University of Massachusetts Amherst
ScholarWorks@UMass Amherst

Doctoral Dissertations | Dissertations and Theses

March 2018

Supporting Scientific Analytics under Data Uncertainty and Query Uncertainty

Liping Peng
University of Massachusetts - Amherst

Follow this and additional works at: https://scholarworks.umass.edu/dissertations_2

 Part of the Databases and Information Systems Commons

Recommended Citation

Peng, Liping, "Supporting Scientific Analytics under Data Uncertainty and Query Uncertainty" (2018). *Doctoral Dissertations*. 1206.
https://scholarworks.umass.edu/dissertations_2/1206

This Open Access Dissertation is brought to you for free and open access by the Dissertations and Theses at ScholarWorks@UMass Amherst. It has been accepted for inclusion in Doctoral Dissertations by an authorized administrator of ScholarWorks@UMass Amherst. For more information, please contact scholarworks@library.umass.edu.

SUPPORTING SCIENTIFIC ANALYTICS UNDER DATA UNCERTAINTY AND QUERY UNCERTAINTY

A Dissertation Presented

by

LIPING PENG

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 2018

College of Information and Computer Sciences

© Copyright by Liping Peng 2018

All Rights Reserved

SUPPORTING SCIENTIFIC ANALYTICS UNDER DATA UNCERTAINTY AND QUERY UNCERTAINTY

A Dissertation Presented

by

LIPING PENG

Approved as to style and content by:

Yanlei Diao, Chair

Anna Liu, Member

Benjamin Marlin, Member

Gerome Miklau, Member

James Allan, Chair
College of Information and Computer Sciences

ACKNOWLEDGMENTS

First, I want to say thank you to all the people who have supported me during my Ph.D. study. With the help and love I received from them, the challenges and tears in this journey turned to be rewards and happiness.

Foremost, I would like to express my sincere gratitude to my advisor, Professor Yanlei Diao, without whom the doctorate would not be possible. Her guidance, support and thoughtfulness were invaluable in my Ph.D. journey. I am also grateful for the dedication of Professor Anna Liu. I benefited a lot from her immense knowledge in statistics and her patience. I would also like to thank my other committee members, Professor Gerome Miklau and Professor Benjamin Marlin, who have given insightful comments on the work.

I am grateful to my colleagues and friends at UMass. I particularly thank Dr. Thanh T. L. Tran, from whom I learnt experiences that helped me start my Ph.D. journey easily. I would like to thank Enhui Huang (Ecole Polytechnique), Yuqing Xing, Kyriaki Dimitriadou (Brandeis University) and Wenzhao Liu for the close collaboration on research. I also want to thank all the members of the database group for sharing thoughts without reservation and making the lab an enjoyable place to work.

I would like to thank my mentors and colleagues at IBM Almaden Research Center, Dr. Andrey Balmin, Dr. Yannis Sismanis, Dr. Vuk Ercegovac and Dr. Peter J. Haas. Their guidance and support made my summer internship fruitful and memorable.

Finally, I want to express my heartfelt appreciation to my dear family, my father Shangze Peng, my mother Anju Yang, my sister Lijun Peng, and my husband Boduo Li. They stand by me through the times good and bad. I can always feel their love, which has encouraged me to keep moving forward.

ABSTRACT

SUPPORTING SCIENTIFIC ANALYTICS UNDER DATA UNCERTAINTY AND QUERY UNCERTAINTY

FEBRUARY 2018

LIPING PENG

B.Sc., HARBIN INSTITUTE OF TECHNOLOGY

M.Sc., HARBIN INSTITUTE OF TECHNOLOGY

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Yanlei Diao

Data management is becoming increasingly important in many applications, in particular, in large scientific databases where (1) data can be naturally modeled by continuous random variables, and (2) queries can involve complex predicates and/or be difficult for users to express explicitly. My thesis work aims to provide efficient support to both the "data uncertainty" and the "query uncertainty".

When data is uncertain, an important class of queries requires query answers to be returned if their existence probabilities pass a threshold. I start with optimizing such threshold query processing for continuous uncertain data in the relational model by (*i*) expediting selections by reducing dimensionality of integration and using faster filters, (*ii*) expediting joins using new indexes on uncertain data, and (*iii*) optimizing a query plan using a dynamic, per-tuple based approach. Evaluation results using real-world data and

benchmark queries show the accuracy and efficiency of my techniques and the dynamic query planning has over 50% performance gains in most cases over a state-of-the-art threshold query optimizer and is very close to the optimal planning in all cases.

Next I address uncertain data management in the array model, which has gained popularity for scientific data processing recently due to performance benefits. I define the formal semantics of array operations on uncertain data involving both *value uncertainty* within individual tuples and *position uncertainty* regarding where a tuple should belong in an array given uncertain dimension attributes, and propose a suite of storage and evaluation strategies for array operators, with a focus on a novel scheme that bounds the overhead of querying by strategically placing a few replicas of the tuples with large variances. Evaluation results show that for common workloads, my best-performing techniques outperform baselines up to 1 to 2 orders of magnitude while incurring only small storage overhead.

Finally, to bridge the increasing gap between the fast growth of data and the limited human ability to comprehend data and help the user retrieve high-value content from data more effectively, I propose to build interactive data exploration as a new database service, using an approach called "explore-by-example". To build an effective system, my work is grounded in a rigorous SVM-based active learning framework and focuses on the following three problems: (*i*) accuracy-based and convergence-based stopping criteria, (*ii*) expediting example acquisition in each iteration, and (*iii*) expediting the final result retrieval. Evaluation results using real-world data and query patterns show that my system significantly outperforms state-of-the-art systems in accuracy (18x accuracy improvement for 4-dimensional workloads) while achieving desired efficiency for interactive exploration (2 to 5 seconds per iteration).

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Overview

Data management problems are crucial in large-scale scientific applications such as severe weather monitoring [58, 98], computational astrophysics [92], and asteroid threat detection [29]. Recent studies [29, 91, 92] show that almost all scientific data are noisy and uncertain. Take the *massive astrophysical surveys* as an example: The observations are inherently noisy as the objects can be too dim to be recognized in the captured images, and repeated observations of objects are made to derive continuous probability distributions for uncertain attributes, e.g., the location and luminosity of objects, in the data cooking process. Such surveys are expected to enable real-time detection of transient events and anomalies as well as long-term tracking of objects of interest. Therefore, capturing uncertainty in data processing, from data input to query output, has become a key issue in scientific data management.

Besides "data uncertainty" described above, "query uncertainty" has also become a data management issue in large-scale scientific applications. For example, in the face of an extremely large scientific database, such as the Large Synoptic Survey Telescope (LSST) [63] and the Sloan Digital Sky Survey (SDSS) [92], a user may not be able to express her data interests precisely and there is a strong need to support "interactive data exploration" which can navigate users through a subspace in large-scale scientific data sets and retrieve data relevant to the user interest.

To address both data uncertainty and query uncertainty, I propose novel techniques to provide efficient query processing for the following three problems: data uncertainty in

the relational model (Section 1.2), data uncertainty in the array model (Section 1.3), and interactive data exploration (Section 1.4). The challenges of and our contributions towards each problem are elaborated in the next three sections.

1.2 Supporting Data Uncertainty in Relational Databases

The SDSS benchmark [92] shows the following characteristics of the uncertain data management problem:

Continuous uncertain data. Most attributes that resulted from scientific measurements or their data cooking processes are uncertain. These attributes are naturally modeled by continuous random variables. Gaussian distributions are the most commonly used distributions [91, 92] while more complex distributions such as asymmetric and bimodal distributions can be useful in special domains such as tornado detection [98]. As a concrete example, the Galaxy table in the SDSS archive has 297 attributes, out of which 151 attributes are uncertain. We illustrate the schema of the SDSS data set in Table 1.1.

name	**type**	**description**
OBJ_ID	bigint	SDSS identifier with [run, ..., field, obj]
...	...	
(*rowc*, *rowc_err*)	real	(row center position, error term)
(*colc*, *colc_err*)	real	(column center position, error term)
(*q_u*, *qErr_u*)	real	(stokes Q parameter, error term)
(*u_u*, *uErr_u*)	real	(stokes U parameter, error term)
(*ra*, *dec*, *ra_err*, *dec_err*, *ra_dec_corr*)	real	(right ascension, declination, error in ra, error in dec, ra/dec correlation)
...	...	

Table 1.1: Schema of the Galaxy table in the Sloan Digital Sky Survey (SDSS). Attributes in italics are uncertain.

Complex selection and join predicates. An important class of queries employs a Select-From-Where SQL block using a wide variety of predicates. Let us consider the following two queries from the SDSS benchmark [92], where the attributes in the lower case are uncertain attributes:

```
Q1:  SELECT *
     FROM Galaxy G
     WHERE G.r < 22
     AND G.q_r^2+G.u_r^2 > 0.25;
```

```
Q2:  SELECT *
     FROM Galaxy AS G1, Galaxy AS G2
     WHERE G1.OBJ_ID < G2.OBJ_ID
     AND |(G1.u-G1.g)-(G2.u-G2.g)| < 0.05
     AND |(G1.g-G1.r)-(G2.g-G2.r)| < 0.05
     AND (G1.rowc-G2.rowc)^2+(G1.colc-G2.colc)^2<1E4;
```

Queries Q1 and Q2 involve four types of predicates: (*i*) predicates on deterministic attributes, (*ii*) range predicates on a single uncertain attribute, e.g., the first selection predicate in Q1; (*iii*) multivariate linear predicates on uncertain attributes, e.g., the second and third join predicates in Q2; (*iv*) multivariate quadratic predicates on uncertain attributes, e.g., the last selection predicate in Q1 and the last join predicate in Q2. Each query can have an arbitrary mix of these types of predicates.

Efficient processing of threshold queries. Given uncertainty of input data, the user would want to retrieve query answers of high confidence, reflected by high existence probabilities of these answers. A common practice is that the user specifies a threshold so that only those tuples whose existence probabilities pass the threshold are finally returned. In many scenarios, such threshold queries need to be processed efficiently, for instance, in *near real-time* to detect dynamic features, transient events, and anomalous behaviors, or with short delay to support *interactive analysis* where scientists issue explorative queries and wait for quick answers online.

In Chapter 2, we address efficient threshold query processing on continuous uncertain data. We support selection-join-projection queries with a threshold on the existence probabilities of query answers. We propose to optimize such query processing using a suite of

new techniques grounded in statistical theory and a new design of the query optimizer. Our contributions include:

Selections (Section 2.2): Selections with complex predicates on continuous uncertain data often involve high-dimensional integrals such as with Q1 and Q2. In this work, we propose optimizations to reduce dimensionality of integration. We further develop fast filters for a wide range of predicates which efficiently compute an upper bound of the probability that a tuple satisfies the predicates. Hence, these filters can be used to prune tuples quickly.

Joins (Section 2.3): Joins on continuous uncertain data have traditionally used the strategy of a cross-product followed by a selection. This strategy can be highly inefficient as it may generate a large number of intermediate tuples. A join index can potentially prune many intermediate tuples. However, the design of a join index for continuous uncertain data is challenging because the index not only stores continuous distributions, but also takes a search condition based on the distribution from the probing tuple, which is not deterministic, and returns all candidates that can potentially produce join results that pass the threshold filter. The (only) relevant type of join index [23, 24] is based on primitive statistical results and has limited filtering power. We propose several new indexes based on much stronger statistical results and support a range of join predicates.

Query optimization (Section 2.4). Query optimization for continuous uncertain data fundamentally differs from traditional query optimization because selectivity becomes a property of each tuple that carries a distribution. Depending on the attribute distribution, the optimal plan for one tuple can be bad for another tuple. This change dictates per-tuple based query planning. Furthermore, in data stream systems the selectivity of operators cannot be estimated until the tuple arrives, and the selectivity of post-join operators cannot be estimated until the join results with new joint distributions are produced. Hence, selectivity estimation and query planning need to be performed during query execution. We design a query optimizer that supports such dynamic, per-tuple based planning at a low cost.

Evaluation (Section 2.5). Using real data and benchmark queries from SDSS, we demonstrate the accuracy and efficiency of our join indexes, selection filters, and optimization technique. Our results further demonstrate remarkable performance gains over the state-of-the-art X-BOUND join indexes [23, 24] and optimizer for threshold queries [73]:(*i*) For selective queries, the devised filters can drop many non-viable tuples with negligible time cost compared to expensive integration cost. In our experiments on 2-dimensional queries, the performance gain is up to 66x for rectangular queries and 400x for circular queries. (*ii*) Our indexes on continuous uncertain data modeled by Gaussian distributions returns exactly the true match, which offers 1.4x$\sim$7.1x performance gain over X-BOUND in the stream setting and 14x$\sim$570x gain in the disk setting. (*iii*) Our query planner gets over 50% performance gains over the state-of-the-art query planner in most cases and is very close to the optimal planning in all cases.

1.3 Supporting Data Uncertainty in Array Databases

For supporting scientific applications, relational technology has proven useful in some applications like SDSS [92]. However, there is a recent realization that most scientific data naturally reside in multi-dimensional arrays rather than relations. This is because most scientific data are produced to characterize physical phenomena that rely heavily on the notions of "adjacency" and "neighborhood" in a multi-dimensional space. Hence, array databases have recently been developed for scientific data processing [18, 29, 90]. Besides convenient expression of array operations, array databases also offer remarkable performance benefits over relational databases [88]. In particular, the new chunk-based storage scheme enables better alignment of logical locality (i.e., objects close in the logical array) and physical locality (i.e., objects close to each other are likely to be stored in the same chunk). Since many array operations exploit logical locality of data, e.g., finding objects close to a location, their associated physical locality can lead to significant I/O savings.

The increasing popularity of array data management has significant implications on uncertain data management: Recent work on multidimensional arrays [43, 42] 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) [92] 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 [90]. 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 Chapter 3.

In Chapter 3, 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 [18, 90], 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 to fully exploit the locality benefit of array databases and bound the overhead of querying. More specifically, our contributions include:

Semantics (Section 3.1). We define the formal semantics of array operations on uncertain data involving both position and value uncertainty. 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.

Subarray (Section 3.2). We provide native support for its operation on arrays with uncertain dimension attributes. 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.

Structure-Join (Section 3.3). We propose a new evaluation strategy, called the subarray-based join (SBJ), which works without a pre-built 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.

Evaluation (Section 3.4). We evaluate our techniques using both synthetic workloads and SDSS. 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*.

1.4 Explore-by-Example for Interactive Data Exploration

Today data is being generated at an unprecedented rate, so much that 90% of the data in the world has been created in the past two years[1]. However, the human ability to comprehend data remains as limited as before. As such, the Big Data era is presenting us with an increasing gap between the growth of data and the human ability to comprehend data. Consequently, there has been a growing demand of data management tools that can bridge this gap and help the user retrieve high-value content from data more effectively.

To respond to such needs, we build a new database service for interactive exploration in a framework called "*explore-by-example*". In this service, the database system requests user feedback on strategically collected database samples through a series of "conversations" (or iterations). In each iteration, the user characterizes a database sample as relevant or irrelevant to her interest. The user feedback is incorporated into the system to build a *user interest model*. The model is then used in the next iteration to steer the user towards a new area in the data space, and further improved using the user label of a new sample from that area. Eventually, the model characterizing the relevant objects is turned into a *user interest query* that will retrieve all relevant objects from the database. This service can be used to support two types of applications:

Ad-Hoc Exploration: Consider an example that a novice scientist comes to explore a large sky survey database such as SDSS [92]. She may not be able to express her data

[1]See a recent survey at https://www-01.ibm.com/software/data/bigdata/

interest precisely. Instead, she may prefer to navigate through a region of the sky, see a few samples of sky objects, provide yes or no feedback, and ask the database system to find more objects relevant to her interest. Such Ad-hoc exploration tasks are constrained by the amount of feedback that a user is willing to provide. Instead of requiring 100% accuracy of the user interest model, they often prefer a quick improvement of the accuracy with the first few dozens of samples that the user has reviewed.

Precise Exploration: In this setting, the user is engaged in a long-term conversation with the database system with explicit or implicit feedback on database objects. *Systematic reviews* are an example of a comprehensive assessment of the totality of evidence to address a question such as the effect of a treatment at a given time on mortality. Such reviews involve repeated querying of the clinical trial database and classifying the retrieved trials as relevant or not; some may take a year to complete. System-aided exploration that records user-reviewed trials continually may derive a model of the relevant trials more quickly than manual exploration by the user.

Across both applications the performance goals of explore-by-example include: (a) *Accuracy*: the system must maximize the accuracy of the user interest model with a limited amount of user feedback, or minimize the total user feedback needed to achieve a high accuracy level. (b) *Interactive performance*: the time cost in each iteration of exploration must be kept under a few seconds as the user may be waiting online for the next sample to review.

Our framework closely integrates model learning (from labeled examples thus far), space exploration (deciding new data areas to explore), and efficient example acquisition and final result retrieval (from large underlying databases). We choose Support Vector Machines (SVMs) to build the classification model because they can handle both linear and non-linear patterns. For space exploration, active learning theory is particularly helpful in deciding which database example, across all unlabeled objects in the database, for the user to label next in order to quickly improve the model accuracy. Recent active learning for SVM [16]

proposed to choose the example closest to the decision boundary of the current SVM model. Grounded on the SVM active learning theory, we made the following contributions in Chapter 4:

Optimizing SVM active learning for convex queries (Section 4.2). We develop a new algorithm which focuses on a common class of user interest queries with a convex shape in the data space. For such queries, we propose a novel algorithm, called TSM, which augments SVM active learning theory with a polytope-based partitioning function of the data space. TSM reduces user labeling from using active learning theory alone, which means achieving at least the same accuracy with a smaller number of user-labeled examples.

Stopping Criteria (Section 4.2). We propose the following two stopping criteria. (1) *Accuracy-based*: Besides improving the accuracy over SVM learning, TSM also enables us to prove a monotonically increasing lower bound of F1-score over iterations. This is the first formal result on the model accuracy based on F1-score without requiring a user-labeled test set, to the best of our knowledge, and enables either the system to develop an accuracy-based stopping criterion in the precise exploration scenario, or the user to decide on termination based on a quantitive estimate in the ad-hoc exploration scenario. (2) *Convergence-based*: For general query patterns, we capture the model change rate and the trend of this measure over recent iterations. It allows the system to provide a qualitative statement of whether the model has exhibited the trend for convergence.

Optimizations on retrieval (Section 4.3). We devise novel decision-tree based approximation to reduce the time cost of retrieving the most uncertain example from the database and an index-based optimization to reduce the running time of the final query to retrieve all relevant objects.

Evaluation (Section 4.4). Evaluation using real datasets and queries from Sloan Digital Sky Survey [92] shows the following results: (1) For convex queries, our TSM algorithm shown to offer a lower-bound for F1-score, and reduce the user labeling effort. On 2-dimensional workloads, when TSM is applied, the user only needs to label 20% of the

examples needed using traditional active learning without the TSM technique to achieve the same accuracy. On 4-dimensional workloads, the number is 80%. This means that if the database system offers query templates for the user to choose, then for known convex templates, it can choose TSM for the above benefits. (2) For general queries, our change rate metric can be used to detect model convergence. (3) We further compared our system to two state-of-the-art systems for explore-by-example, AIDE [34, 35] and LifeJoin [26]. Our system significantly outperforms these two in accuracy when the user interest involves 4 dimensions or above, considering both ad-hoc exploration (with limited, say 100, user labeled samples) and precise exploration (with up to 500 user labeled samples), while maintaining the per-iteration time within 2 to 5 seconds and the final retrieval time within a few minutes.

1.5 Thesis Organization

As is mentioned earlier, due to the intrinsic uncertainty in the measurements in scientific applications as well as in other data management applications, we address the problem of supporting data uncertainty in the relational model and in the array model in Chapter 2 and Chapter 3. In Chapter 4, we solve the problem of supporting query uncertainty when users cannot explicitly express his interests but are able to provide a binary judgement to each point in the data space. In Chapter 5, we summarize this thesis and discuss furture research directions.

CHAPTER 2

SUPPORTING DATA UNCERTAINTY IN RELATIONAL DATABASES

In this chapter, we address efficient threshold query processing on uncertain data modeled by continuous random variables. We support selection-join-projection queries with a threshold on the existence probabilities of query answers. We start with our data model (Section 2.1), and next propose a suite of new techniques grounded in statistical theory to optimize selections (Section 2.2) and joins (Section 2.3), and finally design a query optimizer (Section 2.4) that supports dynamic, per-tuple based planning at a low cost.

2.1 Background

In this section, we present a data model for probabilistic query processing on continuous uncertain data. This model provides a technical context for our discussion in later sections.

Probability distributions. A Gaussian Mixture Model (GMM) describes a probability distribution using a convex combination of Gaussian distributions. A multivariate Gaussian Mixture Model (multivariate GMM) naturally follows from the definition of multivariate Gaussian distributions. They are formally defined as follows:

Definition 1. *A Gaussian Mixture Model (GMM) for a continuous random variable X is a mixture of m Gaussian variables $X_1, X_2, \cdots, X_m$. The probability density function (pdf) of X is:*

$$f_X(x) = \sum_{i=1}^{m} p_i f_{X_i}(x),$$
$$f_{X_i}(x) = \frac{1}{\sigma_i\sqrt{2\pi}} e^{-\frac{(x-\mu_i)^2}{2\sigma_i^2}} \qquad (X_i \sim N(\mu_i, \sigma_i^2)),$$

where $0 \leq p_i \leq 1$, $\sum_{i=1}^{m} p_i = 1$, *and each mixture component is a Gaussian distribution with mean* μ_i *and variance* σ_i^2.

Definition 2. *A multivariate Gaussian Mixture Model (multivariate GMM) for a random vector* $\mathbf{X}$ *naturally follows from the definition of multivariate Gaussian distributions:*

$$f_{\mathbf{X}}(\mathbf{x}) = \sum_{i=1}^{m} p_i f_{\mathbf{X}_i}(\mathbf{x}),$$
$$f_{\mathbf{X}_i}(\mathbf{x}) = \frac{1}{(2\pi)^{k/2}|\Sigma_i|^{1/2}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_i)^T \Sigma_i^{-1}(\mathbf{x}-\boldsymbol{\mu}_i)} \quad (\mathbf{X}_i \sim N(\boldsymbol{\mu}_i, \Sigma_i)),$$

where k *is the random vector size, and each mixture component is a* k*-variate Gaussian with mean* $\boldsymbol{\mu}_i$ *and covariance matrix* Σ_i.

GMMs offer two key benefits to uncertain data management: First, theoretical results have shown that GMMs can approximate any continuous distribution arbitrarily well [44]. Hence, they are suitable for modeling complex real-world distributions. Second, GMMs allow efficient computation of relational operators based on Gaussian properties and advanced statistical theory as shown in our prior work [98] and later sections in this chapter.

