multiple cells in an array, referred to as *position uncertainty*. Consider a tuple t with uncertain dimension attributes. When the tuple position follows a (joint) discrete distribution, it can be stored in the cells corresponding to the possible values in the distribution. When the position follows a (joint) continuous distribution, instead, enumerating all values in the distribution is not possible. Hence, we define the tuple's **possible range** R_t as a hyper-rectangle within which the tuple existence probability is (approximately) 1. More specifically, we can construct R_t by taking t's marginal distribution, f_i, of each uncertain dimension. For example, if f_i is a uniform distribution $\mathrm{U}(a, b)$, the possible range is simply $[a, b]$ and the existence probability within this range is 1. If f_i is a normal distribution $\mathrm{N}(\mu, \sigma)$, the possible range $(\mu{-}3\sigma, \mu{+}3\sigma)$ achieves a probability 0.997. If f_i is an arbitrary distribution with 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. As convention in this paper, we always "round up" the possible region R_t to the boundary of cells, i.e., to be the smallest set of cells that contain R_t.

In this paper, we focus on the position uncertainty which has not been sufficiently addressed before. Our solution is compatible with existing techniques on value uncertainty because we aim to retrieve all tuples that overlap with a query region on the dimension attributes with a required probability (formally defined below). Once such tuples are returned as a set, uncertain value attributes can be handled by any techniques for relational databases [31, 41].

Example 2.1 *Fig. 1 shows an array, $\mathbb{A}(x\_loc, y\_loc; luminosity)$, where continuous uncertain attributes, x_loc and y_loc, are dimension attributes, and discretized by the $floor$ function for the 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 its mean value is in 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. The top-right corner in Fig. 1 shows the corresponding relation of array $\mathbb{A}$ in the relational model.*

2.2 Array Algebra

For multidimensional arrays, SciQL [25, 46] and the Array Query Language (AQL) [33] are two popular high-level declarative languages, while the Array Functional language (AFL) [33] is a functional language with a list of array operators. Since our work focuses on query processing, below we survey directly the operators in AFL. As those operators are proposed for tuples with deterministic values, we also extend their semantics to work under the uncertain data model, as shown in the following two categories.



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.

Value-based operators 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 *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 [42].

Structure-based operators operate on dimension attributes and optionally on value attributes as well. The common ones include:

(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 [33].[1]

[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.

(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 (a.k.a. l_1-distance) 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. 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 $\mathbb{A}^d$ 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.*

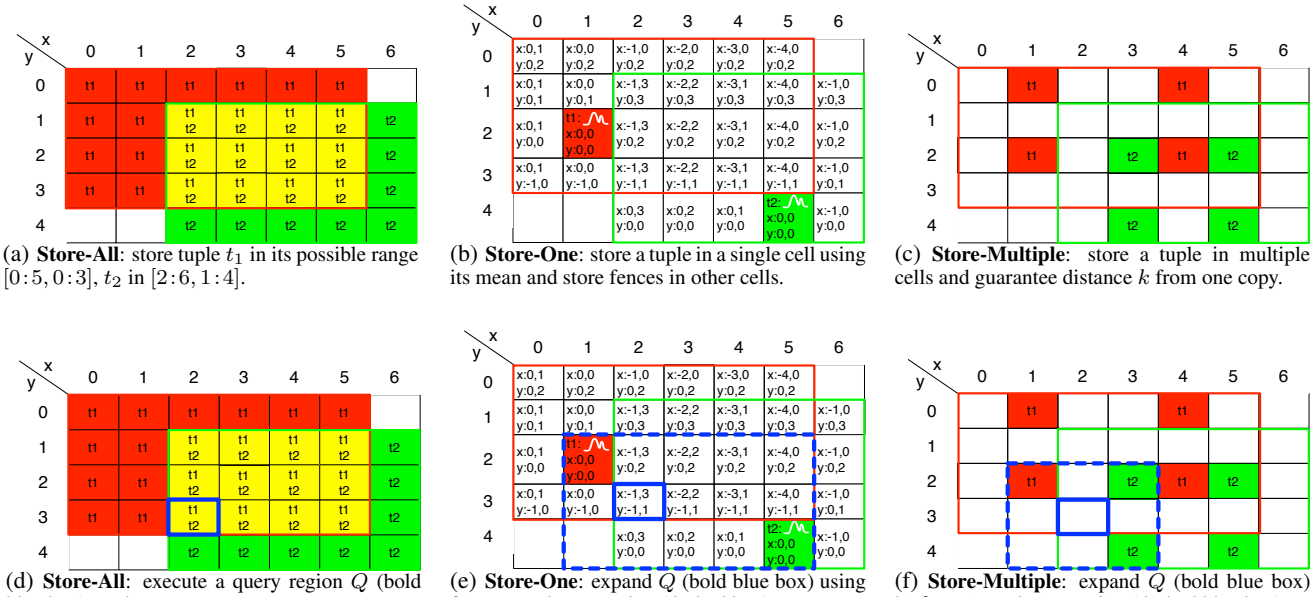
We survey additional structure operators in Appendix A. At the end of discussion, we show that many other structure operators can be implemented using *Subarray* and *Structure-Join*. Hence, we focus on them in the rest of the paper.

3. NATIVE SUPPORT FOR SUBARRAY

In this section, we focus on the *Subarray* operator under position uncertainty. More specifically, we focus on $Subarray(\mathbb{A}, \theta, \lambda)$, where $\theta = \bigwedge_{i=1}^{d}(a_i \leq \mathbb{A}.d_i \leq b_i)$ defines a hyper-rectangle in the d-dimensional space. In our work, other predicate shapes are supported by first relaxing them to a hyper-rectangle and then validating them using exact integration.

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 [10, 11, 41] and moments [31] of tuple distributions. However, these indexes may not be effective when the filtering power is low and can trigger many index I/O's, as we will show in §5. The second option is to build native support of *Subarray* in array databases where logical and physical localities are aligned. For instance, *Subarray* that exploits logical locality of data, e.g., looking for adjacent array cells from a point, may need to retrieve only a few relevant physical storage units called chunks. This effect of exploiting physical locality in an array database is similar to using a clustered primary index on the tuples in a relational database, but without having to build the index.

Hence, in this work we focus on native support of array operations on uncertain data. The task is challenging due to *position uncertainty*: each tuple can belong to multiple cells with non-zero probabilities and such cells form the tuple's "*possible range*" as defined in §2.1. The evaluation of *Subarray* takes two steps: (1) I/O step: the cells that store any tuple whose possible range overlaps with the 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 proba-



(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.

bility threshold. Basically, the first step ensures that no true results are missed and the second step guarantees that only the true results are returned. We aim to reduce both the number of chunks loaded (*I/O cost*), and the number of costly integrations to compute tuple probabilities (*CPU cost*) for all the tuples in the loaded chunks.

3.1 Storage and Evaluation Schemes

Below we propose a few schemes with the guarantee that tuple t can be retrieved if its possible range overlaps with a 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 missed results. The disadvantages include possibly high storage overheads and high I/O costs in querying because each logical cell may contain many physical chunks to store the replicated tuples.

Store-Mean: To reduce storage overheads, 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 fix the problem, the query region must be expanded. For ease of composition, given a region Q we define $\boldsymbol{C}(Q)$ to be the minimum set of cells that cover Q.

Definition 3.1 (Expanded Query Region) *Given a hyper-rectangle query region Q, its expanded query region $\tilde{Q}$ is a super hyper-rectangle $\tilde{Q}$ ($\supseteq Q$) such that any tuple whose possible range overlaps with Q has at least one copy stored in $\boldsymbol{C}(\tilde{Q})$.*

It is easy to see that reading all cells in $\boldsymbol{C}(\tilde{Q})$ in the I/O step can avoid missed results. However, the size of $\tilde{Q}$ varies with the storage scheme. For *store-all*, the expanded query region $\tilde{Q} = Q$ covers the least number of cells. For *store-mean*, without any auxiliary information, $\tilde{Q}$ should cover the whole array in the worst case.

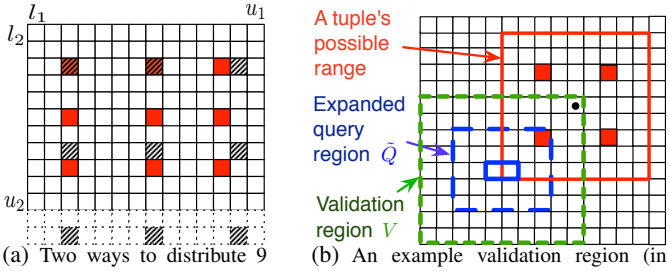
To constrain $\tilde{Q}$, we can augment each cell with upper and lower bounds for each dimension, indicating the distance to travel along each dimension in order to find all tuples that could belong to that cell—we call these bounds the *upper and lower fences* for expanding the query region from this 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. Fig. 2(e) shows that the query region (the solid blue box) covers a single 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. After walking on both dimensions, the expanded query region, marked by a dashed blue box, covers 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 other cell $\bar{C}$ in the tuple's possible range, compute its distance from the cell C, then expand $\bar{C}$'s 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.

The advantage of this strategy is small storage overhead in each cell, i.e., only two fences for each dimension, in contrast to *store-all* . However, the issue is that the expanded query region can grow very large, containing both relevant and irrelevant tuples, which will incur both high I/O cost for fetching all the tuples and high CPU cost for validating them using costly integration based on 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, walking at most k cells (steps) along each dimension is able to find a copy of the tuple. We call k the **step size**. Below we define an expanded query region for *store-multiple* and prove its optimality under this storage scheme.

Proposition 3.1 *Consider an array $\mathbb{A}^d$ under store-multiple with a step size configuration $\langle k_1, k_2, \cdots, k_d \rangle$ and a query region Q : (a_i, b_i) on each dimension i. Then $\tilde{Q}$: $(a_i - k_i s_i, b_i + k_i s_i)$, where*



(a) Two ways to distribute 9 tuple copies (the step size is 2).

(b) An example validation region (in green) for a given query region (in blue).

Figure 3: Tuple copy distribution and validation under *store-multiple*.

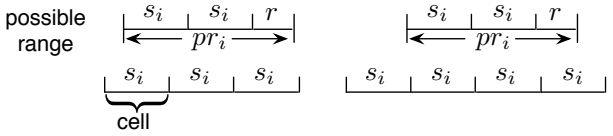


Figure 4: Illustration of the cells that a possible range overlaps with

s_i is the length of cell, is a valid expanded query region and requires to read the least number of cells on $\mathbb{A}$.