Data model. We consider an input data set that follows the schema $\mathbf{A}^d \cup \mathbf{A}^p$. The attributes in $\mathbf{A}^d$ are deterministic attributes, like those in traditional databases. The attributes in $\mathbf{A}^p$ are continuous-valued uncertain attributes, such as the location of an object and the luminosity of a star. In each tuple, $\mathbf{A}^p$ is modeled by a vector of continuous random variables, $\mathbf{X}$, that has a joint pdf, $f_{\mathbf{A}^p}(\mathbf{x})$. According to the schema, $\mathbf{A}^p$ can be partitioned into independent groups of correlated attributes. Each group of correlated attributes can be modeled by a (multivariate) GMM denoted by $f_j(\mathbf{x}_j)$. Then the joint distribution for $\mathbf{A}^p$ can be written as $\prod_j f_j(\mathbf{x}_j)$. For simplicity, we use $\mathbf{A}$ to refer to uncertain attributes when our discussion focuses on uncertain attributes only. To address inter-tuple correlation in our data model, we adopt the use of history to capture dependencies among attribute sets as a result of prior database operations [86]. The history, $\mathbf{H}$, of an attribute set is defined as follows: (1) For a newly inserted tuple t, $\mathbf{H}(t.\mathbf{A}) = t.\mathbf{A}$. (2) If a new set of attributes, $\bar{t}.\bar{\mathbf{A}}$,

is derived from multiple attribute sets, $\{t_i.\mathbf{A_i}|i = 1,2,\ldots\}$, via a database operation, then $\mathbf{H}(\bar{t}.\bar{\mathbf{A}}) = \cup\mathbf{H}(t_i.\mathbf{A_i})$. That is, the history of the new attribute set includes the base pdf's that can be used to derive the joint pdf of this set of attributes. Finally, if two attribute sets intersect, they become correlated. Then a joint distribution of the two sets can be computed from their histories to capture correlation.

2.2 Optimizing Threshold Selection

In this section, we consider probabilistic threshold selections over a relation on a set of continuous uncertain attributes. Our goal is to support efficient evaluation of such selections, especially when they contain complex predicates.

Definition 3 (Probabilistic Threshold Selection)**.** *A probabilistic threshold selection, $\sigma_{\theta,\lambda}$, over a relation T is defined as:*

$$\sigma_{\theta,\lambda}(T) = \{t \mid Pr[R_\theta(\mathbf{X})] \geq \lambda, t \in T\},$$

where θ is the selection condition on continuous uncertain attributes $\mathbf{A}$, λ is the probability threshold, and R_θ is the selection region defined as $\{\mathbf{a}|\mathbf{a} \in \mathbb{R}^{|\mathbf{A}|} \wedge \theta(\mathbf{a}) = true\}$. For each tuple t, $\mathbf{X}$ is the random vector for $t.\mathbf{A}$, and $Pr[R_\theta(\mathbf{X})]$ is the probability for $\mathbf{X}$ to satisfy the selection condition, i.e., $Pr[R_\theta(\mathbf{X})] = \int_{R_\theta} f_{\mathbf{X}}(\mathbf{x})\mathrm{d}\mathbf{x}$.

A basic evaluation strategy for the selection follows the definition above, using an integral of the joint attribute distribution $f_{\mathbf{X}}$ for each tuple. For instance, the WHERE clause in Q2 (in Section 1.2) specifies a condition θ involving ten uncertain attributes. Assume that a cross-product is first performed. Then the selection with condition θ involves a ten-dimensional integral for each tuple.

An improvement is to factorize this integral into lower dimension integrals based on independence [86]. Suppose that the schema indicates that the uncertain attribute set $\mathbf{A}$ can be partitioned into attribute groups that are independent of each other. Denote this partitioning

using a set system $S = \{\mathbf{A}_1, \mathbf{A}_2, \ldots \mathbf{A}_G\}$. Then consider each predicate in the condition θ: if the predicate involves attributes from different groups, merge these groups into one. After doing so for all predicates, we obtain a new set system $S' = \{\mathbf{A'}_1, \mathbf{A'}_2, \ldots \mathbf{A'}_{G'}\}$. Now we can rewrite the big integral as the product of the integrals for the attribute groups in S'. Revisit Q2. The SDSS schema shows that $S = \{\{G_1.u\}, \{G_1.g\}, \{G_1.r\}, \{G_1.rowc, G_1.colc\}, \{G_2.u\}, \{G_2.g\}, \{G_2.r\}, \{G_2.rowc, G_2.colc\}\}$. The predicates in WHERE yields $S' = \{\{G_1.u, G_1.g, G_1.r, G_2.u, G_2.g, G_2.r\}, \{G_1.rowc, G_1.colc, G_2.rowc, G_2.colc\}\}$. Hence for each tuple, we will perform a 6-dimensional integral plus a 4-dimensional integral.

As can be seen, the basic evaluation approach can be expensive or even intractable when the integral has a high dimensionality and a complex shape of the selection region. In this section, we propose two classes of optimization techniques grounded in statistical theory: the first class reduces the dimensionality of integration, while the second class efficiently filters (most of) tuples whose probabilities fall below the threshold without using integrals.

2.2.1 Reducing Dimensionality of Integration

We first propose to reduce the dimensionality of integration by leveraging the following result [75]:

Linear Transformation: Let $\mathbf{X} \sim N_k(\boldsymbol{\mu}, \Sigma)$. For a given $l \times k$ matrix $\mathbf{B}$ of constants and a l-dimensional vector $\mathbf{b}$ of constants, $\mathbf{Y} = \mathbf{B}\mathbf{X} + \mathbf{b} \sim N_l(\mathbf{B}\boldsymbol{\mu} + \mathbf{b}, \mathbf{B}\Sigma\mathbf{B}^T)$.

The result states that a linear transformation of multivariate normal random vector still has a multivariate normal distribution. It is natural to extend linear transformation to a GMM: we simply perform a linear transformation of each mixture component separately.

To apply the above result, given a selection condition θ, we define a transformed selection region $R'_\theta = \{\mathbf{y} | \mathbf{y} = \mathbf{B}\mathbf{x} + \mathbf{b} \wedge \mathbf{x} \in R_\theta\}$. If there exists a transformation matrix $\mathbf{B}_{l \times k}$ ($l < k$) such that $Pr[R_\theta(\mathbf{X})] = Pr[R'_\theta(\mathbf{Y})]$, we can reduce the dimensionality of integration, i.e.,

$$\int_{R_\theta} f_{\mathbf{X}}(\mathbf{x})\, \mathrm{d}\mathbf{x} = Pr[R_\theta(\mathbf{X})] = Pr[R'_\theta(\mathbf{Y})] = \int_{R'_\theta} f_{\mathbf{Y}}(\mathbf{y})\, \mathrm{d}\mathbf{y}. \tag{2.1}$$

Given a condition θ on a set of continuous uncertain attributes, we can construct $\mathbf{B}$ and $\mathbf{b}$ by taking the following steps: (1) Partition attributes into independent groups based on the schema and θ, as described at the beginning of the section. (2) For each group of attributes, define a random vector $\mathbf{X}$. Find maximum linear subexpressions relevant to $\mathbf{X}$ from θ, and denote them using a new vector $\mathbf{Y}$. Rewrite each variable in $\mathbf{Y}$ as the product of a row vector and $\mathbf{X}$, plus a constant. Let $\mathbf{B}$ be the matrix that contains all the row vectors and $\mathbf{b}$ be the column vector that contains all the constants. (3) If $\mathbf{B}$ does not have full row rank, remove rows from $\mathbf{B}$, one at a time, until it has full row rank. Remove elements from $\mathbf{b}$ accordingly. Below we show the correctness of the procedure.

Denote the matrix returned as $\mathbf{B}_{l\times k}$. Since it has full row rank, $l \leq k$. If $l < k$, we can apply Eq. (2.1) to transform the integration from the space for $\mathbf{X}$ to that for $\mathbf{Y}$; If $l = k$, linear transformation does not help to reduce the dimensionality of integration.

Example 2.2.1. *For Q2, we obtain two independent groups of attributes after step* (1)*, which is the factorization described at the beginning of the section. In step* (2)*, let us consider the first group:* $S_1' = \{G_1.u, G_1.g, G_1.r, G_2.u, G_2.g, G_2.r\}$*. Let* $\mathbf{X}$ *be the random vector for* S_1'*. There are two maximum linear subexpressions in* θ *for* S_1'*. Let* $y_1 = (G1.u - G1.g) - (G2.u - G2.g)$*,* $y_2 = (G1.g - G1.r) - (G2.g - G2.r)$*. Then, we have*:

$$\mathbf{Y} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 0 & -1 & 1 & 0 \\ 0 & 1 & -1 & 0 & -1 & 1 \end{pmatrix} \mathbf{X} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \mathbf{B}\mathbf{X} + \mathbf{b}$$

Since $\mathbf{B}$ *has full row rank, step* (3) *is omitted. We can get the pdf of* $\mathbf{Y}$ *using linear transformation from* $\mathbf{X}$*. Finally based on Eq* (2.1)*, the integral dimensionality is reduced from 6 to 2:*

$$\int_{R_\theta} \cdots \int \prod_{i=1}^{6} \left(f_{X_i}(x_i)\, \mathrm{d}x_i \right) = \int_{-0.05}^{0.05} \int_{-0.05}^{0.05} f_{\mathbf{Y}}(y_1, y_2)\, \mathrm{d}y_1\, \mathrm{d}y_2$$

2.2.2 A Filtering Framework without Integrals

Although linear transformation can improve performance by decreasing the dimensionality of integration, it still requires the use of one integral for each tuple. In this section, we propose a filter operator, $\tilde{\sigma}$, that computes an upper bound ($\tilde{p}$) of the true probability (p) that a tuple satisfies the selection condition without using integrals. When such upper bounds are tight enough, most tuples that fail to pass the threshold selection can be removed by the filter $\tilde{p} < \lambda$. In (rare) cases that $p < \lambda \leq \tilde{p}$, the original selection operator (σ) with exact integration is needed to compute the true probability. Hence, a key issue in designing the filter is how to derive a tight upper bound at a cost much lower than the integration cost.

2.2.2.1 A General Filtering Technique

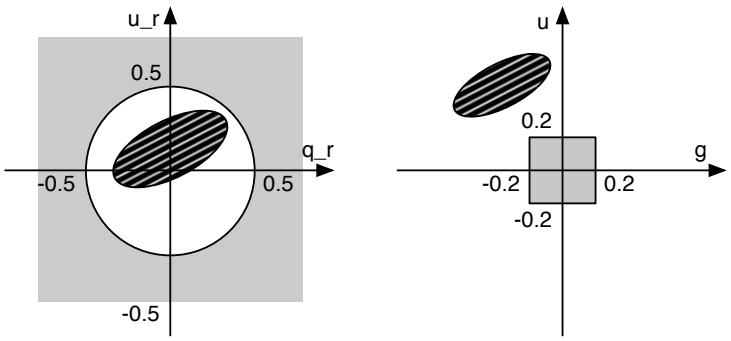
We first propose a general filtering technique that leverages the *multidimensional Chebyshev's inequality*, and explores its relationship with a selection region in a high-dimensional space. Let $\mathbf{X}$ be a k-dimensional random vector with expectation $\boldsymbol{\mu}$ and covariance matrix Σ. If Σ is an invertible matrix, then for any real number $a > 0$, multidimensional Chebyshev's inequality states that:

$$Pr[(\mathbf{X} - \boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{X} - \boldsymbol{\mu}) > a^2] \leq \frac{k}{a^2}. \tag{2.2}$$

To leverage the above result, we transform threshold selection evaluation into a geometric problem. Besides the predicate region $R_\theta \subseteq \mathbb{R}^k$ from Definition 3, we also define a geometric region specific to each given random vector $\mathbf{X}$ of size k and a threshold λ:

Definition 4 (Chebyshev region)**.** *A Chebyshev region $R_\lambda(\mathbf{X})$ for a given λ and a random vector $\mathbf{X}$ with mean $\boldsymbol{\mu}$ and variance Σ is:*

$$R_\lambda(\mathbf{X}) = \{\mathbf{x} \mid (\mathbf{x} - \boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu}) < \frac{k}{\lambda}\}. \tag{2.3}$$



(a) q_r^2+u_r^2>0.25 (b) $|u|<0.2$ and $|g|<0.2$

Figure 2.1: Illustration of the predicate region (shaded in gray) and a tuple's chebyshev region (shaded in stripes).

Geometrically, the Chebyshev region is an ellipse (in $\mathbb{R}^2$) or ellipsoid (in $\mathbb{R}^k$ where $k \geq 3$) centered at $\boldsymbol{\mu}$. According to Eq. (2.2), we can see that $Pr[R_\lambda(\mathbf{X})] > 1-\lambda$. That is, for the random vector $\mathbf{X}$, the Chebyshev region covers the probability mass of more than $1-\lambda$. Therefore, when the Chebyshev region $R_\lambda(\mathbf{X})$ for a given tuple does not overlap with the predicate region R_θ, it is easy to bound the probability mass of this tuple in the predicate region: $Pr[R_\theta(\mathbf{X})] \leq 1 - Pr[R_\lambda(\mathbf{X})] < \lambda$. We can then safely filter the tuple. As such, the threshold selection problem is transformed into the geometric problem of judging whether the predicate region and a tuple's Chebyshev region are disjoint in a k-dimensional space.

Example 2.2.2. *For a bivariate random vector* $\mathbf{X}$ *with mean* $\boldsymbol{\mu}$ *and covariance* Σ*, the Chebyshev region is a region bounded by an ellipse centering at* $\boldsymbol{\mu}$*, shown as the areas shaded in stripes in Figure 2.1(a) and Figure 2.1(b). The predicate "q_r^2+u_r^2>0.25" in Q1 is marked by the grey area outside the circle with center* $(0,0)$ *and radius* 0.5 *in Figure 2.1(a). The predicate region of "*$|u|<0.2$ *and* $|g|<0.2$*" is a square shown by the grey area in Figure 2.1(b).*

Detecting disjoint regions. The R_θ and $R_\lambda(\mathbf{X})$ regions are disjoint in the space $\mathbb{R}^k$ if they satisfy two conditions: (1) the center of $R_\lambda(\mathbf{X})$, which is $\boldsymbol{\mu}$, falls outside of R_θ; (2) the boundary of R_θ is outside $R_\lambda(\mathbf{X})$. When R_θ has a simple shape, e.g., a rectangle as shown

in Figure 2.1(b), condition (2) is satisfied if none of the edges intersects with the boundary of $R_\lambda(\mathbf{X})$ and the center of R_θ lies outside $R_\lambda(\mathbf{X})$. Generally, to test condition (2), we can minimize $(\mathbf{X}-\boldsymbol{\mu})^T\Sigma^{-1}(\mathbf{X}-\boldsymbol{\mu})$ on R_θ. If the minimum is larger than k/λ, condition (2) is satisfied. While constrained optimization in general can be a difficult problem, in many common cases it can be solved efficiently. For example, when the region R_θ is an intersection of ellipsoids, the constrained optimization becomes the so called Quadratically Constrained Quadratic Program (QCQP), which can be solved as easily as the linear programs [17]. When the boundary of R_θ can be readily defined by equalities, the minimization can be done on the boundary and solved using the *Lagrange multiplier*. For example, when R_θ is an ellipsoid, the Lagrange multiplier leads to a system of linear equations.

In summary, for those common predicates whose regions are of simple shapes or whose boundary can be defined by linear or quadratic equalities, our general technique based on the multidimensional Chebyshev's inequality can provide efficient filtering.

2.2.2.2 Fast Filters for Common Predicates

For several common types of predicates, we can devise fast filters for threshold selection evaluation by exploiting known statistical results. Consider the following predicates on the attribute set $\mathbf{A}$:

1. *One-dimensional*: $|\mathbf{A}| = 1$, $R_\theta = \bigcup_{i=1}^{n}(a_i, b_i)$, where $a_i > b_{i-1}$. An example is "$ra^2 > 1$" , whose selection region can be written as $(-\infty,-1)\cup(1,+\infty)$.

2. *Multi-dimensional quadratic forms*: $|\mathbf{A}|>1$, $R_\theta = \{\mathbf{a}|\mathbf{a}^T\Lambda\mathbf{a}\ op\ \ \delta\}$, where Λ is an $|\mathbf{A}|$-dimensional symmetric matrix, op is "$>$" or "$<$". An example is "$q\_r^2+u\_r^2 > 0.25$" in Q1, where Λ is the identity matrix.

3. Predicates that can be reduced to category (1) or (2) above by applying linear transformation. Consider "$|(G_1.u-G_1.g)-(G_2.u-G_2.g)| < 0.05$" in Q2. By letting $z=(G_1.u-G_1.g)-(G_2.u-G_2.g)$, we have "$|z|<0.05$", which belongs to category (1). Next consider "$(G_1.rowc-G_2.rowc)^2+(G_1.colc-G_2.colc)^2<4E6$" in Q2. With

$z_1 = G_1.rowc - G_2.rowc$ and $z_2 = G_1.colc - G_2.colc$, we have "$z_1^2 + z_2^2 < 4E6$", which belongs to category (2).

One-dimensional predicates. We exploit the following statistical results to devise fast filters.

Markov's inequality states that for a random variable X, $Pr[|X| \geq a] \leq E|X|/a$ for any real number $a > 0$. It can be applied to predicates $R_\theta = \bigcup_{i=1}^{n}(a_i, b_i)$, if the point 0 does not lie in the predicate region (otherwise we will get a trivial upper bound of value 1). We consider three cases that apply Markov's inequality based on the different relationships between the point 0 and the predicate region:

- $0 < a_1$: $Pr[R_\theta(X)] < Pr[(a_1, +\infty)] \leq E|X|/a_1$
- $b_n < 0$: $Pr[R_\theta(X)] < Pr[(-\infty, b_n)] \leq -E|X|/b_n$
- $b_{i-1} < 0 < a_i$: $Pr[R_\theta(X)] < Pr[(-\infty, b_{i-1})] + Pr[(a_i, +\infty)] \leq E|X|/\min\{-b_{i-1}, a_i\}$

The distribution of $|X|$ can be computed as follows: When X follows a GMM with m components, each identified by parameters (p_i, μ_i, σ_i^2), we have $E|X| = \sum_{i=1}^{m} p_i E|X_i|$, where $E|X_i| = \sigma_i\sqrt{2/\pi}\exp(-\mu_i^2/(2\sigma_i^2)) + \mu_i(1 - 2\Phi(-\mu_i/\sigma_i))$ and Φ is the *cdf* of a standard normal distribution.

Chebyshev's inequality and Cantelli's inequality: Chebyshev's inequality states that for a random variable X with expected value μ and standard deviation σ, $Pr[|X - \mu| \geq a\sigma] \leq 1/a^2$ for any real number $a > 0$. Cantelli's inequality, known as the one-sided version of Chebyshev's inequality, provides a tighter bound on each side of the distribution: $Pr[X \geq \mu + a\sigma] \leq 1/(1+a^2)$, $Pr[X \leq \mu - a\sigma] \leq 1/(1+a^2)$. Both inequalities can be applied to predicates $\bigcup_{i=1}^{n}(a_i, b_i)$ if μ does not reside in the predicate region. Again, we consider three cases below. In the first two cases, Cantelli's inequality gives a tighter upper bound. In the third last case, we need to compute upper bounds using both inequalities and choose the smaller one.

- $\mu < a_1$: $Pr[R_\theta(X)] < Pr[(a_1, +\infty)] \leq \dfrac{\sigma^2}{\sigma^2 + (a_1 - \mu)^2}$
- $b_n < \mu$: $Pr[R_\theta(X)] < Pr[(-\infty, b_n)] \leq \dfrac{\sigma^2}{\sigma^2 + (\mu - b_n)^2}$
- $b_{i-1} < \mu < a_i$: $Pr[R_\theta(X)] < Pr[(-\infty, b_{i-1})] + Pr[(a_i, +\infty)] \leq$

$$\min\left\{\frac{\sigma^2}{(\min\{\mu - b_{i-1}, a_i - \mu\})^2}, \frac{\sigma^2}{\sigma^2 + (\mu - b_{i-1})^2} + \frac{\sigma^2}{\sigma^2 + (a_i - \mu)^2}\right\}$$

Multi-dimensional quadratic forms. If $\mathbf{X}$ follows a GMM, its quadratic form $\mathbf{X}^T \Lambda \mathbf{X}$ yields a new random variable for which we can compute the mean and variance. This allows us to apply Chebyshev's inequality and Cantelli's inequality similarly as above.

More specifically, if $\mathbf{X}$ follows a GMM, its quadratic form $\mathbf{X}^T \Lambda \mathbf{X}$ yields a new random variable. We first derive the new distribution as follows: For $\mathbf{X} \sim N(\boldsymbol{\mu}, \Sigma)$, we have

$$E[\mathbf{X}^T \Lambda \mathbf{X}] = tr[\Lambda \Sigma] + \boldsymbol{\mu}^T \Lambda \boldsymbol{\mu}$$
$$Var[\mathbf{X}^T \Lambda \mathbf{X}] = 2tr[\Lambda \Sigma \Lambda \Sigma] + 4\boldsymbol{\mu}^T \Lambda \Sigma \Lambda \boldsymbol{\mu},$$

where $tr[\cdot]$ denotes the trace of a matrix.

For $\mathbf{X}$ follows a GMM with m components, each identified by $(p_i, \boldsymbol{\mu}_i, \Sigma_i)$ $(i = 1 \cdots m)$,

$$\begin{aligned}
E[\mathbf{X}^T \Lambda \mathbf{X}] &= \sum_{i=1}^{m} p_i E[\mathbf{X}_i^T \Lambda \mathbf{X}_i] \\
Var[\mathbf{X}^T \Lambda \mathbf{X}] &= E\left[\left(\mathbf{X}^T \Lambda \mathbf{X}\right)^2\right] - \left(E[\mathbf{X}^T \Lambda \mathbf{X}]\right)^2 \\
&= \sum_{i=1}^{m} p_i E\left[\left(\mathbf{X}_i^T \Lambda \mathbf{X}_i\right)^2\right] - \left(E[\mathbf{X}^T \Lambda \mathbf{X}]\right)^2 \\
&= \sum_{i=1}^{m} p_i \left(Var[\mathbf{X}_i^T \Lambda \mathbf{X}_i] + \left(E[\mathbf{X}_i^T \Lambda \mathbf{X}_i]\right)^2\right) - \left(E[\mathbf{X}^T \Lambda \mathbf{X}]\right)^2
\end{aligned}$$

Now suppose that the quadratic form $\mathbf{X}^T \Lambda \mathbf{X}$ yields a new random variable with mean μ_0 and variance σ_0^2. This allows us to apply Chebyshev's inequality and Cantelli's inequality as before. However, in the case of comparing a quadratic form with a constant, the predicate

contains only one interval: either $(-\infty, \delta)$ or $(\delta, +\infty)$. Hence, when u_0 lies outside the predicate region, Cantelli's inequality always gives a tighter bound. Formally,

- If the predicate region is $\mathbf{X}^T \Lambda \mathbf{X} < \delta$, when $\delta < \mu_0$:

  $Pr[R_\theta(\mathbf{X})] \leq \sigma_0^2 / \left(\sigma_0^2 + (\mu_0 - \delta)^2\right)$

- If the predicate region is $\mathbf{X}^T \Lambda \mathbf{X} > \delta$, when $\delta > \mu_0$:

  $Pr[R_\theta(\mathbf{X})] \leq \sigma_0^2 / \left(\sigma_0^2 + (\delta - \mu_0)^2\right)$

2.3 Optimizing Threshold Join

In this section, we consider probabilistic threshold joins of relation R and relation S on a set of continuous uncertain join attributes.

Definition 5 (Probabilistic Threshold Join)**.** *A probabilistic thre- shold join of R and S on continuous uncertain attributes $\mathbf{A}$ is:*

$$R \bowtie_{\theta,\lambda} S = \{(r,s) \mid Pr[R_\theta(\mathbf{X}_r, \mathbf{X}_s)] \geq \lambda, r \in R, s \in S\},$$

where θ is the join predicate, λ is the probability threshold, $\mathbf{X}_r$ and $\mathbf{X}_s$ are the random vectors for $r.\mathbf{A}$ and $s.\mathbf{A}$, R_θ is the predicate region in $\mathbb{R}^{2|\mathbf{A}|}$, and $Pr[R_\theta(\mathbf{X}_r, \mathbf{X}_s)]$ is the probability for $\mathbf{X}_r$ and $\mathbf{X}_s$ to satisfy the join condition.

When the input relations R and S are independent, $Pr[R_\theta(\mathbf{X}_r, \mathbf{X}_s)] = \iint_{R_\theta} f_{\mathbf{X}_r}(\mathbf{x}_r) f_{\mathbf{X}_s}(\mathbf{x}_s) \mathrm{d}\mathbf{x}_r \mathrm{d}\mathbf{x}_s$. If R and S tuples are correlated, we can compute the joint distribution using history [86].

A default evaluation strategy for the threshold join $R \bowtie_{\theta,\lambda} S$ is to perform a cross-product $R \times S$ followed by a threshold selection with the condition $\theta(R.\mathbf{A}, S.\mathbf{A})$ and the threshold λ. The cross-product can create a large number of intermediate tuples, hence highly inefficient. In this section, we propose new join indexes to implement a *filtered cross-product*, denoted by $R \times_{\theta,\lambda} S$, which returns a superset of true join results but a subset

of the cross-product results. Then $R \bowtie_{\theta,\lambda} S = \sigma_{\theta,\lambda}(R \times_{\theta,\lambda} S)$, that is, the true join results are produced by further applying a threshold selection.

Designing a join index for continuous random variables is much more difficult than its counterpart for deterministic values. Consider $R \bowtie_{R.A-S.A<\delta} S$, and we want to build an index on S. First, the design of the join index needs to answer two questions: (1) what is the search key of the index? (2) given a R tuple, how do we form a query region over the index? In a traditional database, $S.A$ has a deterministic value and naturally forms the search key of the index. Given a R tuple, the join predicate is instantiated with $R.A = v$, which naturally yields a query region, $S.A > v - \delta$, on the index. Now consider the join where $R.A$ and $S.A$ are random variables and each follows a distribution. To build an index on $S.A$, it is not clear which aspects of the distribution of $S.A$ can be used as the index key, and given an R tuple, how we use the distribution of $R.A$ to form the query region over the index.

Second, the index for probabilistic threshold join needs to take into account the probability threshold λ. For each tuple r in R, probing the index should return all those tuples s in S that can possibly satisfy $Pr[R_\theta(\mathbf{X}_r, \mathbf{X}_s)] \geq \lambda$, called *candidate tuples*. In other words, we want to ignore other tuples $\tilde{s}$ for which we know for sure $Pr[R_\theta(\mathbf{X}_r, \mathbf{X}_{\tilde{s}})] < \lambda$, hence improving performance.

Our main idea is that if we can find a necessary condition for $Pr[R_\theta(\mathbf{X}_r, \mathbf{X}_s)] \geq \lambda$, then the negation of the necessary condition identifies all those tuples $\tilde{s}$ that can be ignored in the index lookup. To improve the index's filtering power, we seek necessary and sufficient conditions if possible, or necessary conditions that are "tight" enough. Furthermore, to have real utility for index design, the necessary condition has to meet two requirements: (1) In the necessary condition, the quantities concerning S can be used to form the search key of a common index structure such as an R-tree [10]; (2) Given an R tuple, after the necessary condition is instantiated with all the quantities concerning R, it should yield a query region that can be easily tested for overlap with the index entries.

Existing join indexes for continuous uncertain attributes [23, 24] make simplifying assumptions about attribute distributions, and use a "loose" necessary condition in index design, resulting in poor performance as we will show in Section 2.5. Below we derive tighter necessary conditions for common join predicates, including a necessary and sufficient condition, and develop new indexes based on them.

2.3.1 Band Join of General Distributions

We start with the simple case of a single join attribute. A band join uses the predicate "$a < R.A - S.A < b$". Given a tuple r from relation R, we use X_r to denote the random variable of its join attribute, which follows a univariate GMM. Similarly, X_s denotes the random variable for the join attribute in an s tuple, again following a GMM. We denote the mean, variance, and *pdf* of X_t using μ_t, σ_t^2, $f_{\mu_t,\sigma_t^2}(x_t)$, respectively $(t = r, s)$.

As shown in Section 2.2.1, the linear transformation $Z = X_r - X_s$ can transform the original band-shaped integral region into a single interval:

$$Pr[a < Z = (X_r - X_s) < b] = \int_a^b f_{\mu_r-\mu_s,\sigma_r^2+\sigma_s^2}(z)\,\mathrm{d}z.$$

For a single variable Z, the following theorem provides **a necessary condition** for $Pr[a < Z < b] \geq \lambda$.

Theorem 2.3.1. *Given a range $[a, b]$, if a random variable Z with mean μ and variance σ^2 satisfies the condition $Pr[a < Z < b] \geq \lambda$, then $\mu + \sigma\sqrt{(1-\lambda)/\lambda} \geq a$ and $\mu - \sigma\sqrt{(1-\lambda)/\lambda} \leq b$.*

Since $Z = X_r - X_s \sim N(\mu, \sigma^2)$, plug $\mu = \mu_r - \mu_s$, $\sigma^2 = \sigma_r^2 + \sigma_s^2$ back to the inequalities in the theorem. Then we obtain a necessary condition for $Pr[a < X_r - X_s < b] \geq \lambda$:

$$\mu_r - \mu_s + \sqrt{\frac{1-\lambda}{\lambda}(\sigma_r^2 + \sigma_s^2)} \geq a, \tag{2.4a}$$

$$\mu_r - \mu_s - \sqrt{\frac{1-\lambda}{\lambda}(\sigma_r^2 + \sigma_s^2)} \leq b. \tag{2.4b}$$

Index Construction and retrieval. We now design an index on the S relation to provide efficient support for the filtered cross product $R \times_{a<R.A-S.A<b,\ \lambda} S$. In Eq. (2.4a) and (2.4b), μ_s and σ_s^2 are the quantities from relation S. We use them to form the *search key* of an index: for each tuple s, we insert the pair (μ_s, σ_s^2) together with the tuple id into an R-tree index [10]. This index essentially indexes points in a two-dimensional space in the leaf nodes and groups them into minimum bounding rectangles in non-leaf nodes. All existing R-tree construction methods can be used.

For each probing tuple r, the *query region* is naturally formed by instantiating μ_r and σ_r^2 in Eq. (2.4a) and (2.4b). However, this query region has a nonstandard shape, so we re-implement the *overlap* method in the R-tree, which returns True when a minimum bounding rectangle in a tree node, denoted by R_I, overlaps with the query region, denoted by R_Q. Let (x, y) denote the search key of the index, that is, $x = \mu_s$ and $y = \sigma_s^2$. Then R_I is a rectangle $[x_1, x_2; y_1, y_2]$. The query region R_Q has two conditions. By setting $x = \mu_s$ and $y = \sigma_s^2$ in Eq. (2.4a), we can rewrite the first condition as:

$$R_{Q1}\text{: } (1) x \le \mu_r - a, \text{ or}$$
$$(2) x > \mu_r - a \text{ and } y \ge \lambda(x - \mu_r + a)^2/(1-\lambda) - \sigma_r^2.$$

It is not hard to see that R_I overlaps with the union of region (1) and region (2) in R_{Q1} if its upper left vertex (x_1, y_2) lies in either region. We can rewrite the second condition from Eq. (2.4b) and develop the test condition in a similar way.

2.3.2 Band Join of Gaussian Distributions

We next consider the most common distributions, Gaussian distributions, for continuous random variables. The known Gaussian properties allow us to find **a sufficient and necessary condition** and hence design an index with better filtering power.

Theorem 2.3.2. *Given a range $[a, b]$, a normally distributed random variable $Z \sim N(\mu, \sigma^2)$ satisfies the condition $Pr[a < Z < b] \ge \lambda$ **iff** there exists an $\alpha \in (0, 1-\lambda)$ such that*

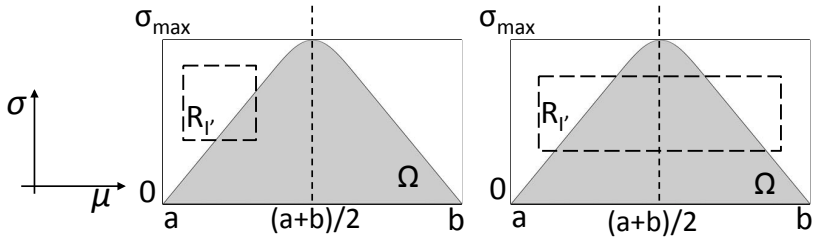


Figure 2.2: Two cases when R'_I and Ω overlap.

$a-\Phi^{-1}(\alpha)\sigma \leq \mu \leq b-\Phi^{-1}(\lambda+\alpha)\sigma$, *where* Φ^{-1} *is the inverse of the standard normal cdf* (*also called the quantile function*).

Given a range $[a,b]$ and a threshold λ, Theorem 2.3.2 essentially identifies all normally distributed random variables, i.e., all the (μ,σ) pairs, that satisfy $Pr[a<Z<b] \geq \lambda$. Let Ω denote this collection of (μ,σ). Formally,

$$\Omega(a,b,\lambda) = \bigcup_{0\leq\alpha\leq 1-\lambda} \left\{ (\mu,\sigma) \mid a-\Phi^{-1}(\alpha)\sigma \leq \mu \leq b-\Phi^{-1}(\lambda+\alpha)\sigma \right\} \tag{2.5}$$

The Ω region will play a key role in the index design, in particular, representing the query region. The shaded region in Figure 2.2 shows the shape of Ω when $\lambda = 0.7$, where the x and y axes denote μ and σ, respectively. In general, λ controls the shape of Ω, and the a and b values determine the stretch along both dimensions.

Index Construction and retrieval. We next present a new index that exploits the above sufficient and necessary condition and thus returns only the true matches for each probing tuple. Recall that the join predicate is "$a<R.A-S.A<b$", and X_r and X_s denote the join attribute of a r tuple and an s tuple. As in Section 2.3.1, we build an R-tree index on $S.A$: we take the mean μ_s and variance σ_s^2 of the variable X_s for each tuple and insert them as a pair to the R-tree.

Given each probing tuple r, we next design the query region over the R-tree. As before, consider two variables X_r and X_s, and let $Z = X_r - X_s$. Eq. (2.5) has defined all possible

distributions of Z that would satisfy $Pr[a < Z < b] \geq \lambda$. Now plug $\mu = \mu_r - \mu_s$ and $\sigma = \sigma_r^2 + \sigma_s^2$ into Eq. (2.5). Since for a particular probing tuple r, μ_r and σ_r are simply constants, Eq. (2.5) naturally yields a query region over all the distributions (μ_s, σ_s^2) in the R-tree.

Given the query region, the next task is to design the "overlap" routine that directs the search in the R-tree by comparing the query region (R_Q) with the minimum bounding rectangles (R_I) in each non-leaf node of the tree. However, the above query region has a complex shape and hence it is slow to test the overlap between R_Q and R_I. The first technique we use is to transform both R_Q and R_I to a different domain through a mapping. Letting $(x, y) = (\mu_s, \sigma_s^2)$ be the search key of the index, the mapping $\mathcal{F}$ has:

$$x' = \mu_r - x \text{ and } y' = \sqrt{\sigma_r^2 + y}$$

Each rectangle in the index $R_I = [x_1, x_2; y_1, y_2]$ is transformed to:

$$R_I' = \left[u_r - x_2, u_r - x_1; \sqrt{\sigma_r^2 + y_1}, \sqrt{\sigma_r^2 + y_2}\right]$$

Finally, the query region becomes:

$$R_Q' = \bigcup_{0 \leq \alpha \leq 1-\lambda} \left\{(x', y') \mid a - \Phi^{-1}(\alpha) y' \leq x' \leq b - \Phi^{-1}(\lambda + \alpha) y'\right\},$$

which is exactly Ω. It is also known R_I and R_Q overlap if and only if R_I' and R_Q' overlap because $\mathcal{F}$ is a one-to-one mapping.

Now our task becomes testing the overlap between R_I' and $R_Q' = \Omega$ (Eq. (2.5)). For ease of explanation, simply describe the index entry R_I' as $[\mu_1, \mu_2; \sigma_1, \sigma_2]$.

First, we show that $\Omega \not\subset R_I'$: Since σ is the standard deviation of a random variable Z, $\sigma \geq 0$. In the extreme case when $\sigma = 0$, Z is reduced to a constant, so $\sigma_{min} = 0$, where

σ_{min} is the minimum value of σ in Ω. For a valid search key, it is impossible that $\sigma_1 < 0$, so $\Omega \not\subset R'_I$.

Given that $\Omega \not\subset R'_I$, testing whether R'_I and Ω overlaps is the same as to test whether there exists a point (μ_0, σ_0) on the edges of R'_I, such that $(\mu_0, \sigma_0) \in \Omega$.

Define a function $g(\mu, \sigma) = \Phi\left(\frac{b-\mu}{\sigma}\right) - \Phi\left(\frac{a-\mu}{\sigma}\right)$, where Φ is the *cdf* of the standard normal distribution. It is straightforward to see that $(\mu, \sigma) \in \Omega$ *iff* $g(\mu, \sigma) \geq \lambda$. Then the problem is again transformed to checking whether the maximum value of $g(\mu, \sigma)$ on edges of R'_I, denoted as g_{max}, is no less than λ.

Without loss of generality, let us consider one edge of R'_I, defined as $E_{\sigma_1} = \{(\mu, \sigma) \mid \mu_1 \leq \mu \leq \mu_2, \sigma = \sigma_1\}$. The goal is to find g_{max} for all points on E_{σ_1}. Let $g'(\mu) = g(\mu, \sigma_1) = \Phi\left(\frac{b-\mu}{\sigma_1}\right) - \Phi\left(\frac{a-\mu}{\sigma_1}\right)$. By taking the derivative w.r.t. μ and setting it to be 0, we get $\phi\left(\frac{b-\mu}{\sigma_1}\right) = \phi\left(\frac{a-\mu}{\sigma_1}\right)$, where ϕ is the *pdf* of the standard normal distribution. According to the symmetry of the standard normal distribution, $g'(\mu)$ has an extreme value at $\mu = (a+b)/2$. Then g_{max} can be gained as follows:

$$g_{max} = \begin{cases} \max\left\{g'(\mu_1), g'(\mu_2), g'\left(\frac{a+b}{2}\right)\right\} & \text{if } \mu_1 < \frac{a+b}{2} < \mu_2 \\ \max\left\{g'(\mu_1), g'(\mu_2)\right\} & \text{otherwise} \end{cases}$$

Similar analysis can be made for other edges of R'_I.

An optimization. According to the above discussion, for each index entry R'_I, we find the maximum value of $g(\mu, \sigma)$ on its edges and check whether it is no less than λ. A further optimization would be to find the circumscribed rectangle (minimum bounding box) of Ω, denoted as $MBR_\Omega = [\mu_{min}, \mu_{max}; \sigma_{min}, \sigma_{max}]$: if R'_I does not overlap with MBR_Ω, it is guaranteed that it does not overlap with Ω either. Below we show how to find μ_{min}, μ_{max}, σ_{min} and σ_{max}.

We have already showed that $\sigma_{min}=0$. When $\lambda \geq 0.5$[1], $\forall \alpha \in [0, 1-\lambda]$, $\Phi^{-1}(\alpha) \leq 0$ and $\Phi^{-1}(\lambda+\alpha) \geq 0$. Then $a \leq a-\Phi^{-1}(\alpha)\sigma \leq \mu \leq b-\Phi^{-1}(\lambda+\alpha)\sigma \leq b$. And we get $\mu_{min}=a$ and $\mu_{max}=b$, both obtained when $\sigma=0$. Finally in order to find σ_{max}, we write out the expression of any point (μ, σ) on the boundary of Ω as:

$$\begin{cases} \mu = \dfrac{a\Phi^{-1}(\lambda+\alpha) - b\Phi^{-1}(\alpha)}{\Phi^{-1}(\lambda+\alpha) - \Phi^{-1}(\alpha)} \\ \sigma = \dfrac{b-a}{\Phi^{-1}(\lambda+\alpha) - \Phi^{-1}(\alpha)} \end{cases}$$

Let $\partial\sigma/\partial\alpha = 0$, we have $\phi(\Phi^{-1}(\lambda+\alpha)) = \phi(\Phi^{-1}(\alpha))$, where ϕ is the *pdf* of the standard normal distribution. Then $\alpha = (1+\lambda)/2$ and finally we get

$$\sigma_{max} = \frac{b-a}{\Phi^{-1}(\frac{1+\lambda}{2}) - \Phi^{-1}(\frac{1-\lambda}{2})}.$$

Overlap routine design

Step 1 If R'_I does not overlap with MBR_Ω, return FALSE.

Step 2 Find g_{max} for points on the edges of R'_I, if $g_{max} < \lambda$, return FALSE; otherwise, return TRUE.

2.3.3 Additional Join Indexes

We also provide join indexes for (1) band joins of multivariate GMMs, (2) other joins using linear predicates, and (3) proximity joins using Euclidean distance.

Extension of Band Joins to multivariate GMMs. When we have multiple join attributes $\mathbf{A}$, the band join involves conjunctive predicates that each is a 1-dim range predicate. It is easy to see that

$$Pr[\wedge_{i=1}^{|\mathbf{A}|}(a_i < R.A_i - S.A_i < b_i)] \leq \min_i \{Pr[a_i < R.A_i - S.A_i < b_i]\}.$$

[1]We focus on cases when $\lambda \geq 0.5$, as it is more desirable in real applications.

A necessary condition for $Pr[\bigwedge_{i=1}^{|\mathbf{A}|}(a_i < R.A_i - S.A_i < b_i)] \geq \lambda$ is $Pr[a_i < R.A_i - S.A_i < b_i] \geq \lambda$ for all i. This transforms the join of multivariate GMMs (or Gaussians) into multiple joins of univariate GMMs (or Gaussians). So we build an index for each join attribute in S. For a probing tuple r, all join indexes need to be retrieved, and a tuple s is a candidate match if it is returned by all the indexes.

Other Joins with Linear Predicates and General Distributions. We also support join predicates that are the "opposite" of band joins, e.g., "$|R.A - S.A| > \delta$". Such predicates can be useful for detecting a sudden dramatic change of the value of an attribute, e.g., the brightness of a star. We offer a necessary condition for such joins and build a join index accordingly. We can still apply linear transformation $Z = X_r - X_s$. Then we can prove the following statement by contradiction based on Chebyshev's inequality:

Theorem 2.3.3. *For a random variable Z with mean μ and variance σ^2, if $Pr[|Z| > \delta] \geq \lambda$, then $\delta \leq |\mu| - \frac{\sigma}{\sqrt{\lambda}}$.*

Plugging $\mu = \mu_r - \mu_s$, $\sigma^2 = \sigma_r^2 + \sigma_s^2$ back, we can get the necessary condition for $Pr[|r.A - s.A| > \delta] \geq \lambda$ as follows:

$$\delta \leq |\mu_r - \mu_s| - \sqrt{\frac{\sigma_r^2 + \sigma_s^2}{\lambda}}. \tag{2.6}$$

Define the search key to be $x = \mu_s$ and $y = \mu_s^2 - \frac{\sigma_s^2}{\lambda}$, then Eq. (2.6) can be broken into two cases:

$$\begin{cases} x < \mu_r - \delta \\ y - 2(\mu_r - \delta)x \geq -(\mu_r - \delta)^2 + \frac{\sigma_r^2}{\lambda} \end{cases} \tag{2.7}$$

or

$$\begin{cases} x > \mu_r + \delta \\ y - 2(\mu_r + \delta)x \geq -(\mu_r + \delta)^2 + \frac{\sigma_r^2}{\lambda} \end{cases} \tag{2.8}$$

The index is still an R-tree and the query region is defined by inequalities in Eq. (2.7) and (2.8).

Distance Join of General Distributions. We next consider join predicates that involve the Euclidean distance between two sets of attributes $R.\mathbf{A}$ and $S.\mathbf{A}$:

$$\mathcal{D}(R.\mathbf{A}, S.\mathbf{A}) = \sqrt{\sum_{i=1}^{|\mathbf{A}|}(R.A_i - S.A_i)^2} < \delta$$

Then the probabilistic threshold join is $R \bowtie_{\mathcal{D}(R.\mathbf{A},S.\mathbf{A})<\delta,\ \lambda} S$. Such joins are commonly used to check proximity of objects, e.g., two stars that are within 30 arcseconds of each other in query Q2.

Proximity joins can be supported using our techniques for band joins. When $|\mathbf{A}|=1$, the join predicate can be rewritten to "$-\delta < R.A - S.A < \delta$", and thus can be directly supported as a band join. When $|\mathbf{A}| > 1$, we seek an upper bound of $Pr[\mathcal{D}(R.\mathbf{A}, S.\mathbf{A}) < \delta]$. We know that if the Euclidean distance between $R.\mathbf{A}$ and $S.\mathbf{A}$ is less than δ, then the Manhattan distance between $R.\mathbf{A}$ and $S.\mathbf{A}$ is less than δ in each dimension. So, $Pr[\mathcal{D}(R.\mathbf{A}, S.\mathbf{A}) < \delta] < Pr[\bigwedge_{i=1}^{|\mathbf{A}|}(-\delta < r.A_i - s.A_i < \delta)]$. This allows us to use indexes for band joins of multivariate GMMs or Gaussians for distance joins.

2.4 Per-tuple Based Planning

We next discuss threshold query processing that takes a selection-join-projection query and returns tuples that satisfy the query with a probability over the threshold λ (i.e., their tuple existence probabilities $> \lambda$). A naive approach would be to perform probabilistic query processing as in earlier work and then apply the threshold filter at the end of the processing, wasting a lot of computation on nonviable answers. To prune nonviable answers early, we push the threshold λ earlier to each relational operator in the query plan, and apply our techniques from the previous sections as follows:

$$R \bowtie_{\theta,\lambda} S = \sigma_{\theta,\lambda}(R \times_{\theta,\lambda} S), \quad \sigma_{\theta,\lambda}(T) = \sigma_{\theta,\lambda}(\tilde{\sigma}_{\theta,\lambda}(T)),$$

where $R \times_{\theta,\lambda} S$ is the filtered cross product using a join index (Section 2.3), $\tilde{\sigma}_{\theta,\lambda}(T)$ is the fast filter that prunes tuples with a relaxed condition (Section 2.2.2), and $\sigma_{\theta,\lambda}(T)$ is

the exact selection that evaluates the condition using integrals but possibly with reduced dimensionality (Section 2.2.1). For continuous uncertain attributes, projections do not involve duplicate elimination because there does not exist an *finite* set of values to project onto. Hence, projections do not change tuple existence probabilities and are not further discussed in this work.