PROOF. We first prove that $\tilde{Q}$ is a valid expanded query region, i.e., for any tuple t with possible range $R_t \cap Q \neq \phi$, t will have at least a copy stored in $\boldsymbol{C}(\tilde{Q})$. Denote the cell $\mathbb{A}[x_1, x_2, \ldots, x_d]$ as $\mathbb{A}[\mathbf{x}]$ and a range of cells $\mathbb{A}[x_1 : y_1, x_2 : y_2, \ldots, x_d : y_d]$ as $\mathbb{A}[\mathbf{x} : \mathbf{y}]$ for short. Since $R_t \cap Q \neq \phi$, there exists a cell $\mathbb{A}[\mathbf{o}] \in \boldsymbol{C}(R_t) \cap \boldsymbol{C}(Q)$. It is easy to see that for any cell $\mathbb{A}[\mathbf{x}] \in \boldsymbol{C}(Q)$, $\mathbb{A}[\mathbf{x}-\mathbf{k} : \mathbf{x}+\mathbf{k}] \subseteq \boldsymbol{C}(\tilde{Q})$. Hence $\mathbb{A}[\mathbf{o}-\mathbf{k} : \mathbf{o}+\mathbf{k}] \subseteq \boldsymbol{C}(\tilde{Q})$. According to the definition of *store-multiple*, given $\mathbb{A}[\mathbf{o}] \in \boldsymbol{C}(R_t)$, there must exist one cell in $\mathbb{A}[\mathbf{o}-\mathbf{k} : \mathbf{o}+\mathbf{k}]$ that stores a copy of t. Then t will have at least a copy stored in $\boldsymbol{C}(\tilde{Q})$.

We next prove that reading a strict subset of $\boldsymbol{C}(\tilde{Q})$ can miss results. Assume cell $\mathbb{A}[\tilde{\mathbf{o}}] \in \boldsymbol{C}(\tilde{Q})$ is not read. Apparently, there exists a cell $\mathbb{A}[\mathbf{o}] \in \boldsymbol{C}(Q)$ such that $\mathbb{A}[\tilde{\mathbf{o}}] \in \mathbb{A}[\mathbf{o}-\mathbf{k} : \mathbf{o}+\mathbf{k}]$. Consider a tuple t with its possible range to be just the single cell $\mathbb{A}[\mathbf{o}]$. Then storing only one copy of t in cell $\mathbb{A}[\tilde{\mathbf{o}}]$ satisfies the definition of *store-multiple*. Since $\mathbb{A}[\mathbf{o}] \in \boldsymbol{C}(Q)$, tuple t can be a true result, but it will be missed as cell $\mathbb{A}[\tilde{\mathbf{o}}]$ is not read. □

Store-multiple overcomes the shortcomings of the previous two schemes: First, its controlled expansion of the query region, by k cells, 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. It is also worth noting that *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.

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 from each cell in t_1's possible region (the red rectangle), we need to walk only one step along 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.

Since *store-multiple* uses limited replication to constrain the expanded query region caused by tuples with large possible ranges, duplicate removal, a standard database technique, can be applied at the end of *Subarray* evaluation to remove duplicates. As an optimization for selective queries (which is the common case), the CPU step runs duplicate removal using an in-memory hash table to avoid repeated integrations for copies of the same tuple.

So far we have introduced the *store-multiple* storage and the *Subarray* evaluation under *store-multiple*. Two questions still remain: First, the way to store tuples while guaranteeing the step size k 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 3.2 (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 3.3 (Covering Set) *A given set of cells $\boldsymbol{C}$ is covered by a (discrete) set of cells $\boldsymbol{S}$ if and only if each cell in $\boldsymbol{C}$ is covered by at least one cell in $\boldsymbol{S}$; $\boldsymbol{S}$ is called the covering set of $\boldsymbol{C}$.*

Definition 3.4 (Problem of Tuple Copy Layout) *Given a tuple t, find the minimum covering set $\boldsymbol{S}$ of its possible range $\boldsymbol{C}(R_t) = \mathbb{A}[l_1 : u_1, l_2 : u_2, \cdots, l_d : u_d]$ so that placing one copy of t in each cell in $\boldsymbol{S}$ is a valid layout under store-multiple with the step size configuration $\langle k_1, k_2, \cdots, k_d \rangle$. That is, walking k_i steps from any cell in $\boldsymbol{C}(R_t)$ along all dimensions is able to find a copy of t.*

We address the problem by first showing the lower bound of the size of a covering set, as shown in the following proposition. The proof can be found in Appendix B.

Proposition 3.2 *Given an array $\mathbb{A}^d$ under store-multiple with a step size configuration $\langle k_1, k_2, \cdots, k_d \rangle$, if tuple t's possible range is $\boldsymbol{C}(R_t) = \mathbb{A}[l_1 : u_1, l_2 : u_2, \cdots, l_d : u_d]$, at least $\prod_{i=1}^{d} (\lfloor (u_i - l_i) / (2k_i + 1) \rfloor + 1)$ cells are needed to cover $\boldsymbol{C}(R_t)$.*

Note that in the worst case, Proposition 3.2 may suggest an explosion of the number of cells (and tuple replicas in those cells) to cover a tuple's possible range. In practice, most real-world datasets have 2 to 3 dimensions to reflect our physical space, and the majority of tuples have some degree of concentration in the location distribution. Take SDSS for example. When the cell size is set to 1, the possible ranges of most tuples on dimension attributes $(rowc, colc)$ are within 2.5×2.5 cells on average. According to Proposition 3.2, one copy is enough for most tuples for *store-multiple* with $k = 1$, the same as *store-mean* and 1/9 of *store-all*. We will show how to choose an appropriate step size configuration in §3.3.

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: on the i-th dimension where the possible range is l_i, u_i, the first copy is stored at $l_i + k_i$ and the other copies are stored $2k_i$ cells away

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.

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 and requires at least 9 copies to be placed. The layout of 9 copies is shown by the shaded cells (ignore the red color for now). However, three copies are stored outside the tuple's possible range, which will increase the chance of reading irrelevant copies when a query region falls outside the tuple's possible range. It is thus desirable to store all copies of a tuple inside its possible range. In our work, 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 copies, 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 by the red cells in the figure. Thus we still use the minimum number of copies to cover the tuple's possible range.

3.3 A Cost Model for Optimization

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 [9], 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[2]. For *Subarray* evaluation under *store-multiple*, 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 uniformly distributed 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. These assumptions can be relaxed, 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 with which the possible range $[l_i, u_i]$ overlaps is

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

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. To compute the latter, we first write $u_i - l_i + 1 = pr_i/s_i + 1$ based on Eq.(1), and plug it into Proposition 3.2 to derive the number of copies per tuple t_{copies}:

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

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

$$C_{chunks} = T \cdot t_{copies} \Big/ \prod_{i=1}^{d} n_i \Big/ \lfloor c/b \rfloor . \qquad (3)$$

Multiplying C_{chunks} 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 $\tilde{Q}$, denoted as $\tilde{Q}_{chunks}$:

$$\tilde{Q}_{chunks} = C_{chunks} \cdot \prod_{i=1}^{d}\left(\frac{q_i}{s_i} + 1 + 2k_i\right). \qquad (4)$$

CPU Cost: To capture CPU cost, we model the number of tuples to be validated, T_{val}. Given an expanded query region $\tilde{Q}$, a tuple is retrieved for validation as long as it has one copy stored in $\tilde{Q}$. 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 number of cells in V first. 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, $V = \tilde{Q}$, with $(q_i/s_i + 1 + 2k_i)$ cells. (2) When k_i is small so that all tuples have more than one copy, $V \supset \tilde{Q}$, as shown by Fig. 3(b) with one of the furthest tuples that needs to be validated: the tuple's possible range is the red box; it has a copy in $\tilde{Q}$ but its center of the possible range, marked by a black dot, lies outside $\tilde{Q}$. To get V, we need to further expand $\tilde{Q}$ by the distance between the green and blue dashed lines in Fig. 3(b), denoted as $\Delta = \lceil (pr_i/s_i + 1)/2 \rceil - (k_i + 1)$ cells, along each direction of dimension i. Expanding $\tilde{Q}$ along both directions, V has $(q_i/s_i + 1 + 2k_i) + 2\Delta \approx (q_i/s_i + 1 + pr_i/s_i)$ cells on the i-th dimension. Summarizing the two cases and multiplying the size of V with the average number of tuples per cell, we have:

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

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 using integration.

A Generalized Model. We next relax two assumptions made previously in our model: (1) When tuples 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; (2) When tuples are not evenly distributed in the domain, we can feed statistics of tuples' mean positions and the query position into our model to get a more accurate estimate: instead of using $T/\Pi_{i=1}^{d} n_i$, which is the average number of tuples per cell, we can use the number of tuples in each cell of the query. In practice, we can collect above-mentioned statistics when a batch of tuples comes in. For instance, SDSS [40] updates the scanned image of the sky on a nightly basis and can build the statistics as a

[2]The relationship between logical cells and physical chunks is further discussed in Apprendix E.2.

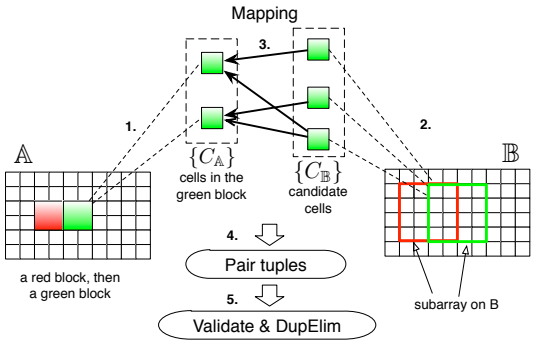


Figure 5: Illustration of subarray-based join.

nightly observation is being produced. If domain knowledge shows that the statistics do 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, at data loading time we estimate the costs of representative queries by running our model for a wide range of step size configurations (which runs once and fast), and choose the configuration that offers the best performance. More implementation details are given in Appendix C.

4. SUPPORT FOR STRUCTURE JOIN

In this section, we focus on the *Structure-Join* operator under position uncertainty. More specifically, we focus on linear proximity (a.k.a. l_1-distance) join, i.e., $SJoin(\mathbb{A}^d, \mathbb{B}^d, \theta, \lambda)$ where $\theta = \bigwedge_{i=1}^{d} |\mathbb{A}.d_i - \mathbb{B}.d_i| < \delta$. Non-linear proximity join based on Euclidean distance, e.g., $\theta = \sum_i (\mathbb{A}.d_i - \mathbb{B}.d_i)^2 < \delta^2$, can be first relaxed to linear proximity join, and then followed by additional filtering using exact integration based on θ. More complex predicates are further discussed in Appendix E.1.

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, existing indexes for relational databases [13, 12, 31] can be built on top of the *store-mean* scheme. As we will show in §5.3, such index-based evaluation can incur many index and data I/O's. Here we propose a new evaluation strategy, called subarray-based join (SBJ), which does not require a pre-built index, as well as model-based optimization to achieve best performance.