Tuple				**Selectivity**	
id	r	q_r	u_r	$r < 24$	$q\_r^2 + u\_r^2 > 0.25$
1	$N(27.0, 2.2)$	$N(1.2, 2.2)$	$N(0.1, 1.1)$	0.08	0.95
2	$N(21.6, 0.1)$	$N(0.1, 0.1)$	$N(-0.1, 0.1)$	1	1.74×10^{-4}

Table 2.1: Illustration of per-tuple based selectivity with Q1 and two tuples: each tuple has three normally distributed attributes, r, q_r, and u_r; for each predicate in Q1, these tuples have different selectivities.

Besides our techniques for joins and selections separately, there remains a query optimization issue: What is the most efficient way to arrange filtered cross products, fast filters for selections, and exact selections in a query plan? We consider both the cost and selectivity of operators as in a traditional query optimizer. However, several key differences exist in the new context: (1) Due to the use of integrals, exact selections can have high costs and should be treated as "expensive predicates". (2) The selectivity of an operator captures its filtering power on the input data. Under attribute uncertainty, *selectivity needs to be defined on a per-tuple basis*. (3) The above property further implies that the optimal order of evaluating operators also varies on a per-tuple basis.

Example 2.4.1. *Consider Q1 and two tuples t_1, t_2 in Table 2.1. For predicate θ_1: "$r < 24$", let X_r^1 and X_r^2 be the random variables for $t_1.r$ and $t_2.r$ correspondingly. $X_r^1 \sim N(27, 2.2)$, so $Pr[X_r^1 < 24] = 0.08$; $X_r^2 \sim N(21.6, 0.1)$ and $Pr[X_r^2 < 24] \approx 1$. So t_1 has a much lower probability of satisfying θ_1, hence more likely to be filtered. To the contrary, for predicate θ_2: "$q\_r^2 + u\_r^2 > 0.25$", t_2 has a much lower probability to pass θ_2 than t_1. Thus, the optimal evaluation order is θ_1 followed by θ_2 for t_1, and the reverse for t_2.*

Due to the above reasons, we advocate a *per-tuple, dynamic* query optimization approach with the following features: (1) A query plan is determined for each tuple rather than a whole set. (2) The query plan arranges all operators based on both cost and selectivity. (3) Such planning is performed at a low cost for each tuple. Traditional query optimizers consider a static query plan for a set of tuples [20], hence not suitable for our problem. Data stream systems [7, 13] can adapt query plans dynamically but only estimate selectivity for a set of tuples and lack support of uncertain attributes. Recent work on probabilistic threshold query optimization establishes algebraic equivalence for query optimization, but still uses static query plans and further ignores operator costs in query planning [73].

In the rest of this section, we detail our new query optimization approach. We focus on the data stream setting: like in existing systems [7], some streams can have indexes built on while other streams are used to probe these indexes. We assume that the decision of which indexes to build has been made separately and focus on query optimization only. Our approach can be applied to stored data by viewing the result of a file scan as a data stream.

To begin with, we define the selectivity, denoted by Γ, of a selection on each tuple t and the selectivity of a filtered cross product between a probing tuple t and a set S:

$$\Gamma^{\sigma}_{\theta,\lambda}(t) = Pr[R_\theta(\mathbf{X^t})], \quad \Gamma^{\times}_{\theta,\lambda}(t,S) = \frac{\text{num. true matches from } S}{|S|}$$

Query optimization requires the knowledge of both cost and selectivity of each operator. Our approach combines offline measurements of unit operation costs, which depend only on the types of predicates, and online selectivity estimation, which depends on the attribute distribution in each tuple.

2.4.1 Cost and Selectivity Estimation

The unit operation in an exact selection is an integral. The integration cost depends on the dimensionality of integration and the shape of the selection region. For instance, we can benchmark 1-dimensional integrals on intervals, 2-dimensional integrals on rectangles and

circles, and higher-dimensional integrals on hyper-rectangles and circles. Regarding the filters for selection, if they use known inequalities, then they have negligible costs. Filters that use optimization techniques such as the Lagrange multiplier may have a non-trivial cost, which can again be measured offline based on the types of predicates. Finally, the unit operation in a filtered cross product is to retrieve a match from the index for each probing tuple. Its cost can be estimated based on the height of the tree and the cost of the overlap test at each level of the tree.

We then define the selectivity, denoted by Γ, of a selection on each tuple t and the selectivity of a filtered cross product between a probing tuple t and a set S:

$$\Gamma^{\sigma}_{\theta,\lambda}(t) = Pr[R_\theta(\mathbf{X^t})], \quad \Gamma^{\times}_{\theta,\lambda}(t,S) = \frac{\text{num. true matches from } S}{|S|}$$

The selectivity of an operator can be estimated only when a tuple arrives with its attribute distribution. For a selection, we estimate its selectivity for a tuple, $\Gamma^{\sigma}_{\theta,\lambda}(t)$, by taking the average of its upper and lower bounds. Recall that we showed many upper bounds derived from statistical inequalities in Section 2.2. Actually we can also obtain lower bounds using appropriate inequalities. For instance, given a one-dimensional range predicate (a,b), Markov's inequality can be applied when $0 \in (a,b)$ and yields a lower bound on the selectivity. Chebyshev's and Cantelli's inequalities can be applied similarly. Finally, to estimate the selectivity of a filtered cross product for a probing tuple, our current solution is to perform the index lookup to count the matches but without retrieving the complete tuples.

2.4.2 Online Query Planning and Execution

In our approach, online query planning and execution for each tuple interleaves selectivity estimation and ordering of operators in iterations. This is because while we can estimate the selectivity of predicates on a base tuple r, we cannot estimate the selectivity of the join predicate on r and s until the tuple including r and s is produced with the new joint attribute

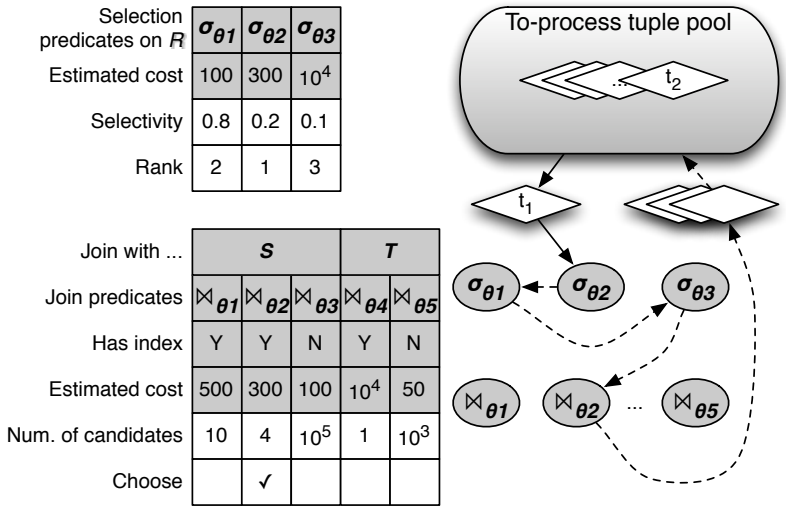


Figure 2.3: Illustration of tuple-based query planning.

distribution. Moreover, we cannot afford to perform an exhaustive search of the global optimal plan as in earlier work [20] due to per-tuple based planning. Therefore, we break a query into several blocks that each involve at most one join. For each query block, we repeat the following steps:

Step 1: Estimate selectivities of selections. We take all predicates specified on an input tuple t, and group these predicates into independent groups as described at the beginning of Section 2.2. For each independent group of predicates θ_i, we then allocate a selection operator $\sigma_i(t)$. We estimate the selectivity of each selection by taking the average of its lower and upper bounds.

Step 2: Rank and execute filters and selections. We expand each selection with all possible filters: $\sigma_i(t) = \sigma_i(\tilde{\sigma}_{ij}(\cdots\tilde{\sigma}_{i1}(t)))$. If there exist fast filters based on known statistical inequalities, we apply all of them as they have negligible costs; otherwise, we apply the filter using constraint optimization. We rank filters and selections in ascending order of selectivity over cost. Filters are ranked before the corresponding selection if they have a lower cost; otherwise, they are unnecessary and should be removed from the plan.

Then we execute the filters and selections in order. The tuple starts with the existence probability $E_p = 1$ and a query threshold λ_q. A selection with the predicate θ reduces the tuple existence probability to $E'_p = E_p \cdot Pr[R_\theta(X^t)]$. A filter estimates an upper bound $\tilde{E}'_p$. The tuple is dropped whenever $E'_p < \lambda_q$ or $\tilde{E}'_p < \lambda_q$.

Step 3: Choose a relation to join with. For all relations that have not been joined with t, we probe all available indexes and count the number of matches of t from each index. We then multiply the number of matches with the cost of an index lookup, and finally choose the join index that yields the smallest value of the product. If we have exhausted join indexes, we simply choose the relation with the smallest size and use a full scan as the access method.

Step 4: Execute the (filtered) cross product. Once we have chosen to join the tuple t with an relation S, we execute the filtered cross product using the index on S if existent, or a cross product using a file scan on S. Once a new tuple t' is emitted, we mark all join predicates relevant to t' as selection predicates, and repeat the above four steps for the next query block.

Example 2.4.2. *Figure 2.3 shows the planning for tuple t_1 from relation R. t_1 has to pass three selection predicates, a join with relation S with three join predicates, and a join with relation T with two join predicates. These predicates and their estimated costs are shown in the shaded rows of the tables. In step 1, the selectivities of three selections are entered into the top table. In step 2, the selections are ranked with θ_2 first, then θ_1, and finally θ_3 (the filters are not shown in this example). In step 3, we choose the join with S using the second predicate because there is a join index and the expected cost of retrieving matches is the lowest. In step 4, tuple t_1 is paired with three matches from the join index. The three new tuples are sent back to the to-process pool for further processing. For these tuples, the join predicates associated with S become selection predicates, while those associated with T remain as join predicates.*

2.5 Experimental Evaluation

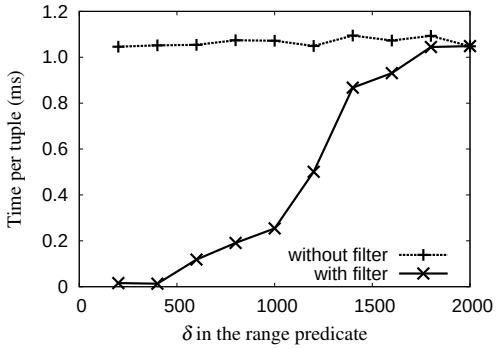
In this section, we evaluate our threshold query processing and optimization techniques. To demonstrate our performance benefits, we compare to the state-of-the-art techniques for indexing continuous uncertain data [23, 24] and for optimizing threshold queries [73]. Our evaluation uses real data and queries from the Sloan Digital Sky Survey (SDSS) [92]. The released data archive takes about 1GB. For experiments on selections and joins, we used the *Star* table, which has 57328 tuples and each column of uncertain attribute is about 1MB. We consider Q1 and Q2 for experiments on query planning, both of which involve the *Galaxy* table with 91249 tuples.

2.5.1 Techniques for Optimizing Selections

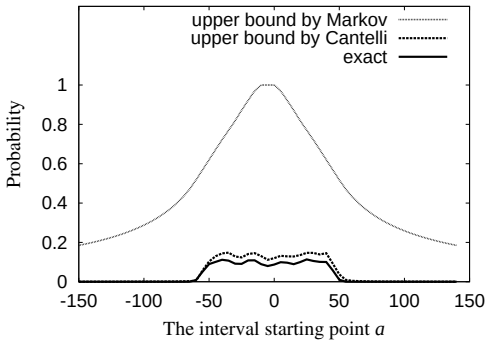
Expt 1: We first evaluate our general filtering technique described in Section 2.2.2.1. We consider a selection "$100 < rowc < 100 + \delta$ and $100 < colc < 100 + \delta$", which selects stars located in a square region anchored at the lower left vertex $(100, 100)$ and with side length δ. We can directly test the overlap between the selection region and each tuple's Chebyshev region due to their regular shapes.

We first set the threshold $\lambda = 0.7$ and varied δ from 200 to 2000, which approximately covers selectivities from 0% to 100%. We report the time cost per tuple for evaluating the selection with and without filters (baseline) in Figure 2.4(a). The baseline has a constant high cost because it computes a 2-dimensional integral for each tuple, no matter what δ value is given. In contrast, using our filter the per tuple cost is very low for small δ values because most tuples can be filtered without computing integrals. As δ grows, more tuples pass the filter and invoke integrals for exact evaluation. The two curves meet when $\delta = 2000$ and 98% tuples satisfy the predicate.

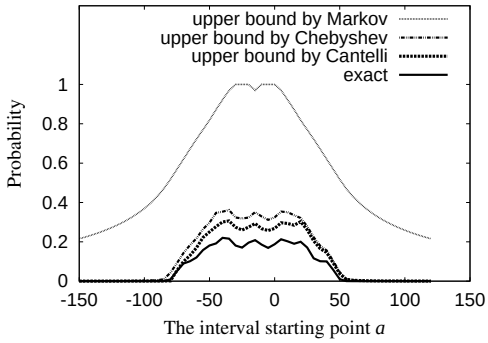
Expt 2: We also evaluate the effectiveness of the fast filters from Section 2.2.2.2. Since they have negligible costs, we focus on how tight their upper bounds are, i.e., their filtering power. We here use synthetic data with various controlled properties in microbenchmarks.



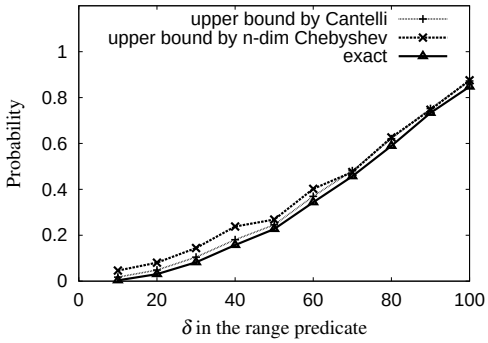
(a) general filter vs exact integration $rowc \in (100, 100+\delta) \wedge colc \in (100, 100+\delta)$



(b) fast filters vs exact integration ($a < x < a+10$)



(c) fast filters vs exact integration $(a < x < a+10)$ or $(a+20 < x < a+30)$



(d) fast filter, general filter, and exact integration($x^2+y^2 < \delta^2$)

Figure 2.4: Experimental results for selections.

Consider predicates on a single attribute (Category 1 in Section 2.2.2.2): (i) $a<x<b$, (ii) $a_1<x<b_1$ or $a_2<x<b_2$. To have workloads with controlled properties, we generated synthetic data where the mean and variance of the normal distribution for each tuple are randomly chosen from (-50,50) and (0,10) respectively. For the predicate (i), the Cantelli filter always gives a tighter upper bound than the Chebyshev filter, so we only compare Markov with Cantelli and use the exact probability as the baseline. We set the interval length to be 10 and vary the starting point of the interval from -150 to 150. As shown in Figure 2.4(b), Cantelli's upper bound is much tighter than Markov and close to the exact probability, which agrees with known statistical results. For predicate (ii), we compare all three filters as they may have tradeoffs. From Figure 2.4(c), we observe similar trends

as before. In addition, the upper bounds by the Chebyshev filter lie in between those by Markov and Cantelli on the average (though for specific tuples, the order of the filters may be otherwise).

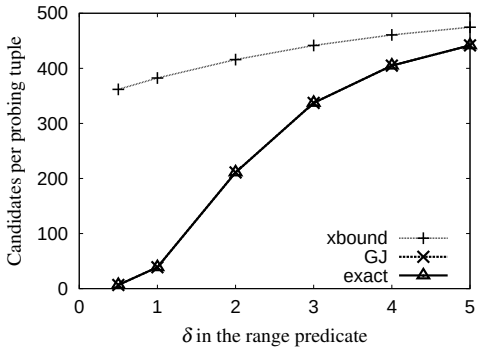
We next consider predicates in a quadratic form: "$x^2+y^2<\delta^2$" (Category 2). We used a synthetic trace with attributes x and y whose mean and variance are uniformly distributed in (0,100) and (0,10) respectively. We compared the Cantelli filter which transforms the quadratic form to a single random variable, with the general multi-dimensional filter that uses the Lagrange multiplier for constraint optimization (for this predicate, there is a fast solver). We varied δ from 10 to 100. Figure 2.4(d) shows that both the Cantelli filter and general filter capture the trend of the exact probability and offer tight bounds, with Cantelli being slightly better. They are both over 400x faster than integration, hence good choices for quadratic predicates.

2.5.2 Techniques for Optimizing Joins

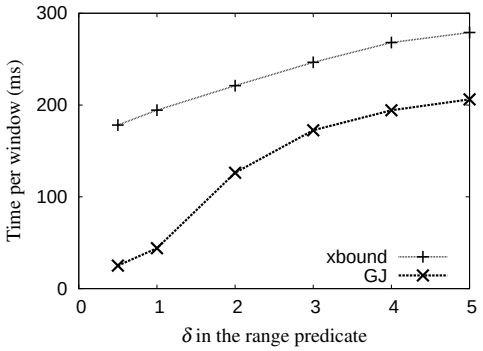
We implemented a state of the art join index on continuous uncertain attributes [24, 23], called X-BOUND join index. It is based on a "loose" necessary condition for the join predicate to be true, hence resulting in poor performance as we demonstrate soon. Below we describe it in more details:

Definition 6 (x-bound)**.** *For a random variable Y with density function f and domain $[l,u]$, given $0 \leq x \leq 1$, the x-bound of a distribution consists of two values, called left-x-bound (l_x) and right-x-bound (u_x), where $\int_l^{l_x} f(y)\mathrm{d}y = x$ and $\int_{u_x}^{u} f(y)\mathrm{d}y = x$.*

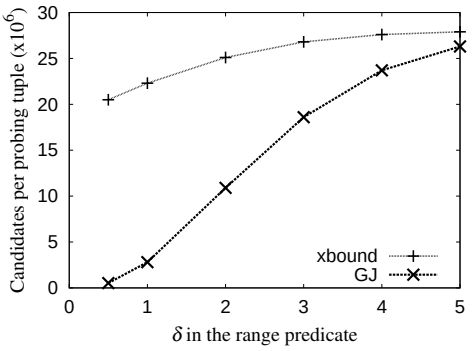
Consider $R \bowtie_{\theta,\lambda} S$, a **necessary condition** for $Pr[|X_r - X_s| \leq \delta] \geq \lambda$ extracted from [23] is $u_{s,\lambda} \geq l_r - \delta$ and $l_{s,\lambda} \leq u_r + \delta$, where (l_r, u_r) is the domain of X_r, $l_{s,\lambda}$ and $u_{s,\lambda}$ are the left-λ-bound and right-λ-bound for X_s. Assume an join index is built on S. Given λ, for each tuple s, insert the **search key** $(l_{s,\lambda}, u_{s,\lambda})$ into an R-tree. When a probing tuple r arrives, the **query region** is $\{(l_{s,\lambda}, u_{s,\lambda}) \mid l_{s,\lambda} \leq u_r + \delta, u_{s,\lambda} \geq l_r - \delta\}$.



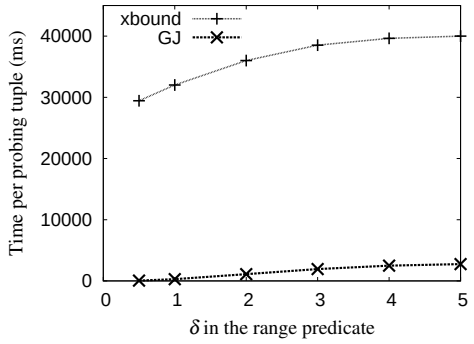
(a) vs GJ in filtering power (stream) ($|R.u - S.u| < \delta$)



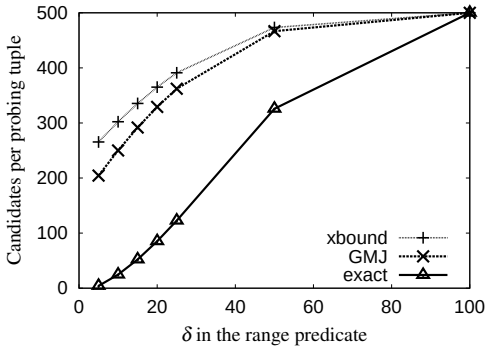
(b) vs GJ in efficiency (stream) ($|R.u - S.u| < \delta$)



(c) vs GJ in filtering power (disk) ($|R.u - S.u| < \delta$)



(d) vs GJ in efficiency (disk) ($|R.u - S.u| < \delta$)



(e) vs GMJ in filtering power (stream) ($|R.u - S.u| < \delta$)

Figure 2.5: Experimental results for joins.

For a probing tuple r with domain $(-\infty, +\infty)$, the above necessary condition has no power in guiding the search of $(l_{s,\lambda}, u_{s,\lambda})$. We modify the domain of a normally distributed random variable to be $(\mu - 5\sigma, \mu + 5\sigma)$, and modify the domain of a GMM to be $(F^{-1}(0.00001), F^{-1}(0.99999))$, where F^{-1} is the *inverse cdf* of the random variable.

Expt3: Band Join of Gaussians. We first study the filtering power and efficiency of our index for Gaussians, called GJ for short, and x-bound. We consider a join "$|R.u - S.u| < \delta$" with the threshold $\lambda = 0.7$ and varied δ.

We first evaluate the join in the stream setting: A tumbling window of size W is applied to both R and S inputs; each window contains a set of tuples. An index is built in-memory on the current window of S. Each tuple in the current R window probes the index after it is constructed. The retrieved (r, s) pairs are finally validated for true matches by computing an integral of the joint distribution. Hence, there are three cost components in this windowed join: *index construction*, *index lookup*, and *validation using integrals*. We observe consistently that the validation step is the dominating cost as integration is indeed very expensive. Below we report results using W =500 (other W sizes reveal similar trends).

Figure 2.5(a) shows the number of candidates returned by our GJ index and the index as well as the number of true matches. We can see that GJ returns exactly the true match set because it uses a sufficient and necessary condition for the join predicate. In contrast, the x-bound index returns much more candidates. The difference becomes smaller as δ increases, because more tuples become true matches; when $\delta = 100$, almost all tuples in the indexed relation are true matches. Figure 2.5(b) shows the efficiency of the two indexes. GJ significantly outperforms because there is no need to validate the candidates returned from the index.

We then evaluate the join in a disk setting, where indexes are pre-computed and stored on disk. Due to the limited size of the real data, we replicated it to 500MB with 28 million tuples. The R-tree took 1.3GB while the memory size was set to 1GB in our Java system.

Since indexes are pre-constructed, their construction costs are not reported. Figure 2.5(c) shows the number of candidates for each probing tuple. While the trend appears similar to that in the stream setting, the absolute number for the y-axis is much larger. This determines the drastic difference between GJ and in time cost shown in Figure 2.5(d). This difference comes from both the validation cost and the I/O cost as returns many false positives.

Expt 4: Band Join of GMMs We then evaluate our join index for general distributions modeled as GMMs (called GMJ for short) and compare to . As the SDSS uses Gaussians only, we generated a synthetic trace of GMMs for this experiment. The attribute u in each tuple has two Gaussian components; the coefficient, mean and variance of each component are uniformly drawn from (0,1), (0, 100), and (0,10) respectively. We report the results using the stream setting with $W = 500$. As Figure 2.5(e) shows, both indexes return more candidates than the true matches because they are both based on necessary conditions for the join predicate. But GMJ is always better than until they meet at $\delta = 100$, where the selectivity is near 100%, because uses a "looser" condition . Since validation is the dominating cost, the time cost follows the same trend as the number of candidates in Figure 2.5(e).

2.5.3 Per-tuple Based Planning and Execution

We finally evaluate our dynamic per-tuple planning technique on Q1 and Q2. We compare it with static query planning [73], where a fixed plan is chosen for each query based on the selectivities of predicates over the entire data set—we give such full knowledge to the static query optimizer, hence showing its best performance. Query Q1 uses two predicates: $\theta_1 : r < \delta_1$; $\theta_2 : q\_r^2 + u\_r^2 > \delta_2^2$. Query Q2 uses two join predicates: $\theta_3 : |(G1.u - G1.g) - (G2.u - G2.g)| < \delta_3$ and $|(G1.g - G1.r) - (G2.g - G2.r)| < \delta_3$; $\theta_4 : (G1.rowc - G2.rowc)^2 + (G1.colc - G2.colc)^2 < \delta_4^2$. We design two query templates based on Q1 and Q2 on the *Galaxy* view in SDSS and show them in Figure 2.6. Basically all tuples will be routed to the quick filters whenever the filters can be applied and there is no specific order of applying filters, as their costs are all very low. After that, the dynamic

	δ_1			δ_2			δ_3			δ_4		
Value	20	22	24	0.2	0.5	1	0.5	1	2	400	800	1600
Selectivity(%)	8.3	61.9	95.9	44.3	14.5	6.9	40.1	81.7	99.4	45.4	83.7	100

Table 2.2: Selectivity for different $\delta_i (i = 1 \cdots 4)$.

optimizer decides the order of evaluating exact selection predicates, shown by a box in Figure 2.6. We vary the parameters δ_1 and δ_2 to control selectivities of predicates for Q1 and δ_3 and δ_4 for Q2, which affects planning.

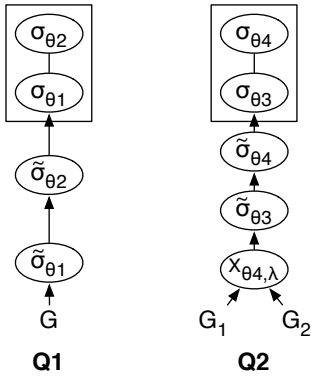


Figure 2.6: Plan space in dynamic planning for Q1 and Q2.

Given various values of δ_i, the selectivity of each predicate averaged over the entire data set is shown in Table 2.2. The static query plan is decided by ordering predicates with lowest selectivity first.

Expt 5: We first consider Q1 and vary δ_1 and δ_2. Table 2.3 shows the time cost per tuple for static, dynamic and optimal planning. The optimal planning loads the optimal plan for each tuple (generated offline) into memory before it runs. The plan space for dynamic planning is shown in Figure 2.6. Our dynamic query planning outperforms the static one in all cases, with over 50% gains in most cases and is very close to the optimal planning. The reasons are three-fold: (1) Each tuple is routed based on its distribution and resulting selectivities of predicates. A tuple may be sent to a predicate that is overall not selective but has a larger chance to filter this tuple. (2) The predicate cost is taken into consideration. It is

δ_1	δ_2	**static order**	**static time (ms)**	**dynamic time (ms)**	**performance gain**	**optimal time (ms)**
20	0.2	[1 2]	0.6	0.181	70%	0.177
20	0.5	[1 2]	0.6	0.068	89%	0.067
20	1	[2 1]	9.6	0.050	99%	0.048
22	0.2	[2 1]	18.2	7.216	60%	7.007
22	0.5	[2 1]	13.9	1.515	89%	1.482
22	1	[2 1]	9.6	0.351	96%	0.348
24	0.2	[2 1]	18.2	15.613	14%	15.287
24	0.5	[2 1]	14.4	6.390	56%	6.334
24	1	[2 1]	9.6	2.264	76%	2.236

Table 2.3: Static planning vs Dynamic planning for Q1.

δ_3	δ_4	**static order**	**static time (s)**	**dynamic time (s)**	**performance gain**
0.5	400	[3 4]	28.0	4.25	85%
0.5	800	[3 4]	80.1	12.1	85%
0.5	1600	[3 4]	142	22.9	84%
1	400	[4 3]	149	16.7	89%
1	800	[3 4]	105	49.3	53%
1	1600	[3 4]	187	97.2	48%
2	400	[4 3]	160	72.1	55%
2	800	[4 3]	486	217	55%
2	1600	[3 4]	487	432	11%

Table 2.4: Static planning vs Dynamic planning for Q2.

possible for a tuple to be routed to a predicate with only a modest chance to filter the tuple but has a very low cost. (3) Our fast filters can drop tuples earlier at a lower cost than using the exact integration to evaluate predicates.

Expt 6: We next consider Q2 and vary δ_3 and δ_4. For this query, a join index is constructed for G_2 on the "rowc" and "colc" attributes for each window of size W. Table 2.4 shows the time cost of joining W tuples from the input G_1 with W tuples from G_2.

To evaluate the join predicates θ_3 and θ_4, all tuples need to be routed to probe the (only) join index first. Moreover, the evaluation costs of δ_3 and δ_4 are close due to the use of 2-dimensional integrals after linear transformation. The dynamic planning is better than the static one in all cases due to the reasons mentioned previously. As we increase δ_3 and δ_4,

more tuples satisfy both predicates. So the difference between the two schemes decreases and is mainly due to the benefit of using fast filters. There are several other interesting observations: (1) When $\delta_3 = 0.5$, the static optimizer always evaluates θ_3 first; the dynamic optimizer tends to choose θ_3 for most tuples as well. This is because for each tuple, most candidates returned by our index are true matches, then the selectivity of each pair of probing tuple and its candidate w.r.t. θ_4 will be estimated to be very close to 1. Since both optimizers route tuples to θ_3 as we increase δ_4, our improvement is mainly gained by the benefit of using fast filters. (2) When $\delta_3 = 1$ and $\delta_4 = 400$, the static optimizer evaluates θ_4 first, which is not a wise choice, because as mentioned above, most candidates returned by the index are true matches, then the evaluation of θ_4 can only filter very few tuples, and most tuples will be passed on to θ_3 next. In contrast, the dynamic optimizer tends to route tuples to θ_3 as discussed above and fewer tuples will be passed on to θ_4. (3) When $\delta_3 = 2$ and $\delta_4 = 1600$, according to the statistics, 80% of tuples satisfy both predicates and the difference between the two planning schemes reduce. The reason for the performance gain is mainly because the dynamic optimizer makes decision based on the content of the tuple and our filters can drop some tuples early.

2.6 Related Work

Most work on probabilistic databases models uncertain data using discrete random variables and evaluates queries based on the possible worlds semantics where a probabilistic database is defined as a probability distribution over numerous possible databases and a query on a probabilistic database returns a probability distribution of the query results on all possible databases.(e.g., [11, 30, 101]). Recent work has argued for using new techniques natural to continuous random variables [91], and showed significant performance gains of such techniques over discretization or Monte Carlo simulation for evaluating relational operators [98] and for ranking [61]. CLARO [96, 98] and ORION [73, 86] are two state-of-the-art systems that provide native support of queries on continuous uncertain data (without

using discretization or sampling). CLARO proposes a GMM based model and algorithms to compute attribute distributions and tuple existence probabilities for inclusion in query answers. However, it does not consider the threshold in query processing; a naive extension that applies the threshold filter at the end of query processing wastes a lot of computation on nonviable answers. ORION has general evaluation strategies for selection-projection-join queries, but uses heuristics for equivalent plans only based on the selectivity and hence lacks optimizations of queries with complex predicates such as those in Q1 and Q2. Furthermore, its query optimizer uses simple static query plans for any tuples with any distributions and results in inefficient execution, which we show in Section 2.5.

2.7 Conclusions

We presented techniques to optimize threshold query processing on continuous uncertain data by (*i*) expediting selections by reducing dimensionality of integration and using faster filters, (*ii*) expediting joins using new indexes, and (*iii*) using dynamic, per-tuple based planning that considers both cost and selectivity of operators. Results using the SDSS benchmark show that: (*i*) For selective queries, the devised filters can drop many non-viable tuples with negligible time cost compared to expensive integration cost. In our experiments on 2-dimensional queries, the performance gain is up to 66x for rectangular queries and 400x for circular queries. (*ii*) Our indexes on continuous uncertain data modeled by Gaussian distributions returns exactly the true match, which offers 1.4x$\sim$7.1x performance gain over X-BOUND in the stream setting and 14x$\sim$570x gain in the disk setting. (*iii*) Our dynamic, per-tuple-based query planner gets over 50% performance gains over the state-of-the-art query planner in most cases and is very close to the optimal planning in all cases.

CHAPTER 3

SUPPORTING DATA UNCERTAINTY IN ARRAY DATABASES

In this chapter, we still focus on the same type of query as in Chapter 2, i.e., threshold selection-projection-join queries on uncertain data modeled by continuous random variables, but in array databases which has gained popularity for scientific data processing recently due to performance benefits over relational databases.

3.1 Array Model and Algebra

In this section, we provide background on the array model and array algebra proposed recently [18, 89]. Furthermore, we extend the array model to accommodate uncertain data and formally define the semantics of array algebra under the uncertain data model.

3.1.1 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 $(\mu-3\sigma,\mu+3\sigma)$

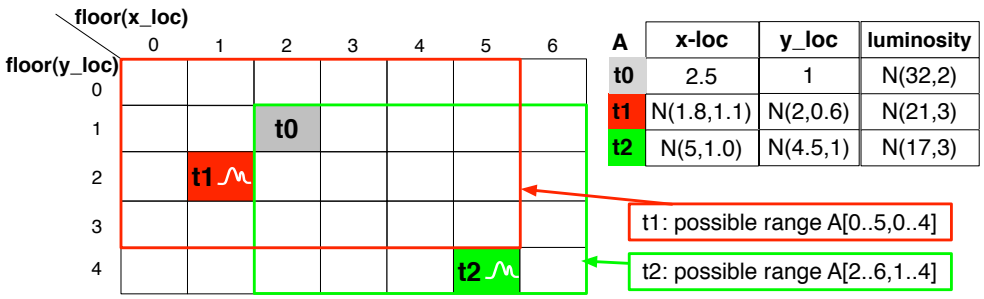
achieves a probability 0.997. If f_i is an arbitrary distribution with mean μ and standard deviation σ, we can define the possible range to be $(\mu - k\sigma, \mu + k\sigma)$ with a sufficiently large k chosen based on Chebyshev's inequality. As convention in this work, we always "round up" the possible region R_t to the boundary of cells, i.e., to be the smallest set of cells that contain R_t.

In this work, 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 [71, 93].

Example 3.1.1. *Figure 3.1 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. Note that Figure 3.1 is only a partial view of an array for illustration purposes. The full view of the logical array does not necessarily starts with (0, 0) and can be unbounded. The top-right corner in Figure 3.1 shows the corresponding relation of array $\mathbb{A}$ in the relational model.*

3.1.2 Array Algebra

For multidimensional arrays, SciQL [56, 105] and the Array Query Language (AQL) [80] are two popular high-level declarative languages, while the Array Functional language (AFL) [80] 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



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

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.

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 [97].

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 7 (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, $Subarray(\mathbb{A}, 1.5 \leq x\_loc \leq 3.3 \text{ and } 2.1 \leq y\_loc \leq 4.8)$. 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 [80].[1] (2) If the dimension attributes are continuous-valued, equi-join is seldom used. Instead, θ takes a form of proximity join. A common form is linear proximity (a.k.a. l_1-distance) join, $|\mathbb{A}.d_i - \mathbb{B}.d_i| < \delta$ for each dimension attribute d_i. The

[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.

join condition essentially defines a band region for each pair of join attributes. As noted earlier, we focus on continuous uncertain data in this work 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 8 (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, $\mathbb{A}[i_1,\cdots,i_d] \bowtie_{\theta,\lambda} \mathbb{B}[i_{d+1},\cdots,i_{2d}]$ $= \{(t_1,t_2) | t_1 \in \mathbb{A}[i_1,\cdots,i_d], t_2 \in \mathbb{B}[i_{d+1},\cdots,i_{2d}], \int\int_\theta f_{t_1}(\mathbf{x}) \cdot f_{t_2}(\mathbf{y}) d\mathbf{x} d\mathbf{y} \geq \lambda\}$, where $f_{t_1}(\mathbf{x})$ and $f_{t_2}(\mathbf{y})$ are the probability density functions for t_1 and t_2, respectively.*

(3) *Regrid-Aggregation* partitions an input array into non-overlapping blocks, and for each block, applies an aggregate function to all the tuples in the block. The output array has one cell for each block which contains the aggregate value computed. 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.

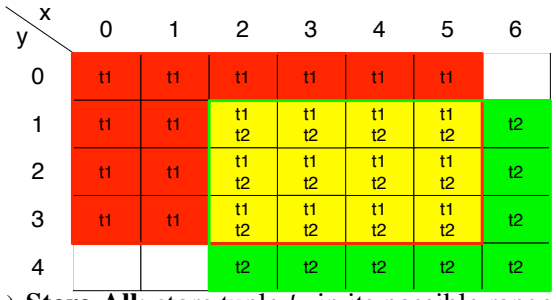
(4) *GroupBy-Aggregation* takes three arguments including an input array $\mathbb{A}^d$, a list of grouping dimensions $\mathbb{G}^{d_1}$, where $d_1 \leq d$, and an aggregate function. Again, it can be viewed as repeated application of *Subarray* to construct array blocks corresponding to the groups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

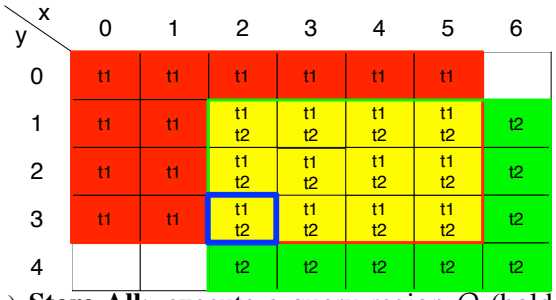
3.2 Native Support for Subarray

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.

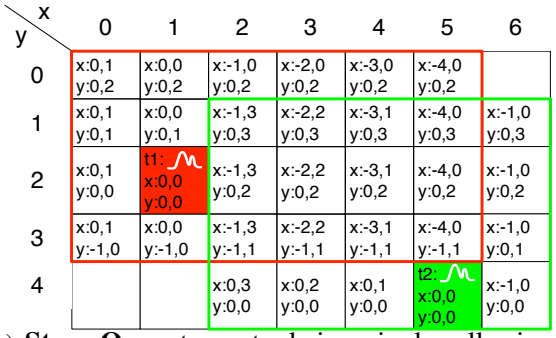
Since *Subarray* is equivalent to selection in relational algebra, there are two options for implementation: The first option is to translate *Subarray* to selection in the relational setting. When the dimension attributes are uncertain, to avoid scanning all tuples in the database, existing work has built various indexes based on statistical quantities such as quantiles [21, 22, 93] and moments [71] 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 Section 3.4. 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



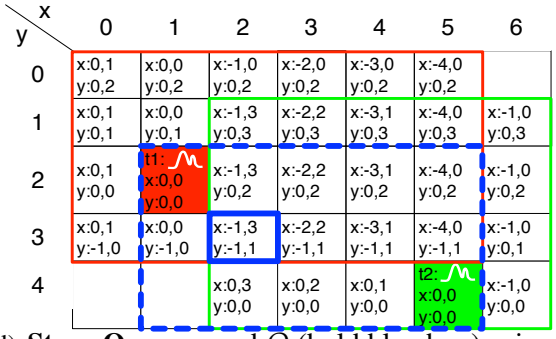
(a) **Store-All**: store tuple t_1 in its possible range $[0{:}5, 0{:}3]$, t_2 in $[2{:}6, 1{:}4]$.



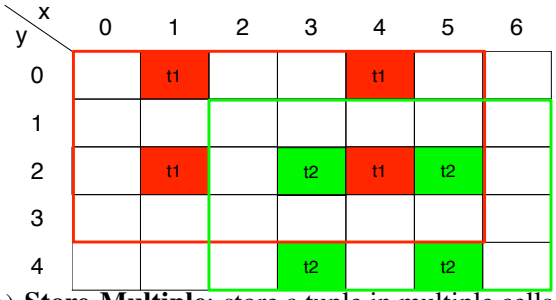
(b) **Store-All**: execute a query region Q (bold blue box) on the array.



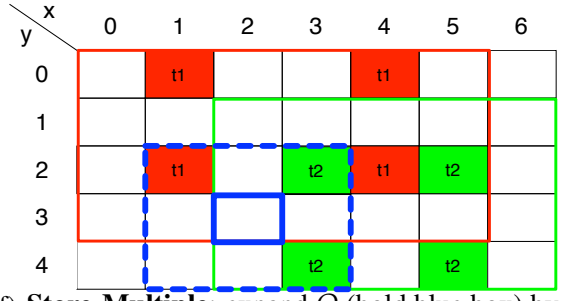
(c) **Store-One**: store a tuple in a single cell using its mean and store fences in other cells.



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



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



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

Figure 3.2: Alternative storage and evaluation strategies for tuples with uncertain dimension attributes.

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 Section 3.1.1. 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 probability 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.

3.2.1 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. Figure 3.2(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 Figure 3.2(b). 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 9 (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. Figure 3.2(c) shows the storage layout for tuples t_1 and t_2. Figure 3.2(d) 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 3.2.1. *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 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.

Figure 3.2(e) 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. Figure 3.2(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

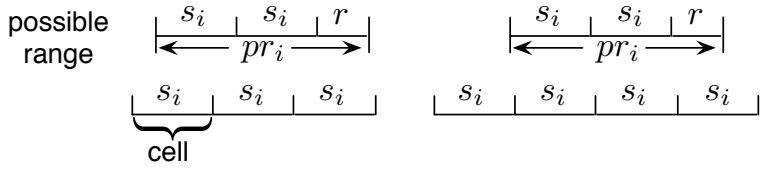


Figure 3.3: Illustration of the cells that overlap with a possible range

and typical query workloads, how do we choose the best configuration of k for optimal performance? We address these two issues in Section 3.2.2 and Section 3.2.3, respectively.

3.2.2 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 10 (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 11 (Covering Set)**.** *A given set of cells $\mathbf{C}$ is covered by a (discrete) set of cells $\mathbf{S}$ if and only if each cell in $\mathbf{C}$ is covered by at least one cell in $\mathbf{S}$; $\mathbf{S}$ is called the covering set of $\mathbf{C}$.*

Definition 12 (Problem of Tuple Copy Layout)**.** *Given a tuple t, find the minimum covering set $\mathbf{S}$ of its possible range $\mathbf{C}(R_t) = \mathbb{A}[l_1 : u_1, l_2 : u_2, \cdots, l_d : u_d]$ so that placing one copy of t in each cell in $\mathbf{S}$ 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 $\mathbf{C}(R_t)$ along all dimensions is able to find a copy of t.*

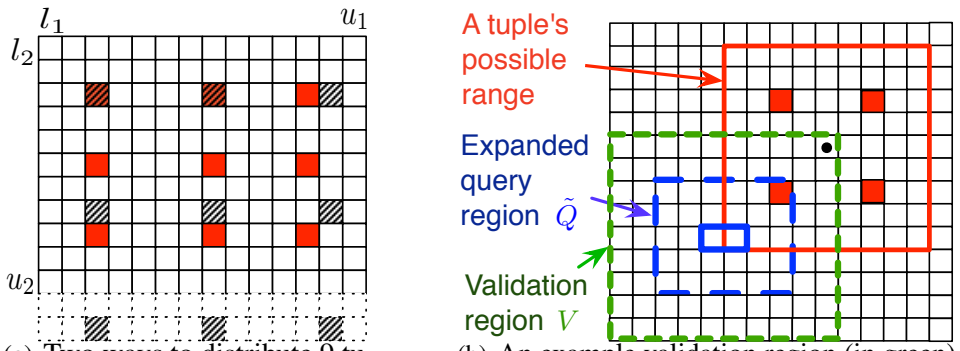
We address the problem by first showing the lower bound of the size of a covering set, as shown in the following proposition.

Proposition 3.2.2. *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 $\boldsymbol{C}(R_t)$.*

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 11, 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. □

Note that in the worst case, Proposition 3.2.2 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 3.2.2,



(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 3.4: Tuple copy distribution and validation under *store-multiple*.

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 Section 3.2.3.

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 from each other. Figure 3.4(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.

3.2.3 Cost Model of Subarray under Store-Multiple

We next propose a cost model for *Subarray* under the *store-multiple* scheme and use the model to find the optimal step size configuration. The symbols used in the model are summarized in Table 3.1. Like in SciDB [18], 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. 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 [36]. 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.

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.

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 3.1: Notation in modeling and analysis.

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 Figure 3.3. 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

$$\frac{s_i - r}{s_i} \left\lceil \frac{pr_i}{s_i} \right\rceil + \frac{r}{s_i} \left(\left\lceil \frac{pr_i}{s_i} \right\rceil + 1 \right) = \frac{pr_i}{s_i} + 1 \tag{3.1}$$

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.(3.1), and plug it into Proposition 3.2.2 to derive the number of copies per tuple t_{copies}:

$$t_{copies} = \prod_{i=1}^{d} \left(\left\lfloor \frac{pr_i/s_i}{2k_i + 1} \right\rfloor + 1 \right). \tag{3.2}$$

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., $\lfloor c/b \rfloor$:

$$C_{chunks} = T \cdot t_{copies} \Big/ \prod_{i=1}^{d} n_i \Big/ \lfloor c/b \rfloor . \tag{3.3}$$

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 $\tilde{Q}_{chunks}$:

$$\tilde{Q}_{chunks} = C_{chunks} \cdot \prod_{i=1}^{d} \left(\frac{q_i}{s_i} + 1 + 2k_i \right). \tag{3.4}$$

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 Figure 3.4(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 Figure 3.4(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:

$$T_{val} = T \Big/ \prod_{i=1}^{d} n_i \cdot \prod_{i=1}^{d} \left(\frac{q_i}{s_i} + 1 + z_i \right), \tag{3.5}$$

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 [92] updates the scanned image of the sky on a nightly basis and can build the statistics as a nightly observation is being produced. If domain knowledge 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. The step size should only be updated when it is believed that the statistics have changed dramatically.