4.1 Subarray-based Join (SBJ)

Similar to block nested loops joins, *Structure-Join* can be transformed into iterative *Subarray* operations on the inner array. Assume that the smaller array, $\mathbb{A}$, is the outer. The basic idea is that for each outer cell $C_\mathbb{A}$, we form a subarray condition $\theta_{C_\mathbb{A}}$ on the inner array $\mathbb{B}$, run the *Subarray* query on $\mathbb{B}$ to retrieve relevant tuples, and finally pair the A tuples and B tuples for exact evaluation using integration. For best performance, 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. Below we propose several necessary conditions for linear proximity join that guarantee to return all join results.

Given a tuple $t_\mathbb{A}$, let $\left(l_{t_\mathbb{A}}^{d_i}, u_{t_\mathbb{A}}^{d_i}\right)$ denote the lower and upper bounds of its possible range on dimension i. Similarly, we have $\left(l_{t_\mathbb{B}}^{d_i}, u_{t_\mathbb{B}}^{d_i}\right)$ 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 $\left(l_{t_\mathbb{A}}^{d_i} - \delta,\ \ u_{t_\mathbb{A}}^{d_i} + \delta\right)$ and $\left(l_{t_\mathbb{B}}^{d_i}, u_{t_\mathbb{B}}^{d_i}\right)$ overlap on each dimension i $(i = 1, \ldots, d)$.*

The proof is in Appendix B. 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 the largest possible ranges of the 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:

- The value range of cell $C_\mathbb{A}$ on dimension d_i is $(l_{C_\mathbb{A}}^{d_i}, u_{C_\mathbb{A}}^{d_i})$.
- 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 $C_\mathbb{A}^+ = \mathbb{A}[x_1 + k_1, \ldots, x_d + k_d]$.

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

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 $\left(l_{t_\mathbb{A}}^{d_i}, u_{t_\mathbb{A}}^{d_i}\right) \subseteq \bigcup_{C_\mathbb{A} \in S_{t_\mathbb{A}}} \left(l_{C_\mathbb{A}^-}^{d_i}, u_{C_\mathbb{A}^+}^{d_i}\right)$: When $t_\mathbb{A}$ needs only one copy to cover its possible range on dimension d_i, assume the copy is stored at $C_\mathbb{A}$, then $\left(l_{t_\mathbb{A}}^{d_i}, u_{t_\mathbb{A}}^{d_i}\right) \subseteq \left(l_{C_\mathbb{A}^-}^{d_i}, u_{C_\mathbb{A}^+}^{d_i}\right)$ because otherwise it needs at least two copies. When $t_\mathbb{A}$ has more than one copy 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(a). So $l_{t_\mathbb{A}}^{d_i} = \min_{C_\mathbb{A} \in S_{t_\mathbb{A}}} l_{C_\mathbb{A}^-}^{d_i}$ and $u_{t_\mathbb{A}}^{d_i} = \max_{C_\mathbb{A} \in S_{t_\mathbb{A}}} u_{C_\mathbb{A}^+}^{d_i}$, i.e., $\left(l_{t_\mathbb{A}}^{d_i}, u_{t_\mathbb{A}}^{d_i}\right) = \bigcup_{C_\mathbb{A} \in S_{t_\mathbb{A}}} \left(l_{C_\mathbb{A}^-}^{d_i}, u_{C_\mathbb{A}^+}^{d_i}\right)$. Combining the two cases, we have $\left(l_{t_\mathbb{A}}^{d_i}, u_{t_\mathbb{A}}^{d_i}\right) \subseteq \bigcup_{C_\mathbb{A} \in S_{t_\mathbb{A}}} \left(l_{C_\mathbb{A}^-}^{d_i}, u_{C_\mathbb{A}^+}^{d_i}\right)$. Then for any tuple $t_\mathbb{B}$, if its possible range $\left(l_{t_\mathbb{B}}^{d_i}, u_{t_\mathbb{B}}^{d_i}\right)$ overlaps with $\left(l_{t_\mathbb{A}}^{d_i} - \delta,\ \ u_{t_\mathbb{A}}^{d_i} + \delta\right)$, 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 can be missed. □

Algorithm 1 Subarray-based Join (SBJ)

```
1: for each read block R_𝔸 in 𝔸 do
2:    toRead.clear(); map.clear();
3:    for each cell C_𝔸 in R_𝔸 do
4:       loadToMemory(C_𝔸);
5:       Q ← formQueryRegion(C_𝔸); S ← Subarray(𝔹, Q);
6:       for each cell C_𝔹 in S do
7:          toRead.add(C_𝔹); map.get(C_𝔹).add(C_𝔸);
8:    for each cell C_𝔹 in toRead do
9:       loadToMemory(C_𝔹);
10:      for each cell C_𝔸 in map.get(C_𝔹) do
11:         for each tuple t_𝔸 in C_𝔸 do
12:            for each tuple t_𝔹 in C_𝔹 do
13:               filter(t_𝔸, t_𝔹); validate(t_𝔸, t_𝔹);
14: removeDuplicates();
```

We now present subarray-based join (SBJ) in Algorithm 1 and illustrate it with Fig. 5. The algorithm processes one block of the outer at a time (Line 1 in Algorithm 1; marked as Step 1 in Fig. 5, 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 $\mathbb{B}$ (Line 5; 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 union of the candidate cells of all outer cells in the current block (Line 7), in $\{C_\mathbb{B}\}$ in Fig. 5. 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. 5 (Line 7; Step 3). Then the algorithm reads relevant cells of $\mathbb{B}$ and pairs tuples accordingly (Line 8-12; Step 4). As optimization, It applies quick filters with negligible costs to the paired tuples to reduce later CPU cost. It finally does validation using the join condition and removes duplicates (Line 13-14; Step 5).

4.2 A Cost Model for Optimization

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 $\mathbb{A}$. 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 with $\mathbb{B}$ blocks (shown in Fig. 5). 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. We have 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 $C_\mathbb{A}$, i.e., cells in the expanded query region, is:

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

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:

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

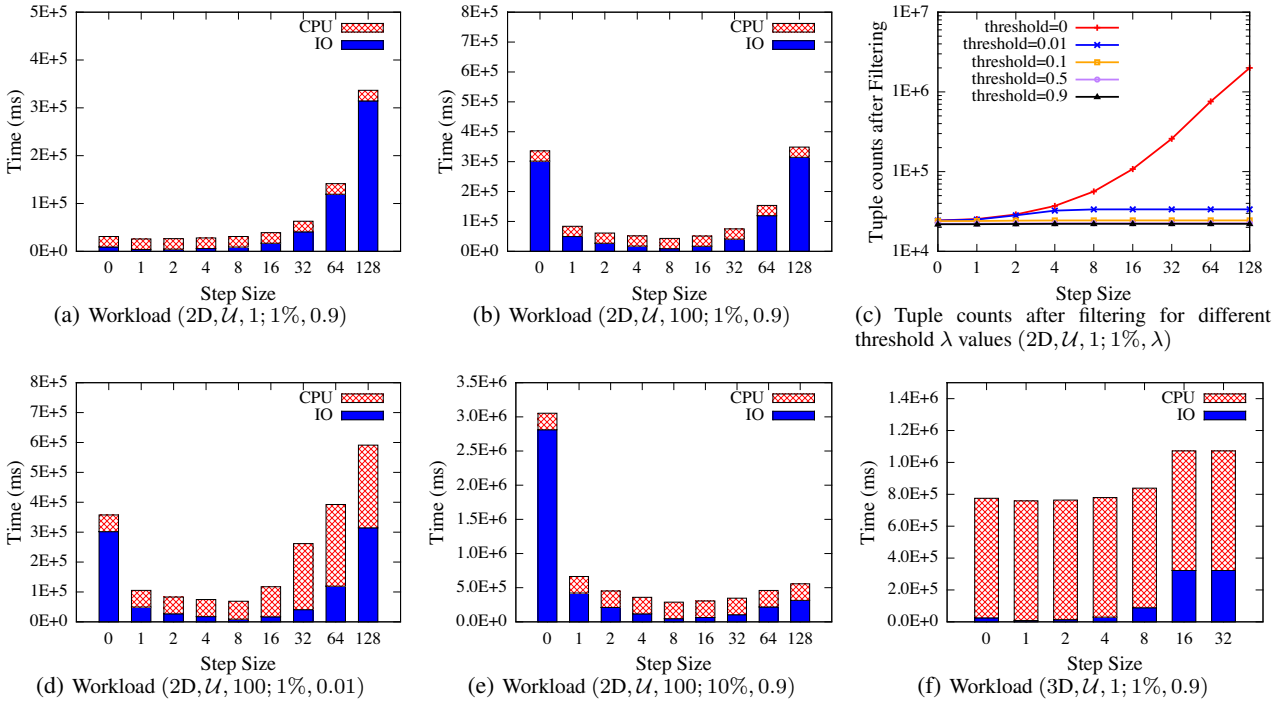
Finally, the combined IO and CPU model allows us to find the optimal step sizes for inner and outer arrays if *SJoin* is the key workload. Statistics needed are the distribution of tuples' possible ranges and common distance values in joins. Collecting such statistics is a common task of the query optimizer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

5. EXPERIMENTS

We evaluate our techniques for *Subarray* and *Structure-Join* using both a wide range of synthetic workloads with controlled properties and the Sloan Digital Sky Survey (SDSS) [40].

5.1 Experimental Setup

SDSS Datasets. Consider queries on dimension attributes ($rowc$, $colc$) in SDSS. SDSS treats them as independent attributes and does not provide any correlation coefficient between them. SDSS describes each dimension attribute using a Gaussian distribution, $N(\mu, \sigma)$. Here, μ is specified by the value of attribute $rowc$ (or $colc$) and determines where the center of a tuple's possible range is located; σ is specified by the value of attribute $rowcErr$ (or $colcErr$) and determines how wide a tuple's possible range is along



(a) Workload $(2D, \mathcal{U}, 1; 1\%, 0.9)$ (b) Workload $(2D, \mathcal{U}, 100; 1\%, 0.9)$ (c) Tuple counts after filtering for different threshold λ values $(2D, \mathcal{U}, 1; 1\%, \lambda)$

(d) Workload $(2D, \mathcal{U}, 100; 1\%, 0.01)$ (e) Workload $(2D, \mathcal{U}, 100; 10\%, 0.9)$ (f) Workload $(3D, \mathcal{U}, 1; 1\%, 0.9)$

Figure 6: Cost breakdown of *Subarray* with varied step sizes for various workloads.

dimension $rowc$ (or $colc$). Without loss of generality, we consider a tuple's possible range per dimension to be $\mu \pm 3\sigma$. The distributions of $rowcErr$ and $colcErr$ are very similar. A tuple's possible range along both dimension $rowc$ and $colc$ is 2.5 on average.