Our implementation follows the *append-only* and *columnar storage* design as used in SciDB [18]. To illustrate, we show the storage of three tuples, colored in yellow, red and green respectively, in Figure 3.5, which each have two dimension attributes, (x_loc, y_loc),

(A)

	id	x_loc	y_loc	luminosity
t0	0	2.5	1	N(32,3)
t1	1	N(1.8,1.1)	N(2,0.6)	N(21,3)
t2	2	N(5,1)	N(4.5,1)	N(17,3)

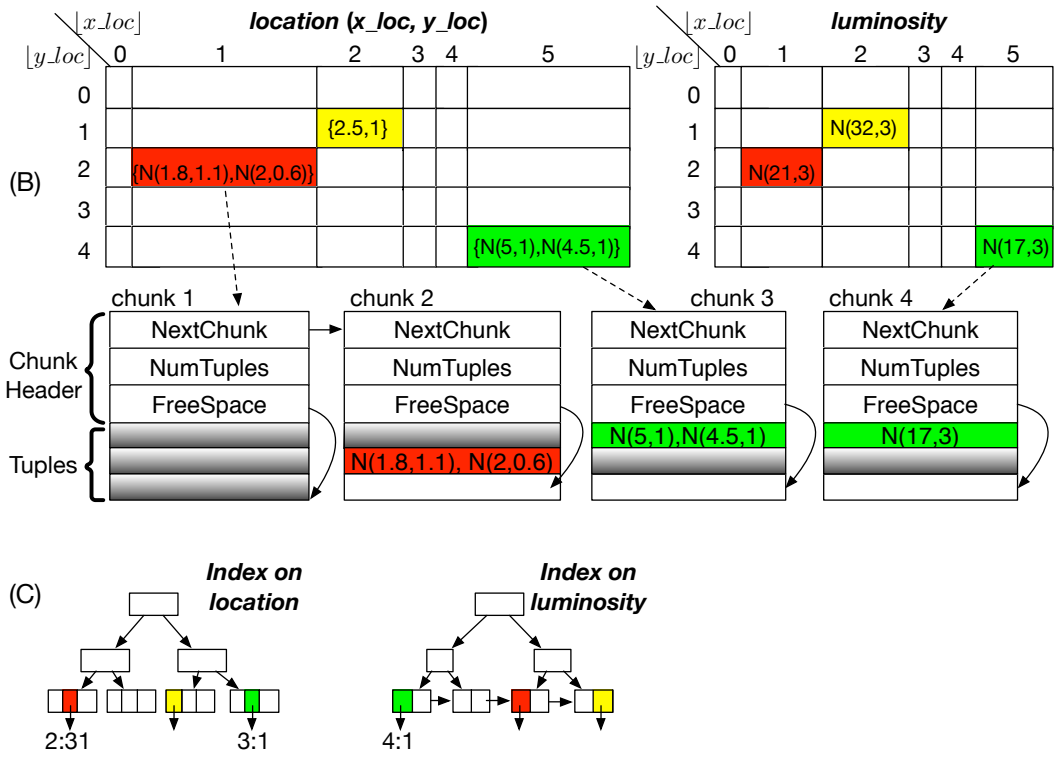


Figure 3.5: Implementation details

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 each dimension attribute have integer index values. In real applications, the discretization function is usually the unit of some metrics, e.g. centimeter or meter for length. We will discuss the selection of the unit when we introduce physical storage later. 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*. Figure 3.5(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). Since each non-empty cell has at least one chunk, when the length of each cell is too small, most cells' physical chunks will not be well-utilized. On the other hand, the cell length being so large that each cell has a lot of chunks is not appropriate when most queries are of high precision requirement (i.e., only a small portion of the chunks contain true query results). In real applications, we could follow heuristics like 70% of the physical chunks are half-full. 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 Figure 3.5(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 Figure 3.5(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 Figure 3.5(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 Figure 3.5(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.

3.3 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 below.

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 Section 3.1.

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 [45]. 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 Section 3.3.

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 $|\mathbb{A}.ra - \mathbb{B}.ra| < r/|\cos(\mathbb{A}.dec) + \varepsilon|$ [45]. 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 Section 3.3 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.

The default evaluation strategy, as stated in Definition 8, 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 [24, 23, 71] can be built on top of the *store-mean* scheme. As we will show in Section 3.4.3, 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.

3.3.1 Subarray-based Join

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

Similar to block nested loops joins, *Structure-Join* can be transformed into iterative *Subarray* operations on the inner array, for each block of the outer array. Assume that the smaller array, $\mathbb{A}$, is the outer. For each cell $C_{\mathbb{A}}$, we do the following: (1) Load it into memory, form a subarray condition $\theta_{C_{\mathbb{A}}}$ on the inner array $\mathbb{B}$, and run the *Subarray* query on $\mathbb{B}$. (2) Pair tuples in $C_{\mathbb{A}}$ with those tuples retrieved by *Subarray* on $\mathbb{B}$. (3) The final validation phase computes the exact probability for each tuple pair $(t_{\mathbb{A}}, t_{\mathbb{B}})$ to satisfy the join condition and compares it with the threshold λ. We describe the subarray condition $\theta_{C_{\mathbb{A}}}$ and the full algorithm in Section 3.3.1.1 and present a cost model for optimization in Section 3.3.2.

3.3.1.1 Subarray Conditions and the SBJ Algorithm

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.

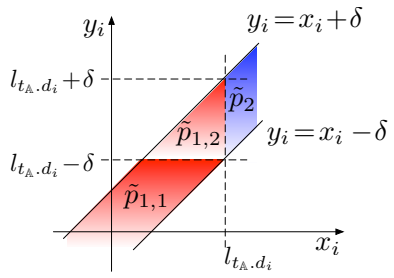


Figure 3.6: Illurstraion of $\tilde{p} = \tilde{p}_1 + \tilde{p}_2 + \tilde{p}_3$.

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 3.3.1. *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 $(i = 1, \ldots, d)$.*

In Section 3.1.1, we define a tuple's possible range as a hyper-rectangle within which the tuple existence probability is (approximately) 1. Before we prove Proposition 3.3.1, let us define it formally.

Definition 13 (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 3.3.1.

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 $\left(l_{t_{\mathbb{A}}}^{d_i} - \delta,\ u_{t_{\mathbb{A}}}^{d_i} + \delta\right)$ and $\left(l_{t_{\mathbb{B}}}^{d_i}, u_{t_{\mathbb{B}}}^{d_i}\right)$ do not

overlap, i.e., $l_{t_{\mathbb{A}}}^{d_i} - \delta > u_{t_{\mathbb{B}}}^{d_i}$ or $u_{t_{\mathbb{A}}}^{d_i} + \delta < l_{t_{\mathbb{B}}}^{d_i}$. Without loss of generality, let us assume $l_{t_{\mathbb{A}}}^{d_i} - \delta > u_{t_{\mathbb{B}}}^{d_i}$. Below we focus on computing probability $p = \iint_{\theta} f_{t_{\mathbb{A}}}(\mathbf{x}) f_{t_{\mathbb{B}}}(\mathbf{y}) \mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y}$ where the integration domain θ is $\{(\mathbf{x}, \mathbf{y}) | \bigwedge_{i=1}^{d} |x_i - y_i| < \delta\}$. 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 Figure 3.6 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.

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

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

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

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. □

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 3.3.2 (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 3.3.3 (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:*

$$\bigwedge_{i=1}^{d} l^{d_i}_{C^-_{\mathbb{A}}} - \delta < \mathbb{B}.d_i < u^{d_i}_{C^+_{\mathbb{A}}} + \delta.$$

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 $\left(l^{d_i}_{t_{\mathbb{A}}}, u^{d_i}_{t_{\mathbb{A}}}\right) \subseteq \bigcup_{C_{\mathbb{A}} \in S_{t_{\mathbb{A}}}} \left(l^{d_i}_{C^-_{\mathbb{A}}}, u^{d_i}_{C^+_{\mathbb{A}}}\right)$: 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 $\left(l^{d_i}_{t_{\mathbb{A}}}, u^{d_i}_{t_{\mathbb{A}}}\right) \subseteq \left(l^{d_i}_{C^-_{\mathbb{A}}}, u^{d_i}_{C^+_{\mathbb{A}}}\right)$ because otherwise it needs at least two copies. When $t_{\mathbb{A}}$ has more than one copy on dimension d_i, according to Section 3.2.2, 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 Figure 3.4(a). So $l^{d_i}_{t_{\mathbb{A}}} = \min_{C_{\mathbb{A}} \in S_{t_{\mathbb{A}}}} l^{d_i}_{C^-_{\mathbb{A}}}$ and $u^{d_i}_{t_{\mathbb{A}}} = \max_{C_{\mathbb{A}} \in S_{t_{\mathbb{A}}}} u^{d_i}_{C^+_{\mathbb{A}}}$, i.e., $\left(l^{d_i}_{t_{\mathbb{A}}}, u^{d_i}_{t_{\mathbb{A}}}\right) = \bigcup_{C_{\mathbb{A}} \in S_{t_{\mathbb{A}}}} \left(l^{d_i}_{C^-_{\mathbb{A}}}, u^{d_i}_{C^+_{\mathbb{A}}}\right)$. Combining the two cases, we have $\left(l^{d_i}_{t_{\mathbb{A}}}, u^{d_i}_{t_{\mathbb{A}}}\right) \subseteq \bigcup_{C_{\mathbb{A}} \in S_{t_{\mathbb{A}}}} \left(l^{d_i}_{C^-_{\mathbb{A}}}, u^{d_i}_{C^+_{\mathbb{A}}}\right)$. Then for any tuple $t_{\mathbb{B}}$, if its possible range $\left(l^{d_i}_{t_{\mathbb{B}}}, u^{d_i}_{t_{\mathbb{B}}}\right)$ overlaps with $\left(l^{d_i}_{t_{\mathbb{A}}} - \delta,\ u^{d_i}_{t_{\mathbb{A}}} + \delta\right)$, which is a necessary condition for $t_{\mathbb{B}}$ being a true match of $t_{\mathbb{A}}$ according to Proposition 3.3.1, it must also overlap with $\bigcup_{C_{\mathbb{A}} \in S_{t_{\mathbb{A}}}} \left(l^{d_i}_{C^-_{\mathbb{A}}} - \delta, u^{d_i}_{C^+_{\mathbb{A}}} + \delta\right)$. 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. □

We now present subarray-based join (SBJ) in Algorithm 1 and illustrate it with Figure 3.7. The algorithm processes one block of the outer at a time (Line 1 in Algorithm 1; marked as Step 1 in Figure 3.7, 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

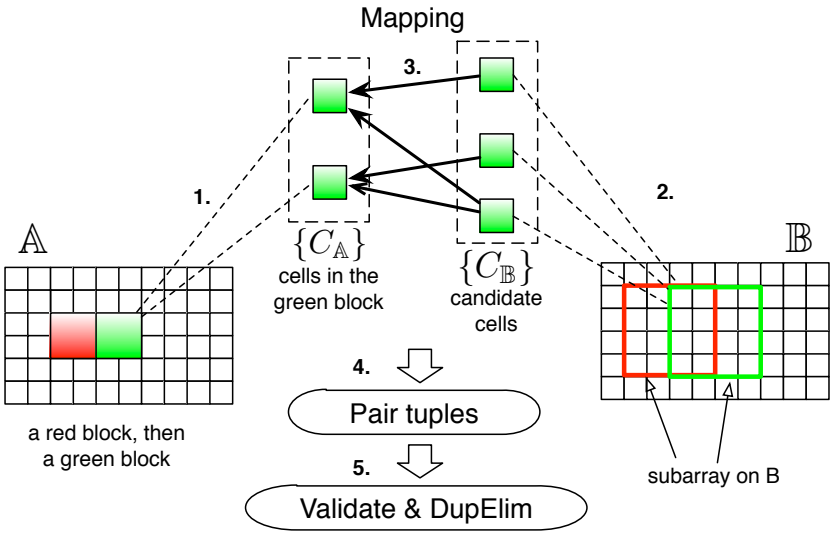


Figure 3.7: Illustration of subarray-based join.

$\mathbb{B}$ (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 Figure 3.7. 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 Figure 3.7 (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).

3.3.2 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 3.1 with subscripts to distinguish inner and outer arrays.

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();
```

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. (3.2) in Section 3.2.3, we have:

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

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 Figure 3.7). 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 3.3.3, 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. (3.1). 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:

$$\beta_{C_{\mathbb{A}}} = \prod_{i=1}^{d} \left(\frac{(1+2k_{\mathbb{A},i})s_{\mathbb{A},i}+2\delta}{s_{\mathbb{B},i}} + 1 + 2k_{\mathbb{B},i} \right) \tag{3.6}$$

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. (3.6), 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. (3.6), 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.

3.4 Experimental Evaluation

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) [92].

3.4.1 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 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.

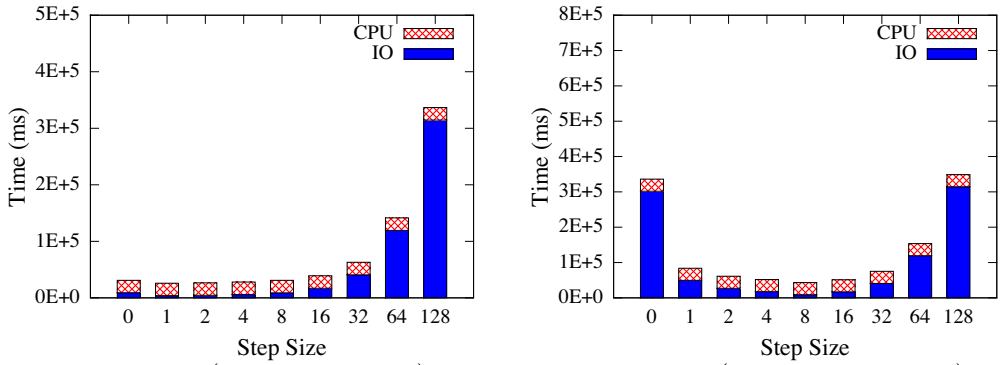
	Parameter	Default Value	Other Values
Data	dimensionality	2	3
	D_μ, distribution of μ	uniform ($\mathcal{U}$)	normal ($\mathcal{N}$)
	S_σ, scale factor of σ	1	16, 100
Query	q, query range / domain	1%	0.01%, 0.1%, 10%
	λ, probability threshold	0.9	0.01

Table 3.2: Parameters in *Subarray* experiments.

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 3.2. 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.[2] 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 [71] and U-index [24,

[2]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.



(a) Workload $(2D, \mathcal{U}, 1; 1\%, 0.9)$ (b) Workload $(2D, \mathcal{U}, 100; 1\%, 0.9)$

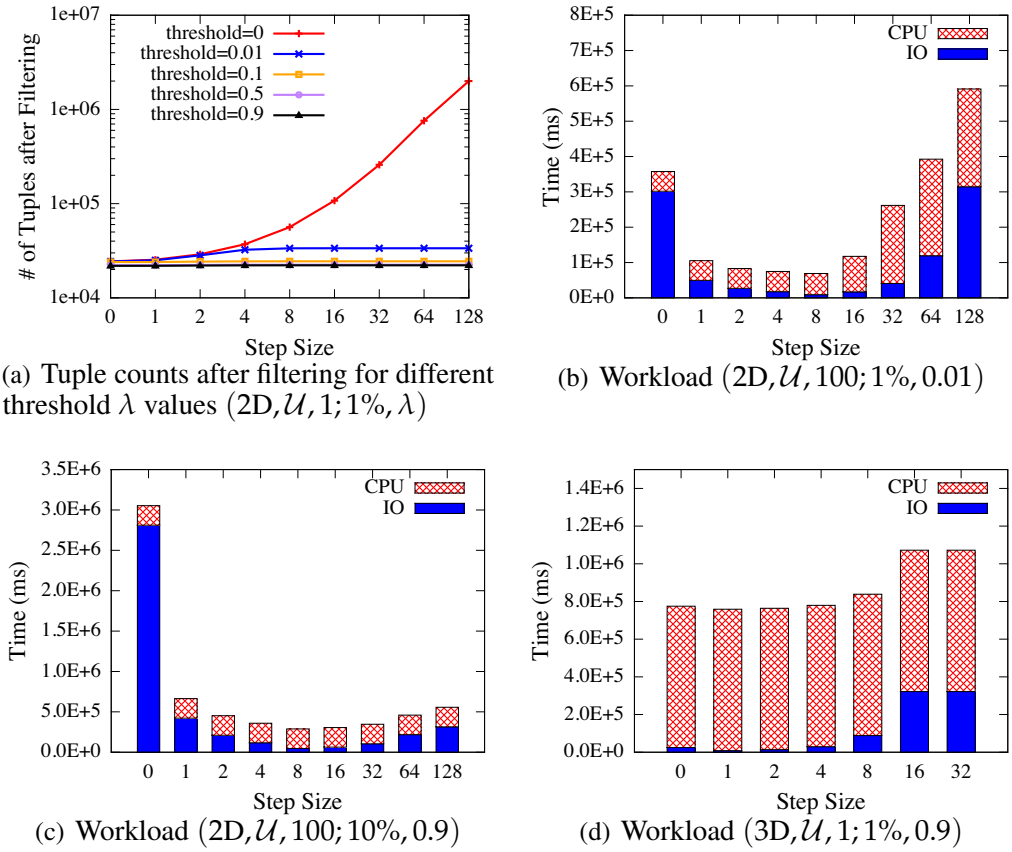
Figure 3.8: Cost breakdown of *Subarray* with varied step sizes for various workloads.

93], 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.

3.4.2 Evaluation of Our Subarray Techniques

We configure *Subarray* queries using the parameters in Table 3.2: 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 [71] 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. Figure 3.8(a) shows results for the default



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

(b) Workload $(2D, \mathcal{U}, 100; 1\%, 0.01)$

(c) Workload $(2D, \mathcal{U}, 100; 10\%, 0.9)$

(d) Workload $(3D, \mathcal{U}, 1; 1\%, 0.9)$

Figure 3.9: Cost breakdown of *Subarray* with varied step sizes for various workloads.

workload, (2D, D_μ=$\mathcal{U}$, S_σ=1; q=1%, λ=0.9), while Figure 3.8(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.

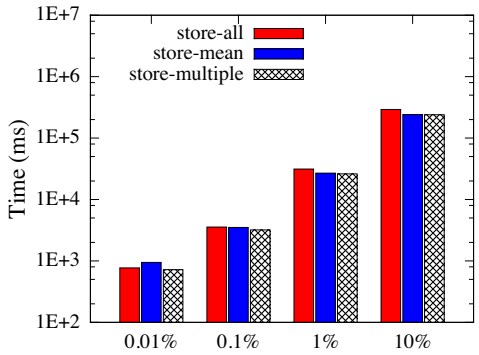
(2) *The CPU cost does not change with the step size when the probability threshold λ is high.* The CPU cost depends on the number of tuples that passed the quick filter and need

to be validated using expensive integration. Figure 3.9(a) shows the number of tuples that pass the filter. When λ is sufficiently high, ≥ 0.1 in this figure, the filter can drop most irrelevant tuples, so the number of tuples after filtering does not change with the step size, or the number of tuples retrieved from storage. We examined the filter's effect using multiple datasets and our observation is consistent.

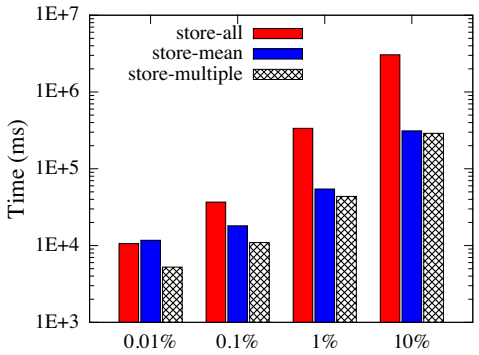
To further study the effect of λ and q, we tune their values from Figure 3.8(b) one at a time: we change λ from 0.9 to 0.01 and show the cost breakdown in Figure 3.9(b); we also change q from 1% to 10% in Figure 3.9(c). Finally Figure 3.9(d) shows the cost breakdown for a 3D workload. It can be seen from these plots that, between CPU cost and I/O cost, which is dominating depends on many factors, including λ (by comparing Figure 3.8(b) and Figure 3.9(b)), the system constants like the per integration cost (by comparing Figure 3.8(a) and Figure 3.9(d)), 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 Figure 3.8(a) and 3.8(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 Section 3.2.3 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 $2\times2\times2$ for 3D datasets that evenly scattered over the array, and report on the average.

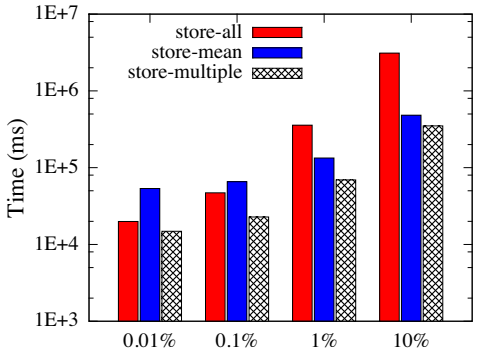
Table 3.2 shows 144 combinations of parameters. Our model returns the optimal step size (i.e., no performance loss) in 89.6% of workloads when the tuples' μ values are uniformly distributed and in 83.3% of workloads when the tuples' μ values are normally distributed.



(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 $(2D, \mathcal{U}, 100; q\%, 0.01)$

Figure 3.10: Evaluation results of *Subarray* on synthetic datasets.

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 Figure 3.10(a)-3.10(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 Figure 3.10(a). However, for datasets when $S_\sigma = 100$, *store-*

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%**

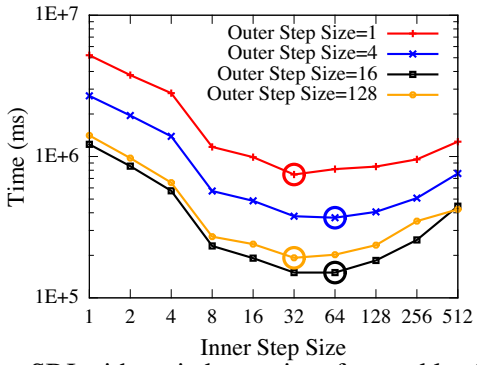
Table 3.3: SBJ Model Accuracy when $\delta = 1\%$

all often incurs tremendous storage overheads and I/O costs in querying, as shown in Figure 3.10(b) and 3.10(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*.

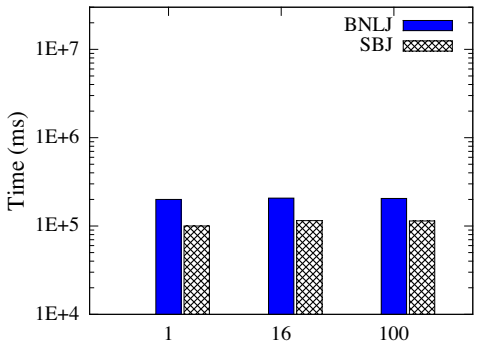
3.4.3 Evaluation of Structure-Join

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 tuples.) We use a recent index on continuous uncertain data [71] 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.

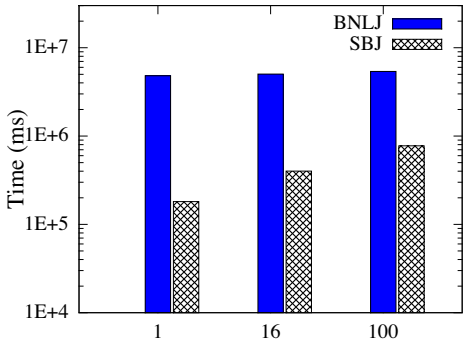
Expt 4: Subarray-Based Join (SBJ). We fix δ to 1% of the domain. SBJ incurs the I/O cost for running repeated *Subarray* queries on the inner array, and the CPU cost for filtering [71]. We find that allocating most memory to the outer block and its mapping structure works the best and use this scheme in all experiments below.



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



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



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

Figure 3.11: Evaluation results *Structure-Join* on synthetic datasets.

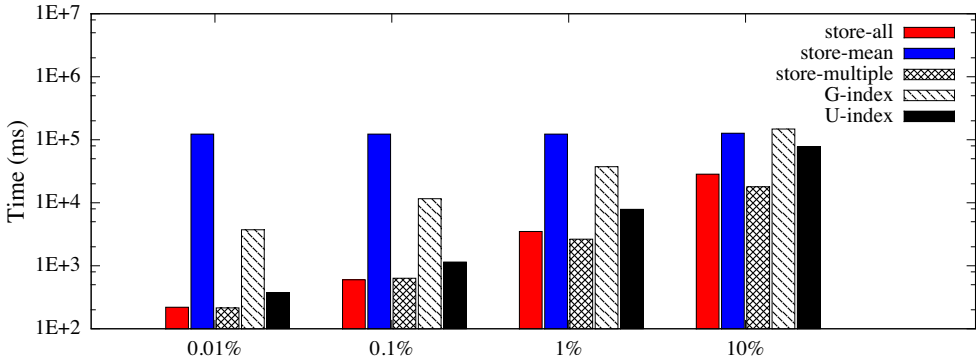
We first demonstrate that SBJ's performance is sensitive to the storage scheme. Figure 3.11(a) shows various combinations of the outer step size, k_{out}, and inner step size, k_{in}, with $\lambda = 0.9$. Each line represents a fixed value of k_{out}, and the x-axis varies values of k_{in}, with the 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 Section 3.3.2 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 3.3, 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).

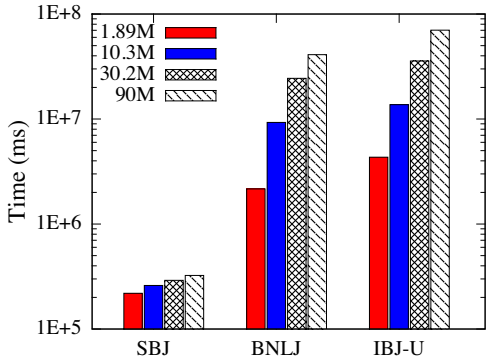
Expt 5: Comparison of Join Algorithms. We now use the step size returned by the model to configure subarray-based join (SBJ), and compare it to a baseline, block nested loops join (BNLJ) where both inner and outer arrays are stored using *store-mean*. Figure 3.11(b) and Figure 3.11(c) show the results 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.

3.4.4 A Case Study using SDSS

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.



(a) *Subarray* on SDSS (λ=0.9, varied q)



(b) *Structure-Join* on SDSS ($\delta = 1$, $\lambda = 0.9$)

Figure 3.12: Case study on SDSS datasets.

We evaluate our techniques for *Subarray* and *Structure-Join* against two state-of-the-art indexes for uncertain data, G-index [71] and U-index [24, 93].

G-index 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

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 3.4: Cache size and node counts for different datasets.

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

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.

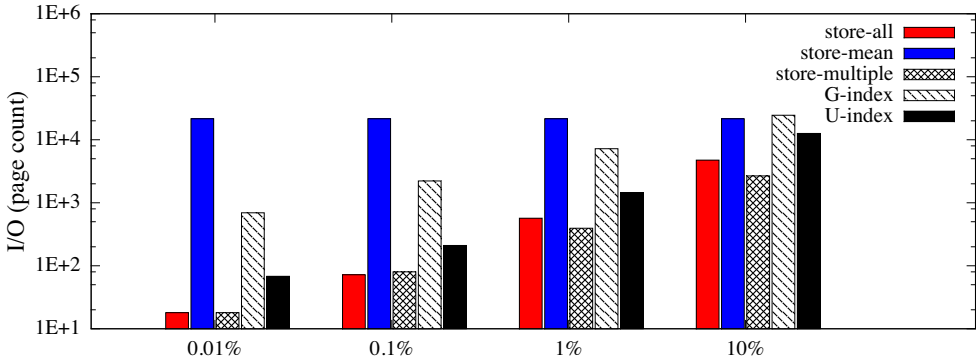
U-index is a variant of R-tree on multi-dimensional uncertain data (e.g., x_loc and y_loc), with each node storing statistical information (i.e., probabilistically constrained rectangles and side lengths) to facilitate queries on uncertain data.

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 [93]). Table 3.4 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.

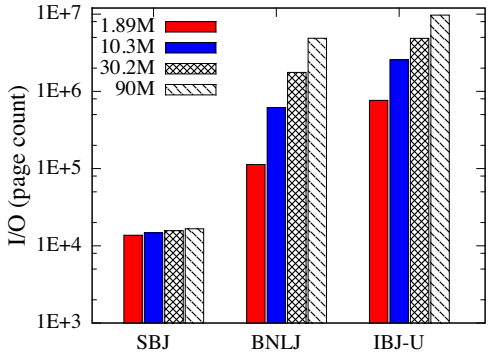
Expt 6: Storage. We first compare our storage schemes with alternative indexing schemes. Table 3.5 shows the disk space that each data structure on the dimension attributes takes. We see that *store-multiple* configured with step size $\langle 1, 1 \rangle$ by our cost model incurs much less storage 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 Figure 3.12(a). (1) *Comparison to results of synthetic data*: The comparison among storage schemes is similar to Figure 3.10(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 Figure 3.10(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.

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 Figure 3.13. The observations are consistent with Figure 3.12(a) and Figure 3.12(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 3.13: I/O counts of *Subarray* and *Structure-Join* on SDSS datasets.

Expt 8: Structure-join. We finally consider a query used in SDSS's sample query set to find neighboring objects:

```
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
```

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. 3.4. For IBJ, as many non-leaf nodes as the memory 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 Figure 3.12(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 3.3.3 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.

3.5 Related Work

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

Probabilistic processing under the array model. Recent work [43] 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 [42]. 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., [30, 5, 97]) and efficient query processing (e.g., [82, 101, 71, 11]). Systems such as ORION [21, 22, 24, 23] and CLARO [71, 97] 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 [21, 24] and more

recent work [93, 57, 41, 71]. They can be leveraged in array databases as well, but can trigger many index I/O's (as we showed in Section 2.5) 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 [15, 1, 78, 4, 3] are designed for similarity and nearest-neighbor queries, not directly applicable to our work. [69] 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 [81, 50, 6, 103]. 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 [6]. 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 [93, 28, 39, 99] 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.

3.6 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).

CHAPTER 4

EXPLORE-BY-EXAMPLE FOR INTERACTIVE DATA EXPLORATION

In previous chapters, we have discussed handling data uncertainty in both relational databases and array databases. In this chapter, we focus on the problem of "query uncertainty" to bridge the increasing gap between the growth of data and the human ability to comprehend data. To help the user retrieve high-value content from large amount of data effectively under query uncertainty, we propose a new database service for interactive exploration in a framework called "*explore-by-example*". In this service, the database system requests user feedback on strategically collected database examples through a series of "conversations" (or iterations). In each iteration, the user characterizes a database sample as relevant or irrelevant to her interest. The user feedback is incorporated into the system to build a *user interest model*. The model is then used in the next iteration to steer the user towards a new area in the data space, and further improved using the user label of a new sample from that area. Eventually, the model characterizing the relevant objects is turned into a *user interest query* that will retrieve all relevant objects from the database.

We present a new active learning algorithm for a common class of user interest queries with a convex shape in Section 4.2. Then, we propose two stopping criteria in the itervative exploration process in Section 4.2. We further improve the efficiency to retreive the most uncertain example and the final result in Section 4.3. Finally, we evaluate our techniques in Section 4.4.

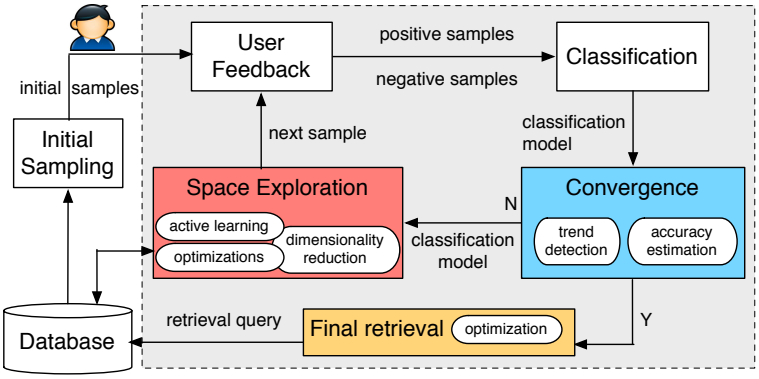


Figure 4.1: System architecture for explore by example.

4.1 Background

In this section, we begin by reviewing our system designed for example-by-example. We then present background on the SVM classification model and basic active learning theory.

4.1.1 System Overview

Our data exploration system is depicted in Figure 4.1. The main concepts and modules are described as follows.

Data space. When a user comes to explore a database, she is presented with the database schema for browsing. Based on her best understanding of the (implicit) exploration goal, she may opt to choose a set of attributes, $\{D_i\}$, $i = 1, \ldots, d$, from a table for consideration[1]. These attributes form a superset of the relevant attributes that will be eventually discovered to characterize the true user interest, but it is the task of the system to discover those relevant attributes. Let us consider the projection of the underlying table to $\{D_i\}$, and pivot the projected table such as each D_i becomes a dimension and the projected tuples are mapped to points in this d-dimensional space – the resulting space is called a *data space* where the user exploration will take place.

[1]If these attributes come from different base tables, we assume that there is a materialized view that stores the related join results in a single table.

Initial examples. To bootstrap data exploration, the user is asked to give an initial positive example and an initial negative example to illustrate her interest. If the user does not have such examples at hand, the system can run an initial sampling algorithm over the data space, as proposed in prior work [34, 62], to help the user find such examples. Since the initial sampling problem has been studied before, our work in this paper focuses on data exploration after such initial examples are identified.

Iterative feedback, learning, and exploration. The iterative exploration process starts with a given positive sample set and a negative sample set, initially each of size one. These samples are called labeled samples. In each iteration, the labeled samples are used as the training set of a classification model that characterizes the user interest (*user interest model*). Before the model reaches convergence or a user-specified accuracy level, the model is used next to navigate the user further in the data space (*space exploration*). In particular, it is used to identify a promising data area to be considered further and to retrieve the next sample from this area to display to the user. In the next iteration, the user labels this sample as positive or negative – such feedback can be collected explicitly through a graphical interface [33], or implicitly based on whether a user clicks on the sample for reviewing or how long she examines the sample.[2] The newly labeled sample is added to an existing labeled sample set, and the above process repeats.

Convergence and final retrieval. At each iteration, our system assesses the current classification model to decide whether more exploration iterations are needed. The process is terminated when the model has exhibited the trend of convergence, or the accuracy of the model reaches a user-defined threshold. At this point, the classification model for its positive class is translated to a query which will retrieve from the database all the objects characterized as relevant. As can be seen, in our work the user interest is characterized by a

[2]This topic is in the purview of human-computer interaction and hence is beyond the scope of this paper, which focuses on uncertainty sampling.

classification model, which is eventually translated to a database query. Therefore, we use the terms, "*user interest*", "*classification model*", and "*query*", interchangeably.

Our work differs from prior work in several aspects. First, the state-of-the-art system on explore-by-example, AIDE [34, 35], used decision trees to build a classification model due to the natural descriptive power of the learned model. However, this approach cannot handle complex user interests characterized by non-linear patterns in the data space. To approximate the circle pattern, AIDE may need to use dozens of range predicates connected by logical *and* and *or* operators, and require the user to label many samples to achieve high accuracy. Therefore, in this work we choose Support Vector Machines (SVMs) to build the classification model because they can handle both linear and non-linear patterns.

Second, active learning theory [16] has suggested choosing the example closest to the decision boundary of the SVM model in each iteration of exploration. In this work we leverage SVM active learning theory to select examples shown to the user. However, existing implementations either scan the entire database [26], which is prohibitively expensive for a large database, or resort to random sampling [16], which cannot strike a balance between accuracy and efficiency. In addition, most active learning work lacks provable results on accuracy at a given iteration of exploration [94, 79]. Recent theoretical work offers provable bounds on classification errors [19, 38, 47, 48, 49], which treat positive and negative classes equally and hence are not practical for use. It is because the true user interest over a large database often amounts to a highly selective query. To learn a classification model for the query, we must emphasize the errors related to the objects in the positive class, i.e., the answers returned by the query. Consider a user interest query with 1% selectivity. A classifier that classifies all database objects to the negative class has a low error rate of 1%, but fails to return any relevant objects to the user. For this reason, we choose *F1-score* as a proper accuracy measure for database exploration because it emphasizes the accuracy regarding the positive class. Active learning theory lacks formal convergence results for this measure.

4.1.2 Support Vector Machines

Since non-linear predicates are prevalent in scientific applications and location-based searches, we seek to support user interests involving both linear and non-linear predicates. In this work, we adopt Support Vector Machines (SVM) to build the classification model. SVM is a **linear classifier** that makes a classification decision based on the value of a linear combination of the dimensions of the data space. To support non-linear patterns in the data space, it uses the *kernel method* to map the user labeled samples into a much higher dimensional space, called the *feature space*, where linear separation can be achieved. In this work we use the Gaussian kernel, and denote the decision boundary in the feature space as $\mathcal{L}$.

For classification, the learning algorithm for SVM takes a set of training examples in the *data space*, each labeled using one of the two output classes, and builds a binary classifier that assigns labels for the new test examples in the same space. SVM is a **linear classifier** in the sense that it makes a classification decision based on the value of a linear combination of an example's characteristics. Interestingly, the algorithm works not only when the training examples from different classes are linearly separable in the data space, but also when they are not. In the latter case, the examples will be mapped into a much higher-dimensional space called the *feature space*, where linear separation can be achieved. Among the many hyperplanes that linearly separate the examples from different classes either in the data space or in the feature space, the one with the largest distance to the nearest (mapped) examples is returned by the algorithm as the *decision boundary*. In general, the larger the margin the lower the generalization error of the classifier. Therefore, SVMs are also called **large margin classifiers**.

Formally, the decision boundary in the feature space can be described as

$$y(\boldsymbol{x}) = \boldsymbol{\omega}^T \phi(\boldsymbol{x}) + b = 0, \tag{4.1}$$

where $\boldsymbol{x}$ denotes a point in the data space and $\phi(\boldsymbol{x})$ is its mapped value in the feature space. We can select two hyperplanes parallel to the decision boundary such that there are no examples between them, and then try to maximize their distance. Without loss of generality, we select $\boldsymbol{\omega}^T\phi(\boldsymbol{x}) + b = 1$ and $\boldsymbol{\omega}^T\phi(\boldsymbol{x}) + b = -1$, the distance between which is $\frac{2}{||\boldsymbol{\omega}||}$, and we can state the constraints that $\boldsymbol{\omega}^T\phi(\boldsymbol{x}) + b \geq 1$ for all positive examples and $\boldsymbol{\omega}^T\phi(\boldsymbol{x}) + b \leq -1$ for all negative examples. Maximizing the distance $\frac{2}{||\boldsymbol{\omega}||}$ is equivalent to minimizing $||\boldsymbol{\omega}||$, or $\frac{1}{2}||\boldsymbol{\omega}||^2$ for mathematical convenience without changing the solution for $\boldsymbol{\omega}$ and b. Putting it all together, we have the following optimization problem:

$$\begin{aligned} \underset{\boldsymbol{\omega},b}{\text{minimize}} \quad & \frac{1}{2}||\boldsymbol{\omega}||^2 \\ \text{subject to} \quad & \bar{y}_i(\boldsymbol{\omega}^T\phi(\boldsymbol{x}_i) + b) \geq 1 \quad i = 1, \ldots, n. \end{aligned} \tag{4.2}$$

where $\bar{y}_i$ is the true label of a training point $\boldsymbol{x}_i$. The above is an optimization problem with a convex quadratic objective and only linear constraints. It can be solved using quadratic programming (QP) and the solution gives us the optimal margin classifier.

To ensure linear separation of training examples, sometimes feature spaces may have an exponential or even infinite number of dimensions, which would make it seem impossible to provide efficient computation [84]. The main theory of SVMs states that one can solve the *dual*, which is derived by introducing Lagrange multipliers α_i, in lieu of the *primal* problem. The dual optimization problem is a maximization problem with parameters being the α_i's:

$$\begin{aligned} \underset{\boldsymbol{\alpha}}{\text{maximize}} \quad & \sum_{i=1}^{n} \alpha_i \bar{y}_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j \phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}_j) \\ \text{subject to} \quad & \sum_{i=1}^{n} \alpha_i = 0 \end{aligned} \tag{4.3}$$

It can be derived that the α_i's are zero except for the *support vectors*, defined to be the training examples that are on the margin. We do not show the derivation of the dual problem

and its solution here due to space constraints, but one important intermediate formula that was derived previously and will be used for our later derivation is:

$$\boldsymbol{\omega} = \sum_{i=1}^{n} \alpha_i \phi(\boldsymbol{x}_i) = \sum_{\phi(\boldsymbol{x}_i) \in \boldsymbol{S}} \alpha_i \phi(\boldsymbol{x}_i) \tag{4.4}$$

where $\boldsymbol{S}$ refers to the set of support vectors. Plugging Eq. (4.4) to Eq (4.1), the decision boundary can be rewritten as

$$y(\boldsymbol{x}) = \sum_{\phi(\boldsymbol{x}_i) \in \boldsymbol{S}} \alpha_i \phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}) + b. \tag{4.5}$$

Now the decision boundary requires merely the inner product between $\phi(\boldsymbol{x})$ and each support vector $\phi(\boldsymbol{x}_i)$.

Another benefit of the dual problem relates to the so-called Kernel trick. Given a mapping ϕ, a **kernel** is defined to be $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}_j)$. Any proposed kernel function must be validated by Mercer's theorem [65]. Most notably, $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ can be inexpensive to calculate because it applies to the variables in the data space, even when $\phi(\boldsymbol{x})$ may be very expensive to calculate due to a high dimensionality. In this case, SVMs can be learned without ever having to explicitly find or represent vectors $\phi(\boldsymbol{x})$.

Commonly used kernels include the linear kernel, polynomial kernel, and Gaussian kernel (a.k.a., radial basis function kernel). Without prior knowledge of what the user interest may be, the Gaussian kernel is considered more flexible than the linear or polynomial kernels, hence used in this work. The Gaussian kernel is defined as: $K_G(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp(-\gamma||\boldsymbol{x}_i - \boldsymbol{x}_j||^2)$. The feature space of the Gaussian kernel is known to be of an infinite number of dimensions [87].

4.1.3 Active Learning for SVM

Our problem of dynamically seeking the next sample to label from a large database of unlabeled objects is closely related to active learning. The active learning framework for

Algorithm 2 A Basic Framework for SVM Active Learning

Input: database D