Synthetic Datasets. Our synthetic datasets of dimensionalities 1, 2 and 3 (which are the most common in scientific applications) are generated based on the statistics of ($rowc$, $colc$) in SDSS. The parameters of synthetic datasets are summarized in Table 2. All the datasets have 2M tuples stored in around 60000 cells of size 1. In order to study the effect of the validation (i.e., integration) cost, different from SDSS datasets, here we generate tuples with correlated dimension attributes; the CPU cost per integration for correlated attributes is much higher than that for independent attributes because it increases exponentially in dimensionality. Like in SDSS, each tuple is described by a (multivariate) Gaussian distribution.[3] We generate μ values using the distribution, D_μ, which is set to either a uniform distribution over the domain or a Gaussian distribution with more tuples clustered at the center. To obtain datasets with various average possible range sizes, we collect the top 10 frequent σ values in SDSS and rescale the possible range size (which is determined by σ) in SDSS by a factor denoted as S_σ. For example, to generates a 2D dataset with $S_\sigma = 16$, σ values collected from SDSS are rescaled by a factor of 4 per dimension. We generate one dataset for each combination of data parameter configurations.

Our evaluation starts with our own techniques, and later in the SDSS case study also compares to state-of-the-art index schemes for uncertain data, G-index [31] and U-index [13, 41], and baseline methods such as Block Nested Loops Join. Our experiments were run on two identical servers, each with Intel(R) Xeon(R) CPU 5160 @3.00GHz, 8GB memory, JVM 1.7.0 on CentOS 6.4.

[3]Other distributions will not change our reported results because: (1) the I/O cost does not vary with the distribution and is only affected by the possible range size; (2) the CPU cost depends on the integration cost which can vary with the distribution, but we have already included a range of integration costs using multivariate distributions.

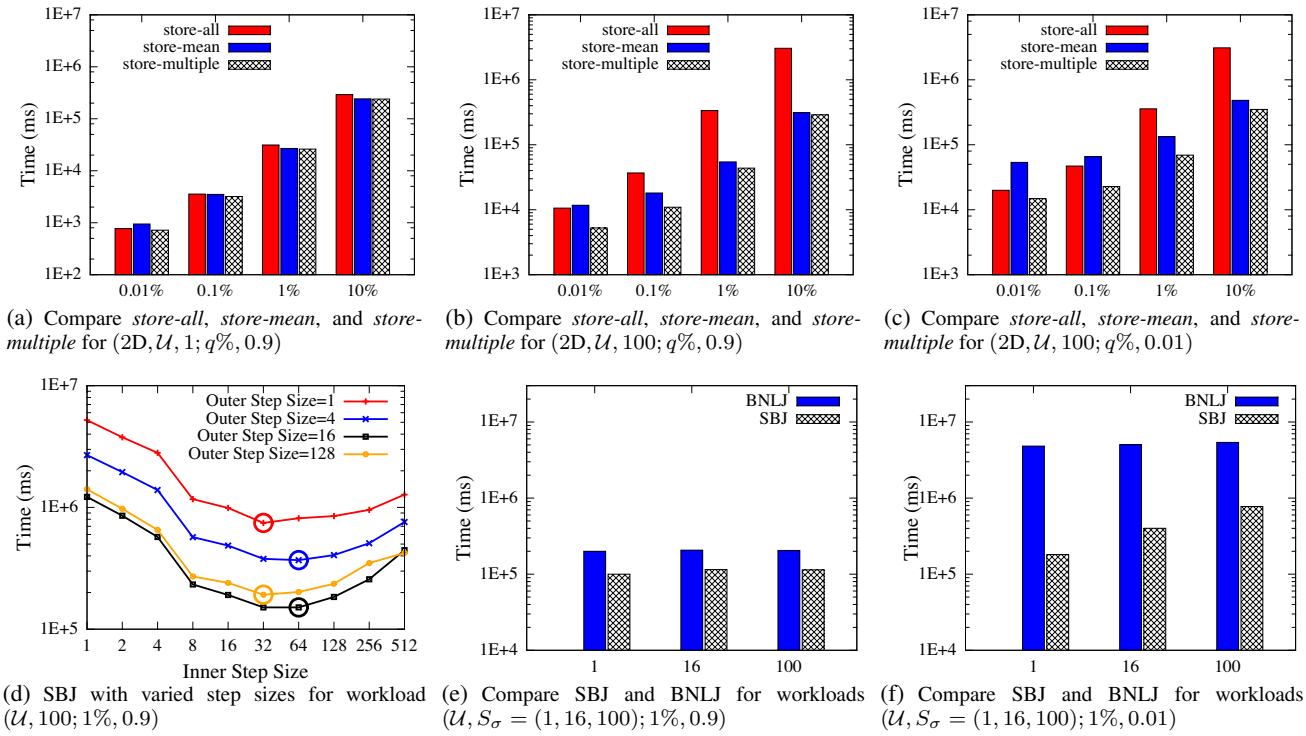
5.2 Evaluation of Our Subarray Techniques

We configure *Subarray* queries using the parameters in Table 2: We vary the query size, q, between 0.01% and 10% of the domain. The threshold, λ, prunes tuples based on the existence probability. Usually the user wants the tuples with high existence probabilities; we use λ=0.9 to represent this workload. We also tested λ=0.01 (e.g., needed if there is an aggregate after *Subarray*). The evaluation of *Subarray* includes both the I/O step and the CPU step. We optimize the CPU step by first running fast filters [31] with negligible costs before computing the expensive integral for the exact existence probability of each retrieved tuple. Memory is set to be 10% of the data size. We first use synthetic data with controlled properties in this set of experiments.

Expt 1: Cost Breakdown. Our *store-multiple* scheme has a 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. Fig. 6(a) shows results for the default workload, (2D, D_μ=$\mathcal{U}$, S_σ=1; q=1%, λ=0.9), while Fig. 6(b) shows results for S_σ=100, with an enlarged average possible range size and magnified trends. The overall trends are:

(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. 6(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



(a) Compare *store-all*, *store-mean*, and *store-multiple* for $(2D, \mathcal{U}, 1; q\%, 0.9)$

(b) Compare *store-all*, *store-mean*, and *store-multiple* for $(2D, \mathcal{U}, 100; q\%, 0.9)$

(c) Compare *store-all*, *store-mean*, and *store-multiple* for $(2D, \mathcal{U}, 100; q\%, 0.01)$

(d) SBJ with varied step sizes for workload $(\mathcal{U}, 100; 1\%, 0.9)$

(e) Compare SBJ and BNLJ for workloads $(\mathcal{U}, S_\sigma = (1, 16, 100); 1\%, 0.9)$

(f) Compare SBJ and BNLJ for workloads $(\mathcal{U}, S_\sigma = (1, 16, 100); 1\%, 0.01)$

Figure 7: Evaluation results of *Subarray* and *Structure-Join* on synthetic datasets.

	Parameter	Default	Other Values
Data	dimensionality	2	1, 3
	D_μ, distribution of μ	uniform ($\mathcal{U}$)	normal ($\mathcal{N}$)
	S_σ, scale factor of possible range	1	16, 100
Query	q, query range / domain	1%	0.01%, 0.1%, 10%
	λ, probability threshold	0.9	0.01

Table 2: Parameters in $Subarray$ experiments on synthetic datasets.

after filtering does not change with the step size, or the number of tuples retrieved from storage. We examined the filter's effect using multiple datasets and our observation is consistent.

To further study the effect of λ and q, we tune their values from Fig. 6(b) one at a time: we change λ from 0.9 to 0.01 and show the cost breakdown in Fig. 6(d); we also change q from 1% to 10% in Fig. 6(e). Finally Fig. 6(f) shows the cost breakdown for a 3D workload. It can be seen from these plots that, between CPU cost and I/O cost, which is dominating depends on many factors, including λ (by comparing Fig. 6(b) and Fig. 6(d)), the system constants like the per integration cost (by comparing Fig. 6(a) and Fig. 6(f)), and the step size configuration (by comparing bars within each plot). It is challenging to find the optimal optimal step size: we observe that *the optimal step size shifts right when the average possible range increases* (by comparing Fig. 6(a) and 6(b)); *it shifts left when λ is very small or the per integration cost is high*, and *it increases with the query region size*.

Expt 2: Model Accuracy. We next use the cost model in §3.3 to determine the step size 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 loss of our model, $(Cost(\tilde{\mathbf{k}}) - Cost(\mathbf{k}^*))/Cost(\mathbf{k}^*)$. 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 3×3 query regions for 2D datasets and $2 \times 2 \times 2$ for 3D datasets that evenly scattered over the array, and report on the average.

Table 2 shows 144 combinations of parameters. Our model returns the optimal step size (i.e., no performance loss) in 89.6% of workloads when the tuples' μ values are uniformly distributed and in 83.3% of workloads when the tuples' μ values are normally distributed. In those cases when our model selects a suboptimal step size, the average performance loss is 2.72%, which shows that our model is effective in configuring the *store-multiple* scheme.

Expt 3: Comparing 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. 7(a)-7(c) for different workloads. Each plot shows four queries with different query region sizes.

In all cases, *store-multiple* works the best. In comparison, when all tuples have small possible ranges, the three storage schemes do not differ much because *store-all* incurs only a small storage overhead and the expanded query region for *store-mean* is also very constrained, as shown in Fig. 7(a). However, for datasets when $S_\sigma = 100$, *store-all* often incurs tremendous storage overheads and I/O costs in querying, as shown in Fig. 7(b) and 7(c). Moreover, *store-multiple* outperforms *store-mean* considerably when the query region q is small, e.g., q <1%, which is the common case, due to a more constrained expanded query region. When q grows larger, e.g., q=10%, their difference is reduced because the optimal step size of *store-multiple* tends to be larger. This means that infrequent replication of tuples works fine if q is large, and most tuples have only one copy under *store-multiple*, similar to *store-mean*.

5.3 Evaluation of Structure-Join

We next consider the *Structure-Join* where both the inner and outer arrays are loaded from the same dataset. We start with 1D *Structure-Join*, $SJoin\,(\mathbb{A}_1, \mathbb{A}_2, |\mathbb{A}_1.x - \mathbb{A}_2.x| < \delta, \lambda)$, of 100,000 tuples, mainly chosen for efficiency reasons. (Later, our case study considers 2D *Structure-Join* on SDSS datasets with up to 90 million

S_σ	λ	Optimal step size	Model step size	Performance loss
1	0.9	$\langle 2\rangle;\langle 4\rangle$	$\langle 4\rangle;\langle 4\rangle$	**5.3%**
	0.01	$\langle 2\rangle;\langle 2\rangle$	$\langle 2\rangle;\langle 2\rangle$	**0%**
16	0.9	$\langle 8\rangle;\langle 8\rangle$	$\langle 8\rangle;\langle 16\rangle$	**3.6%**
	0.01	$\langle 8\rangle;\langle 8\rangle$	$\langle 8\rangle;\langle 8\rangle$	**0%**
100	0.9	$\langle 16\rangle;\langle 32\rangle$	$\langle 16\rangle;\langle 32\rangle$	**0%**
	0.01	$\langle 8\rangle;\langle 8\rangle$	$\langle 16\rangle;\langle 16\rangle$	**0%**

Table 3: SBJ Model Accuracy when $\delta = 1\%$

tuples.) We use a recent index on continuous uncertain data [31] as an in-memory filter whenever possible. This index returns only true matches for 1D joins, so validation is not needed for 1D joins. The memory size is 10% of the data size.

Expt 4: Subarray-Based Join (SBJ). We fix δ to 1% of the domain. SBJ incurs the I/O cost for running repeated *Subarray* queries on the inner array, and the CPU cost for filtering [31]. We find that allocating most memory to the outer block and its mapping structure works the best and use this scheme in all experiments below.

We first demonstrate that SBJ's performance is sensitive to the storage scheme. Fig. 7(d) shows various combinations of the outer step size, k_{out}, and inner step size, k_{in}, with $\lambda = 0.9$. Each line represents a fixed value of k_{out}, and the x-axis varies values of k_{in}, with the 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 (e.g., k^*_{out}=16), because it achieves the best tradeoff between (a) pairing and filtering costs for the same outer tuple, which decreases with larger k_{out}, and (b) the number of candidate cells to consider, which increases with k_{out} due to the enlarged expanded subarray region.

Next we show that the cost model in §4.2 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 be the key workload). We again use the performance loss to evaluate the model accuracy. The results are shown in Table 3, where $\langle k_{out}\rangle;\langle k_{in}\rangle$ denotes the outer and inner step sizes. The model returns the optimal step sizes in most cases and the overall performance loss is within 6% (if any).

Expt 5: Comparison of Join Algorithms. We now use the step size returned by the model to configure subarray-based join (SBJ), and compare it to a baseline, block nested loops join (BNLJ) where both inner and outer arrays are stored using *store-mean*. Fig. 7(e) and Fig. 7(f) show the results when the tuples' possible range sizes are scaled up, for the probability threshold λ=0.9 and λ=0.01, respectively: 1) For all datasets tested, SBJ outperforms BNLJ, e.g., 46.3% better when λ=0.9 and 91.3% 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 read, as opposed to reading the entire inner array, for each outer block. Hence, SBJ saves both I/Os on the inner array and the CPU cost by reducing the number of tuples to be filtered. 2) SBJ's performance is not sensitive to large variance tuples when λ=0.9, but shows an increase in cost when λ=0.01, because many more tuples will be returned as join results.

5.4 Case Study using SDSS Datasets

We next perform a case study using SDSS datasets and queries. We collect SDSS datasets with 1.89 to 90 million tuples, with the total database size ranging from 295MB to 24GB. Each dataset consists of tuples in a subregion of the next bigger dataset. We use $(rowc, colc)$ as dimension attributes. Since they are treated as independent attributes in SDSS, the CPU cost per validation is much reduced from a 2-dimensional integration to two 1-dimensional integrations. Hence, I/O cost dominates in this study. Memory is set to be 10% of the data size.

Datasets	Store-mean	Store-all	Store-multiple	U-index	G-index
1.89M	85MB	178MB	**89MB**	404MB	282MB
10.3M	479MB	1.2GB	**569MB**	2.2GB	1.5GB
30.2M	1.4GB	4.5GB	**1.8GB**	6.3GB	4.4GB
90M	3.8GB	14GB	**4.5GB**	19G	13.2GB

Table 4: Storage comparison of SDSS datasets on ($rowc$, $colc$).

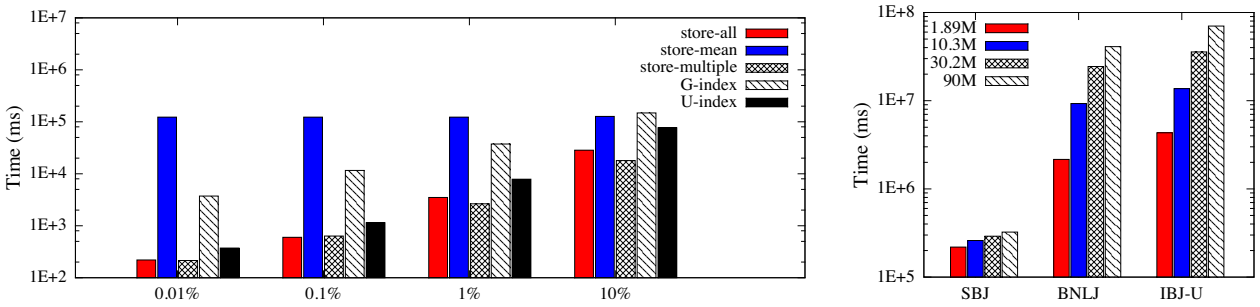
We evaluate our techniques for *Subarray* and *Structure-Join* against two state-of-the-art indexes for uncertain data, G-index [31] and U-index [13, 41] (with general index implementation outlined in Appendix C and reviews of these two indexes in Appendix D).

Expt 6: Storage. We first compare our storage schemes with alternative indexing schemes. Table 4 shows the disk space that each data structure on the dimension attributes takes. We see that *store-multiple* configured with step size <1,1> by our cost model incurs much less storage cost than the index schemes, and it approximates store-mean (which has the smallest possible storage cost) but with much better performance, as shown below. Specifically, over 79% tuples have only 1 copy and over 92% tuples have at most 3 copies.

Expt 7: Subarray. We first evaluate *subarray* queries on ($rowc$, $colc$) with varied query size q and probability threshold λ=0.9, using the SDSS dataset with 1.89M tuples. All non-leaf nodes are prefetched into memory. Since each subarray query hits a region of the array uniformly at random, we do not consider the effect of caching of leaf nodes here. The results are shown in Fig. 8(a). (1) *Comparison to results of synthetic data*: The comparison among storage schemes is similar to Fig. 7(a), except that *store-mean* with fences is orders of magnitude slower than other schemes when $q \leq 1\%$, and is 7 times slower than *store-multiple* when q =10%. This is because its expanded query region almost covers the entire array due to the existence of very large variance tuples (e.g., 2039.78^2), and such tuples' σ values are not in the top 10 frequent values we used to generate our synthetic datasets. The absolute values are much less than those in Fig. 7(a) because the two dimension attributes in SDSS are independent and the per validation cost is much less than that for correlated attributes. (2) *Comparison to index-based methods*: *store-multiple* is 1.7 - 4.3 times faster than U-index and 8.3 - 18.3 times faster than G-index, because it finds a good tradeoff between the tuple replication and query expansion. In contrast, probing on-disk indexes incurs tremendous leaf I/Os due to the nature of multi-path search in R-tree based indexes. Based on our profiling numbers, when we vary q from 0.01% to 10%, the accessed child nodes averaged over all non-leaf nodes are in the range of (10.33%, 52.93%) for U-index and (22.13%, 66.29%) for G-index. Additional analysis of both indexes and results on the I/O counts are shown in Appendix D.

Expt 8: Structure-join. We finally consider a query used in SDSS's sample query set to find neighboring objects (shown in Appendix D). It is a join of two arrays on dimension attributes ($rowc$ and $colc$) and selects 10 value attributes from each with astronomical meanings to further evaluate whether each neighboring pair meets certain criteria. As typical of SDSS queries, a predicate is posed on the outer array such that a subset of it (i.e., a small patch of the sky) is joined with the inner. The probability threshold is 0.9. We evaluate all the join algorithms using the same outer array but four inner arrays with different sizes to test scalability.

Since *structure-join* repeatedly probes the inner array or the index on the inner, caching plays an important role. Our memory setting, 10% of the data size including all dimension and value attributes in the query, is enough to hold all non-leaf nodes, as shown in Table. 5. For IBJ, as many non-leaf nodes as the mem-



(a) *Subarray* on SDSS (λ=0.9, varied q) (b) *Structure-Join* on SDSS ($\delta = 1$, $\lambda = 0.9$)

Figure 8: Case study on SDSS datasets.

ory allows are pre-fetched, and the remaining memory is used as an LRU cache of both the leaf nodes and inner array chunks. For 2D datasets, G-index triggers more index I/Os than U-index as shown in Expt 7, and hence is omitted in the study below. The results are shown in Fig. 8(b), with one group of bars per join algorithm and bars in each group representing different sizes of the inner array.

(1) *Comparison to Index Join*: IBJ with U-index works poorly, 1 to 2 orders of magnitude worse than SBJ. Based on profiling results for 1.89M tuples, **the index I/O dominates**. With store-mean, the large-variance tuples lead to large range queries on the inner and most (even all) of the leaf nodes are accessed. With the (rare) existence of such tuples, the average number of U-index leaf nodes accessed per outer tuple is 38.9. Further, such tuples destroy the locality of caching for the same reason. For the 76830 outer tuples, with a 91.2% cache hit rate, the amount of leaf I/Os is already 263498, more than 10 times worse than SBJ. In contrast, store-multiple addresses large variance tuples with replication and finds a good tradeoff between tuple replication and query expansion. In addition, SBJ puts a tight predicate on the inner array to read relevant inner cells, and utilizes the memory to form blocks of outer tuples so that many of them can share the inner I/Os. As such, SBJ largely preserves the data locality that the array database provides for the access to dimension attributes. To verity this, we compared SBJ with an ideal case where each relevant inner chunk (i.e., containing the join candidates for some outer tuple) is visited exactly once. **SBJ approximates the ideal case with 1.6x-2x I/Os.**

(2) *Comparison to BNLJ*: The difference between SBJ and BNLJ is magnified as BNLJ scans the whole inner array for each outer block, which is much bigger in SDSS than the synthetic datasets.

(3) *Scalability*: SBJ scales the best among the three. For the same outer block and the same δ, the subarray region formed using Proposition 4.3 is exactly the same. However, bigger datasets have more tuples with large possible ranges and increased chance of having tuple copies in formed subarray regions, which results in a modest increase of the cost.

Additional results on I/O counts are given in Appendix D.

6. RELATED WORK

Probabilistic processing under the array model. Recent work [22] 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 [21]. As stated earlier, this line of work focuses on only value uncertainty in array data but not position uncertainty, i.e., it does not consider the fact that uncertain attributes can be used as dimension attributes.

Probabilistic relational databases. There is a large body of work on probabilistic databases in the relational setting, which addresses the semantics (e.g., [16, 4, 42]) and efficient query processing (e.g., [35, 44, 31, 7]). Systems such as ORION [10, 11, 13, 12] and CLARO [31, 42] support uncertain data modeled by continuous random variables, which fit most scientific data. These techniques can be applied in our system to handle value uncertainty.

Of particular relevance to our work on position uncertainty is indexing and storing multi-dimensional uncertain data, including earlier work in ORION [10, 13] and more recent work [41, 26, 20, 31]. They can be leveraged in array databases as well, but can trigger many index I/O's (as we showed in §5) and may not be effective when the filtering power is low. In contrast, we aim to provide native support in the array model, where logical and physical localities are better-aligned and the effect of exploiting physical locality is similar to using a clustered primary index on the tuples in a relational database, but without having to build the index.

Other indexes [8, 1, 32, 3, 2] are designed for similarity and nearest-neighbor queries, not directly applicable to our work. [30] uses secondary storage to record query lineage and efficiently compute tuple existence probabilities, but their focus is on discrete random variables in the relational model, not on continuous random variables in the array model.

Redundant storage for efficient query processing. Also related is the work on using redundant storage for answering point enclosure and range queries in an I/O efficient way [34, 24, 5, 45]. To map to that work, we can translate our subarray query in two steps: First, find all tuples whose possible ranges (bounding boxes of tuples' distributions) intersect the query rectangle, which however cannot be simply solved by point enclosure and range queries [5]. Second, compute the existence probabilities of candidate tuples and validate them against a probability threshold, which is CPU-intensive and not considered in prior work (while our work does).

Spatial databases. Most prior work on spatial databases [41, 14, 19, 43] use the relational model. In contrast, array databases differ by using a new chunk-based storage scheme that allows objects logically close in an array to be likely to be stored in the same physical chunk, a key property that our work leverages for performance.

7. CONCLUSIONS

To address the new challenge posed by position uncertainty in array databases, we proposed a number of storage and evaluation schemes for *Subarray*, in particular, the *store-multiple* scheme, and building on that, the subarray-based join (SBJ) for *Structure-Join*. Our case study on real-world workloads shows that for *Subarray*, *store-multiple* is 1.7x- 4.3x faster than a state-of-the-art index, U-index, and for *Structure-Join*, SBJ is 1 to 2 orders of magnitude faster than U-index based join. Such improvement does not require pre-built indexes and comes with very limited storage overhead: for real datasets, 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 today's big data systems).

8. ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under the grants IIS-1218524.

9. REFERENCES

[1] Uncertain spatial data handling: Modeling, indexing and query. *Computers & Geosciences*, 33(1):42 – 61, 2007.

[2] P. K. Agarwal, B. Aronov, S. Har-Peled, J. M. Phillips, K. Yi, and W. Zhang. Nearest-neighbor searching under uncertainty ii. In *PODS*, 2013.

[3] P. K. Agarwal, A. Efrat, S. Sankararaman, and W. Zhang. Nearest-neighbor searching under uncertainty. In *PODS*, 2012.

[4] L. Antova, T. Jansen, C. Koch, and D. Olteanu. Fast and simple relational processing of uncertain data. In *ICDE*, pp. 983–992, 2008.

[5] L. Arge, V. Samoladas, and K. Yi. Optimal external memory planar point enclosure. *Algorithmica*, 54(3):337–352, 2009.

[6] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system rasdaman. In *SIGMOD*, pp. 575–577, 1998.

[7] O. Benjelloun, A. D. Sarma, A. Y. Halevy, and J. Widom. Uldbs: Databases with uncertainty and lineage. In *VLDB*, pp. 953–964, 2006.

[8] C. Bohm, A. Pryakhin, and M. Schubert. The gauss-tree: Efficient object identification in databases of probabilistic feature vectors. In *ICDE*, 2006.

[9] P. G. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *SIGMOD*, pp. 963–968, 2010.

[10] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In *SIGMOD*, pp. 551–562, 2003.

[11] R. Cheng, S. Singh, and S. Prabhakar. U-dbms: a database system for managing constantly-evolving data. In *VLDB*, pp. 1271–1274, 2005.

[12] R. Cheng, S. Singh, S. Prabhakar, R. Shah, J. S. Vitter, and Y. Xia. Efficient join processing over uncertain data. In *CIKM*, pp. 738–747, 2006.

[13] R. Cheng, Y. Xia, S. Prabhakar, R. Shah, and J. S. Vitter. Efficient indexing methods for probabilistic threshold queries over uncertain data. In *VLDB*, pp. 876–887, 2004.

[14] E. Crestaz and A. Pistocchi. *Spatial Data Management in GIS and the Coupling of GIS and Environmental Models*, pp. 217–252. John Wiley & Sons, Inc., 2014.

[15] P. Cudré-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. J. DeWitt, B. Heath, D. Maier, S. Madden, J. M. Patel, M. Stonebraker, and S. B. Zdonik. A demonstration of scidb: A science-oriented dbms. *PVLDB*, 2(2):1534–1537, 2009.

[16] N. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *VLDB J.*, 16(4):523–544, 2007.

[17] Y. Diao, B. Li, A. Liu, L. Peng, C. Sutton, T. Tran, and M. Zink. Capturing data uncertainty in high-volume stream processing. In *CIDR* 2009.

[18] J. Duggan and M. Stonebraker. Incremental elasticity for array databases. In *SIGMOD*, pp. 409–420, 2014.

[19] A. Fox, C. Eichelberger, J. Hughes, and S. Lyon. Spatio-temporal indexing in non-relational distributed databases. In *IEEE International Conference on Big Data*, pp. 291–299, 2013.

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

[21] T. Ge, D. Grabiner, and S. B. Zdonik. Monte carlo query processing of uncertain multidimensional array data. In *ICDE*, pp. 936–947, 2011.

[22] T. Ge and S. B. Zdonik. A*-tree: A structure for storage and modeling of uncertain multidimensional arrays. *PVLDB*, 3(1):964–974, 2010.

[23] J. Gray, A. S. Szalay, A. R. Thakar, G. Fekete, W. O'Mullane, M. A. Nieto-Santisteban, G. Heber, and A. H. Rots. There goes the neighborhood: Relational algebra for spatial data search. *CoRR*, cs.DB/0408031, 2004.

[24] J. M. Hellerstein, E. Koutsoupias, D. P. Miranker, C. H. Papadimitriou, and V. Samoladas. On a model of indexability and its bounds for range queries. *J. ACM*, 49(1):35–55, 2002.

[25] M. Kersten, Y. Zhang, M. Ivanova, and N. Nes. Sciql, a query language for science applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pp. 1–12, 2011.

[26] H. Kimura, S. Madden, and S. B. Zdonik. Upi: A primary index for uncertain databases. *PVLDB*, 3(1):630–637, 2010.

[27] J. F. Kurose, E. Lyons, D. McLaughlin, D. Pepyne, B. Philips, D. Westbrook, and M. Zink. An end-user-responsive sensor network architecture for hazardous weather detection, prediction and response. In *AINTEC*, pp. 1–15, 2006.

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

[29] J. Niedermayer, A. Züfle, T. Emrich, M. Renz, N. Mamoulis, L. Chen, and H. Kriegel. Probabilistic nearest neighbor queries on uncertain moving object trajectories. *PVLDB*, 7(3):205–216, 2013.

[30] D. Olteanu and J. Huang. Secondary-storage confidence computation for conjunctive queries with inequalities. In *SIGMOD*, pp. 389–402, 2009.

[31] L. Peng, Y. Diao, and A. Liu. Optimizing probabilistic query processing on continuous uncertain data. *PVLDB*, 4, 2011.

[32] B. E. Ruttenberg and A. K. Singh. Indexing the earth moverÕs distance using normal distributions. In *VLDB*, 5(3):205–216, 2012.

[33] SciDB. Scidb array functional language. `http://scidb.org/HTMLmanual/13.3/scidb_ug/ch01s04s01.html`.

[34] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *VLDB*, pp. 507–518, 1987.

[35] P. Sen, A. Deshpande, and L. Getoor. Exploiting shared correlations in probabilistic databases. In *VLDB*, 2008.

[36] M. Stonebraker, C. Bear, U. Çetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. B. Zdonik. One size fits all? part 2: Benchmarking studies. In *CIDR*, pp. 173–184, 2007.

[37] M. Stonebraker, J. Becla, D. J. DeWitt, K.-T. Lim, D. Maier, O. Ratzesberger, and S. B. Zdonik. Requirements for science data bases and scidb. In *CIDR*, 2009.

[38] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. The architecture of scidb. In *SSDBM*, pp. 1–16, 2011.

[39] D. Suciu, A. Connolly, and B. Howe. Embracing uncertainty in large-scale computational astrophysics. In *MUD*, 2009.

[40] A. S. Szalay, P. Z. Kunszt, A. Thakar, J. Gray, D. R. Slutz, and R. J. Brunner. Designing and mining multi-terabyte astronomy archives: The sloan digital sky survey. In *SIGMOD*, pp. 451–462, 2000.

[41] Y. Tao, X. Xiao, and R. Cheng. Range search on multidimensional uncertain data. *ACM Trans. Database Syst.*, 32(3), Aug. 2007.

[42] T. T. L. Tran, L. Peng, Y. Diao, A. McGregor, and A. Liu. Claro: modeling and processing uncertain data streams. *VLDB J.*, 21(5):651–676, 2012.

[43] F. Urbano and H. Dettki. Storing tracking data in an advanced database platform (postgresql). In *Spatial Database for GPS Wildlife Tracking Data*, pp. 9–24. Springer International Publishing, 2014.

[44] D. Z. Wang, E. Michelakis, M. Garofalakis, and J. Hellerstein. Bayesstore: Managing large, uncertain data repositories with probabilistic graphical models. In *VLDB*, 2008.

[45] T. Zäschke, C. Zimmerli, and M. C. Norrie. The ph-tree: A space-efficient storage structure and multi-dimensional index. In *SIGMOD*, pp. 397–408, 2014.

[46] Y. Zhang, M. Kersten, and S. Manegold. Sciql: Array data processing inside an rdbms. In *SIGMOD*, pp. 1049–1052, 2013.

APPENDIX

A. MORE ON ARRAY OPERATORS

Below, we survey additional structure operators besides (1) sub-array and (2) structure-join.

(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. It 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 (a superset of those that are physically stored 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 above, *Subarray* and *Structure-Join* are the two most important primitives in array algebra. Hence, we focus on efficient implementation of them under position uncertainty in this paper.

B. PROOFS

B.1 Proof of Proposition 3.2

PROOF. We can pick a subset of cells from the region $\boldsymbol{C}(R_t) = \mathbb{A}[l_1 : u_1, l_2 : u_2, \cdots, l_d : u_d]$ as follows: $\boldsymbol{C'} = \{\mathbb{A}[x_1, x_2, \cdots, x_d] | \ \forall i \in \{1, 2, \ldots, d\},\ x_i = l_i + p_i(2k_i + 1)$ 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 $|\boldsymbol{C'}|$ is $\prod_{i=1}^{d} (\lfloor (u_i - l_i)/\ \ (2k_i + 1) \rfloor + 1)$. Based on Definition 3.3, if we can prove that at least $|\boldsymbol{S'}|$ cells are already needed just to cover $\boldsymbol{C'}$, then at least $|\boldsymbol{C'}|$ cells are needed to cover the superset $\boldsymbol{C}(R_t)$.

Let us assume a cell $\mathbb{A}[x_1, x_2, \cdots, x_d] \in \boldsymbol{C'}$ 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 $\mathbb{A}[x'_1, x'_2, \cdots, x'_d] \in \boldsymbol{C'} - \{\mathbb{A}[x_1, x_2, \cdots, x_d]\}$, 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 $\boldsymbol{C'}$ can be covered by the same cell. In other words, at least $|\boldsymbol{C'}|$ cells are needed in order to cover $\boldsymbol{C'}$. Then to cover $\boldsymbol{C}(R_t)$, a superset of $\boldsymbol{C'}$, at least $|\boldsymbol{C'}| = \prod_{i=1}^{d} (\lfloor (u_i - l_i)/(2k_i + 1) \rfloor + 1)$ cells are needed. □

B.2 Proof of Proposition 4.1

In §2.1, we define a tuple's possible range as a hyper-rectangle within which the tuple existence probability is (approximately) 1. Before we prove Proposition 4.1, let us define it formally.

Definition B.1 (Possible Range) *For a tuple whose dimension attributes are modeled by a joint distribution $f(\mathbf{x})$, its possible range on dimension i is (l_i, u_i) such that $\int_{-\infty}^{l_i} f(x_i)\mathrm{d}x_i = \epsilon/2$ and $\int_{u_i}^{+\infty} f(x_i)\mathrm{d}x_i = \epsilon/2$, where $f(x_i)$ is the marginal distribution of $f(\mathbf{x})$ on dimension i and ϵ is 0 or a sufficiently small positive number.*

For queries considered in this paper, the query threshold λ should be (much) greater than ϵ. Below we prove Proposition 4.1.

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 $\left(l^{d_i}_{t_{\mathbb{A}}} - \delta,\ \ u^{d_i}_{t_{\mathbb{A}}} + \delta\right)$ and $\left(l^{d_i}_{t_{\mathbb{B}}}, u^{d_i}_{t_{\mathbb{B}}}\right)$ do not overlap, i.e., $l^{d_i}_{t_{\mathbb{A}}} - \delta > u^{d_i}_{t_{\mathbb{B}}}$ or $u^{d_i}_{t_{\mathbb{A}}} + \delta < l^{d_i}_{t_{\mathbb{B}}}$. Without loss of generality, let us assume $l^{d_i}_{t_{\mathbb{A}}} - \delta > u^{d_i}_{t_{\mathbb{B}}}$. 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 $\{(\mathbf{x}, \mathbf{y}) | \bigwedge_{i=1}^{d} |x_i - y_i| < \delta\}$. We start with finding an upper bound. 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. 9 and partitioned into three parts. Denote the probability mass of each partition as $\tilde{p}_1$, $\tilde{p}_2$ and $\tilde{p}_3$ respectively. Below we derive the upper bound for each of them by applying the assumption.

$$\begin{aligned}
\tilde{p}_1 &= \int_{-\infty}^{l^{d_i}_{t_{\mathbb{A}}} - \delta} f_{t_{\mathbb{B}}.d_i}(y_i) \left(\int_{y_i - \delta}^{y_i + \delta} f_{t_{\mathbb{A}}.d_i}(x_i)\mathrm{d}x_i \right) \mathrm{d}y_i \\
&< \int_{-\infty}^{l^{d_i}_{t_{\mathbb{A}}} - \delta} f_{t_{\mathbb{B}}.d_i}(y_i) \left(\int_{-\infty}^{l^{d_i}_{t_{\mathbb{A}}}} f_{t_{\mathbb{A}}.d_i}(x_i)\mathrm{d}x_i \right) \mathrm{d}y_i \\
&= \frac{\epsilon}{2} \int_{-\infty}^{l^{d_i}_{t_{\mathbb{A}}} - \delta} f_{t_{\mathbb{B}}.d_i}(y_i)\mathrm{d}y_i < \frac{\epsilon}{2} \\
\tilde{p}_2 &= \int_{l^{d_i}_{t_{\mathbb{A}}} - \delta}^{l^{d_i}_{t_{\mathbb{A}}} + \delta} f_{t_{\mathbb{B}}.d_i}(y_i) \left(\int_{y_i - \delta}^{l^{d_i}_{t_{\mathbb{A}}}} f_{t_{\mathbb{A}}.d_i}(x_i)\mathrm{d}x_i \right) \mathrm{d}y_i \\
&< \int_{u^{d_i}_{t_{\mathbb{B}}}}^{+\infty} f_{t_{\mathbb{B}}.d_i}(y_i) \left(\int_{-\infty}^{l^{d_i}_{t_{\mathbb{A}}}} f_{t_{\mathbb{A}}.d_i}(x_i)\mathrm{d}x_i \right) \mathrm{d}y_i \\
&= \frac{\epsilon}{2} \int_{u^{d_i}_{t_{\mathbb{B}}}}^{+\infty} f_{t_{\mathbb{B}}.d_i}(y_i)\mathrm{d}y_i = \frac{\epsilon}{2} \cdot \frac{\epsilon}{2} = \frac{\epsilon^2}{4} \\
\tilde{p}_3 &= \int_{l^{d_i}_{t_{\mathbb{A}}}}^{+\infty} f_{t_{\mathbb{A}}.d_i}(x_i) \left(\int_{x_i - \delta}^{x_i + \delta} f_{t_{\mathbb{B}}.d_i}(y_i)\mathrm{d}y_i \right) \mathrm{d}x_i \\
&< \int_{l^{d_i}_{t_{\mathbb{A}}}}^{+\infty} f_{t_{\mathbb{A}}.d_i}(x_i) \left(\int_{u^{d_i}_{t_{\mathbb{B}}}}^{+\infty} f_{t_{\mathbb{B}}.d_i}(y_i)\mathrm{d}y_i \right) \mathrm{d}x_i \\
&= \frac{\epsilon}{2} \int_{l^{d_i}_{t_{\mathbb{A}}}}^{+\infty} f_{t_{\mathbb{A}}.d_i}(x_i)\mathrm{d}x_i = \frac{\epsilon}{2} \left(1 - \frac{\epsilon}{2}\right) = \frac{\epsilon}{2} - \frac{\epsilon^2}{4}
\end{aligned}$$

Finally we have $p < \tilde{p} = \tilde{p}_1 + \tilde{p}_2 + \tilde{p}_3 = \epsilon < \lambda$, 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. □

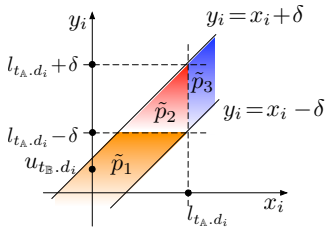


Figure 9: Illurstraion of $\tilde{p} = \tilde{p}_1 + \tilde{p}_2 + \tilde{p}_3$.

C. IMPLEMENTATION DETAILS

Our implementation follows the *append-only* and *columnar storage* design as used in SciDB [9]. To illustrate, we show the storage of three tuples, colored in yellow, red and green respectively, in Fig. 10, which each have two dimension attributes, (x_loc, y_loc), and one value attribute, $luminosity$. The logical structure of the array is determined by applying a user-defined discretization function, e.g., $floor$, on the dimension attributes, so that the cells along

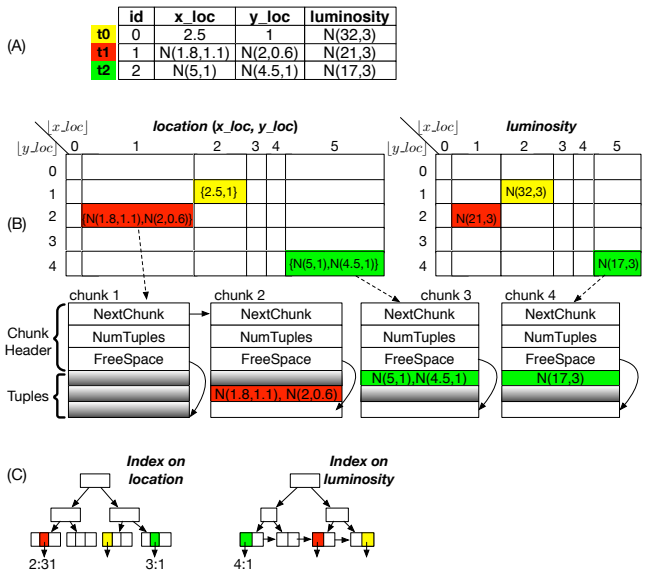


Figure 10: Implementation details

each dimension attribute have index values 0, 1, 2, . . . Given a new batch of tuples, the *insertion* routine takes two steps (which can be easily modified to support tuple insertion one-at-a-time):

1. *Placement in the logical array*: The insertion routine first iterates over the tuples. Based on the values of a tuple's dimension attributes and the storage-scheme in use (e.g., *store-mean* or *store-multiple*), it determines which logical cell(s) the tuple belongs to. At the end, each logical cell obtains a list of tuples to insert into.

2. *Vertical partitioning and chunking*: In physical storage, attributes of tuples are vertically partitioned and written to multiple arrays that share the same logical structure. In our implementation, the dimension attributes are stored in one array as they are usually queried together, which we call the *dimension attribute array*, and each value attribute is stored in its own array, called the *value attribute array*. Fig. 10(B) shows the storage of three tuples in two arrays of the same logical structure, one for the dimension attributes (x_loc,y_loc) and the other for the value attribute $luminosity$.

In the second step of the insertion routine, we iterate over the logical cells. For each logical cell and its associated tuples, we partition the attributes of tuples into the dimension attribute array and value attribute arrays. In a (dimension or value) attribute array, each cell stores tuple attributes in a series of chunks (which is the physical storage unit). Note that the cell id and the insertion order of a tuple in a cell are the same across all the attribute arrays. This allows us to easily reconstruct the tuple with all relevant attributes in query processing, which is a standard technique in column databases. For example, in Fig. 10(B), tuple t_2 is the first tuple in cell $[5, 4]$, and its (x_loc,y_loc) values and the $luminosity$ value are stored as the first item in chunk 3 and chunk 4, respectively.

There is system metadata that records the first chunk, illustrated by the dashed arrows in Fig. 10(B), and the last chunk for each cell. Each chunk has a chunk header and stores multiple tuples. The chunk header consists of the address of the next overflow chunk (if any) with a default value -1, the number of tuples in the current chunk and a pointer to the free space in the current chunk. In Fig. 10(B), cell $location[1, 2]$ has more than one chunks and the luminosity value of tuple t_1 is stored in the second chunk.

Indexes can be built on top of each attribute array (usually only on the copy of a tuple at the mean position). Each entry in the leaf nodes stores the chunk id and insertion order of the corresponding tuple in that cell. As shown in Fig. 10(C), there is one index on (x_loc, y_loc) and another on $luminosity$, and the entries for tuple t_2 in the indexes store the chunk id 3 and 4, respectively, with the same insertion order 1. Given a query on the dimension attributes (a focus of this paper), the index on the dimension attribute array can be used to identify relevant tuples. If the query requires other attributes to be accessed or returned, the additional attributes of those tuples are fetched from other attribute arrays, using standard operations in column databases.

Datasets (tuples)		1.89M	10.3M	30.2M	90M
Memory		29.5MB	163MB	806.6MB	2.4GB
Cache size (4K pages)		7174	39631	196167	607012
U-index	non-leaf	5256	28722	83388	242217
	leaf	98062	535182	1561376	4515894
G-index	non-leaf	1332	7176	20858	62038
	leaf	70469	384988	1123279	3350078

Table 5: Cache size and node counts for different datasets.

D. MORE DETAILS ON CASE STUDY

Query in the case study for structure-joins:

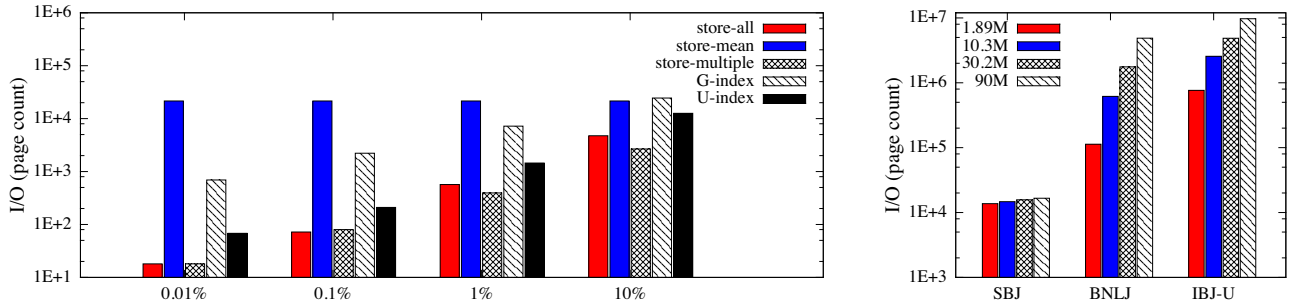
```
SELECT R.objID, R.rowc, R.colc, R.psfMag_u,
   R.psfMag_g, R.psfMag_r, R.psfMag_i, R.psfMag_z,
   R.extinction_u, R.extinction_g, R.extinction_r,
   R.extinction_i, R.extinction_z, S.objID, S.rowc,
   S.psfMag_u, S.psfMag_g, S.psfMag_r, S.psfMag_i,
   S.psfMag_z, S.extinction_u, S.extinction_g,
   S.extinction_r, S.extinction_i, S.extinction_z
FROM PhotoObj_0 as R, PhotoObj as S
WHERE R.rowc>a1 and R.rowc<b1 and R.colc>a2
     and R.colc<b2 and |R.rowc-S.rowc|<1
     and |R.colc-S.colc|<1
```

Analysis of indexing schemes: As mentioned previously, we consider two state-of-the-art indexing techniques for uncertain data in comparison. G-index[31] is designed for uncertain data modeled by (multivariate) Gaussian distributions. It consists of n two-dimensional R-trees for n-dimensional datasets, one R-tree per dimension. In each tree, tuples are clustered based on the mean and variance of the corresponding dimension, rather than the mean values of all dimensions. For datasets with $n = 1$, G-index returns exactly the true matches; for datasets with $n > 1$, the intersection of the candidates retrieved by all n R-trees forms a superset of the true matches. Note that G-index has great filtering power after the intersection, but each single tree actually touches many leaf nodes because it is not aware of the constraints on other dimensions. The above discussion suggests that G-index is not suitable for the I/O-bound query processing on multi-dimensional datasets. This analysis is also validated in Experiment 7.

U-index is a variant of R-tree on multi-dimensional uncertain data (e.g., x_loc and y_loc), with each node storing statistical information (i.e., probabilistically constrained rectangles and side lengths) to facilitate queries on uncertain data.

The page size is 4KB, which allows a fanout of 78 for G-index and 30 for U-index when U-catalog size is 3 (suggested in [41]). Table 5 shows, for each dataset, the number of index nodes that can be cached in memory, as well as a breakdown of non-leaf and leaf nodes. We see that all non-leaf nodes can be cached in memory.

I/O counts of index and subarray based schemes: Besides time measurements reported in previous experiments, we further show the I/O cost, measured in page counts, incurred in the SDSS case study. The page counts in Experiment 6 and Experiment 7 are shown in Fig. 11. The observations are consistent with Fig. 8(a) and Fig. 8(b) because the I/O cost dominates. These observations suggest that building R-tree based indexes for dimension attributes on top of arrays can not easily offer performance benefits because the dimension attributes in arrays naturally serve as the clustered indexes without having to pay the index I/Os.



(a) *Subarray* on SDSS when λ=0.9 with varied q (b) *Structure-Join* on SDSS when δ=1 and λ=0.9

Figure 11: I/O counts of *Subarray* and *Structure-Join* on SDSS datasets.

E. REAL-WORLD APPLICATIONS

We begin by explaining other scientific applications that can benefit from our work, besides SDSS used in our case study.

The most notable example is the Large Synoptic Survey Telescope (LSST) [28], which is currently in the design phase and marks the next-generation technology for digital sky surveys. It will ultimately store 55 PB of raw imagery, with astronomy-specific feature extraction to find observations of interesting astronomical objects (such as stars and galaxies). Since LSST adopts the array database design and the schema of the Digital Sloan Sky Survey (SDSS), our work is directly applicable to LSST.

Another application is severe weather monitoring [27], where scientists have developed distributed radar sensor networks for detecting hazardous weather events like tornados and severe storms. Recent work has developed a data cooking process to produce continuous probability distributions for key meteorological measures, e.g., wind velocity and reflexivity, for each voxel of the air in the Polar coordinate system that radar nodes use. Based on recent conversation with the scientists, they also plan to transform the cooked data into the Cartesian coordinate system so that data produced by different radar nodes can be aggregated. Due to the difference between Polar and Cartesian coordinate systems, the data transformation between them will also introduce location uncertainty regarding where the specific wind velocity and reflexivity were observed. Once the radar data is brought into the Cartesian system and represented using multi-dimensional arrays, our system can be used to address both value and position uncertainty.

Array databases are also gaining adoption in other disciplines including oceanography, atmospheric sciences, climatology, remote sensing, and seismology [9, 15]. Many of them involve locations of measurements and such locations can be uncertain due to device accuracy. As array databases emerge as an alternative to relational databases or spatial databases built on the relational model for managing scientific data, our work can be integrated into array databases to address position uncertainty and value uncertainty.

E.1 Coordinate Systems and Complex Queries

In real applications, the domain experts design the data cooking process and select the dimension attributes. Depending on the selected dimension attributes, the positions of tuples may not be modeled in the Cartesian coordinate system. However, once the domain experts choose the dimension attributes and the function to discretize each dimension attribute into index values, the logical structure of the array is determined as we define in §2.

Our *store-multiple* scheme takes the logical structure of the array as input and decides to place the (limited) tuple replicas with even spacing within the **logical** structure of the array, not within the actual physical space. As such, it does not need to make an assumption of the coordinate system, and is not restricted to the Cartesian coordinate system only.

Support of SDSS: Consider objects in SDSS. The attributes, $(rowc, colc)$, are the row and column center positions, which can serve as dimension attributes. In addition, the attributes, (ra, dec), for the right-ascension and declination in the spherical coordinate system can also be used as dimension attributes.

To compute neighbor pairs of objects in SDSS, a basic approach is to write the predicate as "$|\mathbb{A}.ra – \mathbb{B}.ra| < r \wedge |\mathbb{A}.dec – \mathbb{B}.dec| < r$" as shown in [23]. This is a *Structure-Join* of array $\mathbb{A}$ and $\mathbb{B}$ on dimension ra and dec within a "band" width r. It can be directly supported under the *store-multiple* scheme, as discussed in §4.

In coordinate systems other than the Cartesian, query predicates can also become more complex. Revisit the above example. Since the sphere is round, if the scientists require a more accurate evaluation, the predicate on ra needs to be corrected, for the fact that the right-ascension is "compressed" by $\cos(dec)$ as it moves away from the equator, to $|\mathbb{A}.ra - \mathbb{B}.ra| < r/|\cos(\mathbb{A}.dec) + \varepsilon|$ [23]. For this specific predicate, during the evaluation of the join, as we read each cell in the outer array $\mathbb{A}$, based on the range of dec it covers, we can relax the predicate by plugging in the bounds of $\cos(\mathbb{A}.dec)$. Then the only difference to the *Structure-Join* in §4 is that instead of having a fixed band width δ for all the outer cells, each outer cell will have its own band width computed based on the dec range it covers. The subarray-based joins can be executed with a modest change. Finally the retrieved tuples will be validated against the accurate predicate. For predicates that are not able or hard to be relaxed to a *Subarray* or a *Structure-Join*, to apply our techniques, a backup solution is to convert the original coordinate system to a new one where predicates can be directly written as or easily relaxed to them. The conversion between common coordinate systems is well studied and beyond the scope of this paper.

E.2 Cells versus Chunks

A chunk is the I/O unit. The chunk size in the real-world applications varies a lot and can be very skewed, e.g., the chunks for the Automatic Identification System data have a median size of 924B, with a standard deviation of 232MB [18]. Depending on the chunk size, a cell, which is the logical unit, can contain multiple chunks and multiple cells can be packed into one chunk as well.

There is no universally optimal chunk size. Selective queries can benefit from a relatively small chunk size by reducing the overhead of reading unnecessary data. On the other hand, a big chunk size can potentially reduce the random seeks for non-selective queries.

For both reasons, we do not make explicit assumptions on the chunk size. We believe the choice of the chunk size will not change our main results that are based on the design of logical structure. By default, we use the standard page size as the chunk size.