```
1: D_init ←initialSampling(D)
2: D_labeled ← getUserLabel(D_init)
3: D_unlabeled ← D \ D_init
4: model ← trainSVM(D_labeled)
5: while !isTerminated() do
6:    x ← getNextToLabel(model, D_unlabeled)
7:    {x'} ←getUserLabel({x})
8:    D_labeled ← D_labeled ∪ {x'}
9:    D_unlabeled ← D_unlabeled \ {x}
10:   model ← trainSVM(D_labeled)
11: finalRetrieval(model, D)
```

SVM has a simple structure as shown in Algorithm 2. It starts with initial sampling and proceeds to space exploration in an iterative fashion. In both phases, users are asked to provide labels of retrieved samples (line 2 and line 7). The key focus of active learning is at line 6: to identify at each iteration, the next sample to label to quickly improve the accuracy of the current SVM model. Recent active learning theory [16] proposed to choose in each iteration the *example closest to the current decision boundary*, that is, $min_x f(\phi(\boldsymbol{x}), \mathcal{L})$, where x refers to any point in the data space, $\phi(\boldsymbol{x})$ is its mapping to the feature space, and f is the distance function in the feature space.

Our work follows the same framework and uses the above theory to meet the "most uncertain" requirement of uncertainty sampling. However, our work instantiates this framework with new sampling algorithms to also meet the efficiency and convergence requirements. In particular, we propose new algorithms that address `getNextToLabel()` and `isTerminated()` together in Section 4.2, and a series of optimizations for `getNextToLabel()` and `finalRetrieval()` in Section 4.3.

4.2 New Sampling Algorithms

In this section, we present two uncertainty sampling algorithms that provide rigorous yet practical results on the convergence of the user interest model.

4.2.1 Algorithm for Convex Queries

Our first algorithm focuses on a common class of user interest queries that have a convex shape in the data space. For this class of queries, we seek to design a sampling algorithm that is more efficient than a direct implementation of SVM active learning theory (Algorithm 2), and further enables formal provable results on F1-score as the accuracy measure of the SVM classification model in a given iteration in exploration.[3]

Formally, F1-score is evaluated on a *test set* $D_{test} = \{(\boldsymbol{x}_i, y_i)\}$, where $\boldsymbol{x}_i$ denotes a database object and y_i denotes its label according to the classification model. Then F1-score is defined as:

$$\text{F1-score} = 2 \cdot \frac{precision \cdot recall}{precision + recall},$$

where *precision* is the fraction of points returned by the model (query on D_{test}) that are positive, and *recall* is the fraction of positive points in D_{test} that are returned by the model.

However, capturing F1-score in our uncertainty sampling procedure is difficult because we do not have such a labeled test, set D_{test}, available. We cannot afford to ask the user to label more to produce one since the user labor is an important concern. The challenge here is how to provide any accuracy information related to F1-score with limited labeled points and abundant unlabeled ones.

The key idea is, at each iteration we try to use all available labeled examples, denoted as $D_{labeled}$, to building a partitioning function of the data space. This function divides the data space into the *positive region* (any point inside which is guaranteed to be positive), the *negative region* (any point inside which is guaranteed to be negative) and the *uncertain region*. We define the *evaluation set* D_{eval} as the projection of D_{test} without labels y_i's. Then for each data point in D_{eval}, depending on which region it falls into, D_{eval} can be partitioned into three sets accordingly, denoted as D^+, D^- and D^u. We can compute a metric from the number of data points in the three sets and prove that it is a lower bound

[3]Our solution to this class also provides a foundation for extension to a more general case of a union of convex shapes.

of F1-score evaluated on D_{test}. As more labeled examples being provided, we have more knowledge about the uncertain region, so part of the uncertain region will convert to either the positive or the negative region in later iterations. Accordingly, some data points can be moved from D^u to either D^+ or D^-. Eventually, with enough training data, the uncertain region shrinks to the minimum, $D^u = \emptyset$, and the positive region converges to the query region.

4.2.1.1 Algorithm with Exact Lower Bound of F1-score

We start with a formal definition of the partitioning function, Ψ, of a data space based on the three regions mentioned above. We then introduce a metric built on this partitioning function, called the three-set metric (TSM), and an algorithm that incorporates Ψ and the TSM metric to active learning. We finally present a theorem stating that the TSM metric captures the lower bound of F1-score.

1. Partitioning Function and Metric. The following definitions and propositions depend on the assumption that the query region Q is convex, which means that any point on the line segment connecting two points $\boldsymbol{x}_1 \in Q$ and $\boldsymbol{x}_2 \in Q$ is also in Q. When Q consists of multiple disjoint convex regions which makes itself not convex, we can initiate one exploration task per region and the metric applies to each task.

Definition 14 (Positive Region)**.** *Denote the examples that have been labeled as "positive" as e_i^+, $i = 1, \ldots, n^+$. The convex hull of $\cup_{i=1}^{n^+} e_i^+$, i.e., the smallest convex set that contains $\cup_{i=1}^{n^+} e_i^+$, is called the positive region, denoted as R^+.*

It is known that the convex hull of a finite number of points is a convex polytope [46]. For example, the green triangle in Figure 4.2 and the green pentagon in Figure 4.3 are the positive regions formed by three and five positive examples, respectively, in a two-dimensional space. We can prove the following property of the positive region:

Proposition 4.2.1. *All points in the positive region R^+ are positive.*

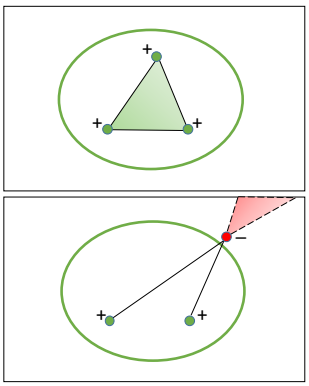


Figure 4.2: Illustration of a positive region (green) with three positive samples, and a negative region (red) with two positive examples and one negative example in 2D space.

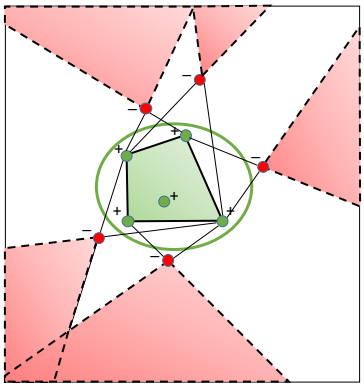


Figure 4.3: Illustration of positive region (green) and negative region (red) with five positive examples and five negative examples in two-dimensional space.

Proof. Since (1) $e_i^+ \in Q$ for $i = 1, \ldots, n^+$, (2) R^+ is the smallest convex set that contains $\cup_{i=1}^{n^+} e_i^+$, and (3) Q is convex, we can derive that $R^+ \subseteq Q$, which means all points in R^+ are guaranteed to be positive. □

Definition 15 (Negative Region)**.** *For a negative example $\boldsymbol{e}_i^-$, we can define a corresponding negative region R_i^- such that the line segment connecting any point $\boldsymbol{x} \in R_i^-$ and $\boldsymbol{e}_i^-$ does not overlap with the positive region R^+, but the ray that starts from $\boldsymbol{x} \in R_i^-$ and passes through $\boldsymbol{e}_i^-$ will overlap with R^+. More formally, $R_i^- = \{\boldsymbol{x} | \overline{\boldsymbol{x}\boldsymbol{e}_i^-} \cap R^+ = \emptyset \wedge \overrightarrow{\boldsymbol{x}\boldsymbol{e}_i^-} \cap R^+ \neq \emptyset\}$. Given n^- negative examples, the negative region R^- is the union of the negative region for each negative example, i.e., $R^- = \cup_{i=1}^{n^-} R_i^-$.*

From the definition, we know that R_i^- is a convex cone generated by the conical combination of the vectors from the positive examples to the given negative example, i.e., $\overrightarrow{e_j^+ e_i^-}$ $(j = 1, \ldots, n^+)$. Further constrained by the bounds of the data space, each negative region becomes a convex polytope. For example, the red triangle in Figure 4.2 and five red polygons in Figure 4.3 are the negative regions in a two-dimensional space. We can prove the following property of the negative region:

Proposition 4.2.2. *All points in the negative region R^- are negative.*

Proof. Let us first prove that all points in each R_i^- are negative. Suppose that some point $\boldsymbol{x}_0 \in R_i^-$ is positive. According to the definition of R_i^-, $\overrightarrow{\boldsymbol{x}_0 \boldsymbol{e}_i^-} \cap R^+ \neq \emptyset$, which means we can find a point $\boldsymbol{x}_1$ such that $\boldsymbol{x}_1 \in R^+$ and $\boldsymbol{x}_1 \in \overrightarrow{\boldsymbol{x}_0 \boldsymbol{e}_i^-}$. Then $\boldsymbol{e}_i^-$ is on the line segment connecting two positive points $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$. This contradicts the convex query assumption. Hence the supposition is false and all points in R_i^- are negative. Since R^- is just a union of all R_i^-'s, all points in R^- are negative as well. □

Definition 16 (Uncertain Region)**.** *Denote the data space as $\mathbb{R}^d$, the uncertain region $R^u = \mathbb{R}^d - R^+ - R^-$.*

Basically R^u is the remaining region, e.g., the white area in Figure 4.2 and Figure 4.3. As mentioned earlier, as more examples being labeled, part of the uncertain region will be converted to either the positive region or the negative region and eventually the uncertain region will shrink to an empty set.

With the three types of regions defined, we can define the three-set metric as follows:

Definition 17 (Three-set Metric)**.** *Denote $D^+ = D_{eval} \cap R^+$, $D^- = D_{eval} \cap R^-$, $D^u = D_{eval} \cap R^u$, and $|S|$ means the size of set S. At a specific iteration of exploration, the three-set metric is defined to be $\frac{|D^+|}{|D^+|+|D^u|}$.*

2. Algorithm. Next we present how to build the TSM metric in the iterative exploration process in Algorithm 3. The input is the database D, evaluating dataset D_{eval} (which could

Algorithm 3 SVM Active Learning with TSM for Convex Queries

Input: database D, evaluation set D_{eval}, accuracy requirement λ

1: $R^+ \leftarrow \emptyset$, $R^- \leftarrow \emptyset$
   // initial sampling:
2: $D_{init} \leftarrow$ `initialSampling(`D`)`
3: $D_{labeled} \leftarrow$ `getUserLabel(`D_{init}`)`
4: $D_{unlabeled} \leftarrow D \setminus D_{init}$
   // estimate accuracy of initial sampling:
5: **for** $\boldsymbol{x} \in D_{labeled}$ **do**
6: $(R^+, R^-) \leftarrow$ `updateRegion(`$R^+, R^-, \boldsymbol{x}$`)`
7: $(D^+, D^-, D^u) \leftarrow$ `updateEvalData(`$R^+, R^-, \emptyset, \emptyset, D_{eval}$`)`
8: $accu \leftarrow$ `estAccuracy(`D^+, D^-, D^u`)`
9: $model \leftarrow$ `train(`$D_{labeled}$`)`
   // space exploration:
10: **while** $accu < \lambda$ **and** !`isTerminated()` **do**
11: $\boldsymbol{x} \leftarrow$ `getNextToLabel(`$model, D_{unlabeled}$`)`
12: **if** $\boldsymbol{x} \in R^+$ **then**
13: $\boldsymbol{x}' \leftarrow (\boldsymbol{x}, 1)$
14: **else if** $\boldsymbol{x} \in R^-$ **then**
15: $\boldsymbol{x}' \leftarrow (\boldsymbol{x}, -1)$
16: **else**
17: $\{\boldsymbol{x}'\} \leftarrow$`getUserLabel(`$\{\boldsymbol{x}\}$`)`
      // estimate current accuracy:
18: $(R^+, R^-) \leftarrow$ `updateRegion(`$R^+, R^-, \boldsymbol{x}$`)`
19: $(D^+, D^-, D^u) \leftarrow$ `updateEvalData(`R^+, R^-, D^+, D^-, D^u`)`
20: $accu \leftarrow$ `estAccuracy(`D^+, D^-, D^u`)`
21: $D_{labeled} \leftarrow D_{labeled} \cup \{\boldsymbol{x}'\}$
22: $D_{unlabeled} \leftarrow D_{unlabeled} \setminus \{\boldsymbol{x}\}$
23: $model \leftarrow$ `train(`$D_{labeled}$`)`
24: `finalRetrieval(`$model$`,` D`)`

be any unlabeled dataset including the database D), and a user-defined accuracy threshold λ. First, we initialize R^+ and R^- as empty sets (line 1). Then, initial sampling is performed (line 2) and the obtained examples are labeled by users (line 3). We keep track of the labeled and unlabeled samples using $D_{labeled}$ and $D_{unlabeled}$. Based on the labeled examples, the regions can be incrementally updated (line 5-6) and hence the corresponding sub-partitions of D_{eval} are updated (line 7). The accuracy is then evaluated according to Definition 17 (line 8) and a model is trained based on the labeled initial samples (line 9). The process next goes into the iterative exploration until the accuracy requirement is met or the user decides to stop

Algorithm 4 `updateEvalData` Incrementally update the partitions of D_{eval}

Input: positive and negative region (R^+, R^-). positive, negative and uncertain class of the evaluation dataset (D^+, D^-, D^u).

1: **for** $\boldsymbol{x} \in D^u$ **do**
2: **if** $\boldsymbol{x} \in R^+$ **then**
3: $D^+ \leftarrow D^+ \cup \{\boldsymbol{x}\}$, $D^u \leftarrow D^u \setminus \{\boldsymbol{x}\}$
4: **else if** $\boldsymbol{x} \in R^-$ **then**
5: $D^- \leftarrow D^- \cup \{\boldsymbol{x}\}$, $D^u \leftarrow D^u \setminus \{\boldsymbol{x}\}$
6: **return** (D^+, D^-, D^u)

the process (line 10). In each iteration, an unlabeled data point with the closest distance to the decision boundary is acquired. If the data point is in R^+ or R^-, which means its label is known without involving the user, it is labeled automatically (line 12-15). Such examples do not add information to R^+ and R^- and hence do not change D^+, D^-, D^u and the three-set metric. Otherwise, the data point is labeled by the user (line 17) and the metric is updated (line 18-20). At the end of each iteration, the selected data point is moved from $D_{unlabeled}$ to $D_{labeled}$ (line 21-22) and a new model is trained (line 23).

There are a few procedures involved in Algorithm 3. Here we emphasize those related to the three-set metric: In `updateRegion`, the regions are updated based on the literature in computational geometry [8]. In `updateEvalData`, the partitions of D_{eval} can be incrementally updated as shown in Algorithm 4, which tests if a data example belongs to the positive or negative region based on its polytope definition. In `estAccuracy`, the metric is computed according to Definition 17.

3. Lower Bound of F1-score. With a good understanding of how the metric works in the exploration process, we now present our main theorem.

Theorem 4.2.1. *The three-set metric evaluated on D_{eval} captures a lower bound of the F1-score if evaluated on D_{test}.*

Proof. At any iteration i, D_{eval} can be partitioned into D^+, D^- and D^u. Recall that D_{eval} is a projection of D_{test} without the labels. We know for certain that the labels for all points

in D^+ (or D^-) are positive (or negative) in D_{test} according to Proposition 4.2.1, 4.2.2 and the definition of D^+ and D^- in Definition 17; only the labels for points in D^u are uncertain.

Let us assume that $p\%$ points in D^u are predicted as positive by the SVM model trained at Line 23 of Algorithm 3. Denote the set of points as D^{u+}. Then $|D^{u+}| = p\% \cdot |D^u|$ and we can write the precision and recall of the trained SVM model as

$$
\begin{aligned}
\mathit{precision} &= \frac{|D^+| + |D^{u+} \cap Q|}{|D^+| + |D^{u+}|} = \frac{|D^+| + |D^{u+} \cap Q|}{|D^+| + p\% \cdot |D^u|} \\
&\geq \frac{|D^+|}{|D^+| + |D^u|} \\
\mathit{recall} &= \frac{|D^+| + |D^{u+} \cap Q|}{|D^+| + |D^u \cap Q|} \geq \frac{|D^+|}{|D^+| + |D^u|}
\end{aligned}
$$

F1-score is the harmonic mean of precision and recall. So F1-score is lower-bounded by $|D^+|/(|D^+| + |D^u|)$. □

The three-set metric has several key advantages. First, it provides an *exact lower bound* of F1-score throughout the exploration process and works for any evaluation set D_{eval} Second, the metric is *monotonic* in the sense that points in the uncertain region before may be in the positive or negative region later, and the metric converges to 1 when $D^u = \emptyset$. The monotonicity means that if the metric is above the desired accuracy threshold at some iteration, it is guaranteed to be greater than the threshold in later iterations, so we can safely stop the exploration. Monotonicity also enables incremental computation: at iteration $i + 1$, we only need to check the points in the uncertain region at iteration i and see if they belong to the positive or negative region of iteration $i + 1$.

Besides the lower-bound on F1-score, the TSM algorithm has several advantages over a direct implementation of active learning theory (Algorithm 2): The TSM algorithm internally maintains a positive sample set and a negative set based on its partitioning function of the data space, and avoids asking the user to label the retrieved sample if it is known to belong

to its positive or negative sample set. From the user's perspective, the convergence has expedited because the same accuracy can be achieved by labeling fewer samples. The performance benefits of this optimization can be significant as we show in the evaluation. In addition, since TSM works for any evaluation set D_{eval} in the data space, which can be set to the entire database or a smaller sample set (used in our approximate bound as described shortly).

4.2.1.2 Approximate Lower Bound of the Metric

When D_{eval} is too large, we may refer to a sampling technique to reduce the time to evaluate the three-set metric. Let p and q be the true proportions of the positive and negative data in D_{eval}, i.e., $p = |D^+|/|D_{eval}|$ and $q = |D^-|/|D_{eval}|$. Then the three-set metric is $b = \frac{p}{1-q}$. Let $\hat{p}$ and $\hat{q}$ be the observed proportions of the positive and negative samples in a random draw of n samples from D_{eval}, and let $X_n = \frac{\hat{p}}{1-\hat{q}}$. Our goal is to find the smallest sample size n such that the estimation error of the exact three-set metric is less than δ with probability no less than λ. That is,

$$\Pr(|X_n - b| < \delta) \geq \lambda.$$

The following theorem will help us find the lower bound of n.

Theorem 4.2.2. *$sup_\epsilon \| \Pr(\sqrt{n}|X_n - b| < \epsilon) - \left(2\Phi(\frac{\epsilon(1-q)}{\sqrt{p(1-p-q)}}) - 1 \right) \| = O(1/\sqrt{n})$ for any ϵ, where Φ is the cumulative distribution function of the standard Normal distribution.*

Proof. Since $(n\hat{p}, n\hat{q})^T$ follows Multinomial$(n, p, q, 1-p-q)$, the vector converges to the bivariate Normal distribution when n increases according to the Central Limit Theorem. Specifically,

$$\sqrt{n}\left(\begin{pmatrix} \hat{p} \\ \hat{q} \end{pmatrix} - \begin{pmatrix} p \\ q \end{pmatrix} \right) \xrightarrow{D} N\left(\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \Sigma \right)$$

where $\Sigma = \begin{pmatrix} p & -pq \\ -pq & q \end{pmatrix}$.

Define $v = (p,q)^T$, $\hat{v} = (\hat{p},\hat{q})^T$, and $g(v) = \frac{p}{1-q}$. Then $b = g(v)$ and $X = g(\hat{v})$. According to the Delta method [27]

$$sup_{\epsilon} \| \Pr(\sqrt{n}|X_n - b| < \epsilon) - \int_{-\epsilon}^{\epsilon} \phi_{\sigma^2}(t)\mathrm{d}t \| = O(1/\sqrt{n})$$

where ϕ is the density function of the Normal distribution with mean zero and variance $\sigma^2 = (\partial g(v)/\partial v)^T \Sigma (\partial g(v)/\partial v) = p(1-p-q)/(1-q)^2$. Therefore, the theorem is proved. □

With the above theorem, we approximate the sample size such that

$$2\Phi(\frac{\sqrt{n}\delta(1-q)}{\sqrt{p(1-p-q)}}) - 1 \geq \lambda.$$

Since $p(1-p-q)/(1-q)^2 \leq 1/4$, it is sufficient for n to satisfy $2\Phi(2\sqrt{n}\delta) - 1 \geq \lambda$ and therefore $n \geq \left(\Phi^{-1}\left(\frac{\lambda+1}{2}\right)\right)^2 / (4\delta^2)$.

4.2.2 Algorithm for General Queries

For general query patterns, our second algorithm augments SVM active learning theory with techniques for capturing the model change rate and the trend of convergence based on this measure.

To begin the discussion, we summarize our notations for SVM as follows, while the details on SVM are given in Section 4.1.2: denote $\boldsymbol{x}$ as a point in the data space, ϕ as the mapping function from the data space to the feature space, K as the kernel function where $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}_j)$; the SVM model can be uniquely identified using either $\boldsymbol{\omega}$ and b in the primal form, or $\boldsymbol{\alpha}$ and b in the dual form.

To capture the model change rate, the starting point of our algorithm design is the following observation: with the Gaussian kernel, all the points in the data space $\mathbb{R}^d$ will be mapped

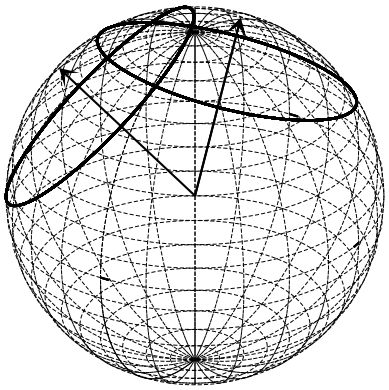


Figure 4.4: Two models (hyperplanes) in the 3-dimensional feature space (hypersphere).

onto a unit hypersphere in the feature space $\mathbb{R}^f$. This is because $K_G(\boldsymbol{x},\boldsymbol{x}) = \exp(-\gamma||\boldsymbol{x}-\boldsymbol{x}||^2) = \exp(0) = 1$. Then for the corresponding ϕ_G, we know $\phi_G(\boldsymbol{x})^T\phi_G(\boldsymbol{x}) = 1$, which means all points in the data space are mapped to a unit hypersphere in the feature space. The decision model at each iteration is a hyperplane that possibly cuts the multi-dimensional ball into one positive hyper-spherical cap and one negative hyper-spherical cap. Figure 4.4 shows two decision models in the feature space in solid circles and the corresponding two $\boldsymbol{\omega}$ vectors pointing to the positive side of the model. Below, we formally define the model change rate in the feature space.

Definition 18 (Model Change Rate)**.** *Denote C_{i-1} and C_i as the positive hyper-spherical caps at iteration $i-1$ and i respectively. We define the difference between C_{i-1} and C_i as $\mathcal{D}_i = (C_i \setminus C_{i-1}) \cup (C_{i-1} \setminus C_i)$ and the model change rate between iteration $i-1$ and i as $A(\mathcal{D}_i)/A(\mathcal{S}^i)$, where $\mathcal{S}^i$ is a unit sphere in i-dimensional space and A is the notation for the surface area.*

The rationale behind this definition is: $C_i \setminus C_{i-1}$ is the set of points in the feature space that are predicted as negative at iteration $i-1$ but positive at iteration i, and similarly, $C_{i-1} \setminus C_i$ is the set of points that are predicted as positive at iteration $i-1$ but negative at

iteration i. Since all points in the data space are mapped to a hypersphere, we use the surface area as the metric.

Closed-form expressions for the surface area of a hyper-spherical cap and that of a hypersphere have been well-studied long ago. Recently, [59] has derived the surface area of the intersection of two hyper-spherical caps given $\boldsymbol{\omega}_{i-1}$ and b_{i-1} and $\boldsymbol{\omega}_i$, b_i. It is straightforward to see that $A(\mathcal{D}_i) = A(C_{i-1}) + A(C_i) - 2A(C_{i-1} \cap C_i)$. Therefore, in order to compute the model change rate in the feature space, the remaining problem is to obtain $\boldsymbol{\omega}$ and b.

As mentioned earlier in Section 4.1.2, SVM is usually solved in the dual form rather than the primal form. A typical SVM solver returns α's and b as an SVM model, but Equation (4.4) allows us to compute $\boldsymbol{\omega}$ from α_i's and $\phi(\boldsymbol{x}_i)$'s. Although $\phi(\boldsymbol{x})$ is intensionally bypassed by the kernel trick, we can apply *Cholesky Decomposition* to the kernel matrix, which is symmetric and strictly positive-definite for Gaussian kernel, and get a unique decomposition, as formally described below.

$$
\begin{aligned}
\mathbf{K}_{i\times i} &= \begin{pmatrix} K(\boldsymbol{x}_1,\boldsymbol{x}_1) & \cdots & K(\boldsymbol{x}_1,\boldsymbol{x}_i) \\ \vdots & \ddots & \vdots \\ K(\boldsymbol{x}_i,\boldsymbol{x}_1) & \cdots & K(\boldsymbol{x}_i,\boldsymbol{x}_i) \end{pmatrix} \\
&= \begin{pmatrix} \phi(\boldsymbol{x}_1)^T\phi(\boldsymbol{x}_1) & \cdots & \phi(\boldsymbol{x}_1)^T\phi(\boldsymbol{x}_i) \\ \vdots & \ddots & \vdots \\ \phi(\boldsymbol{x}_i)^T\phi(\boldsymbol{x}_1) & \cdots & \phi(\boldsymbol{x}_i)^T\phi(\boldsymbol{x}_i) \end{pmatrix} \\
&= \begin{pmatrix} \phi(\boldsymbol{x}_1)^T \\ \vdots \\ \phi(\boldsymbol{x}_i)^T \end{pmatrix}_{i\times i} \times \Big(\phi(\boldsymbol{x}_1),\cdots,\phi(\boldsymbol{x}_i)\Big)_{i\times i}
\end{aligned}
$$

Note that the kernel matrix of the Gaussian kernel always has full rank for distinct examples, which means that the rank is increased by 1 every time a new example $\boldsymbol{x}_i$ is added and its projection and $\phi(\boldsymbol{x}_i)$ is independent of all previous examples' projections

Algorithm 5 Model Change Rate in the Feature Space

`Input :` labeled examples $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_i$ and their coefficients $\alpha_1, \ldots, \alpha_i$[5], b_i, $\boldsymbol{\omega}_{i-1}$ and b_{i-1}.
`Output :` the model change rate in the feature space at iteration i

1: $\mathbf{K}_{i\times i} \leftarrow$`updateKernelMatrix(`$\mathbf{K}_{(i-1)\times(i-1)}, \boldsymbol{x}_i$`)`
2: $\{\phi(\boldsymbol{x}_1), \ldots, \phi(\boldsymbol{x}_i)\} \leftarrow$`CholeskyDecomposition(`$\mathbf{K}_{i\times i}$`)`
3: $\boldsymbol{\omega}_i \leftarrow \sum_{j=1}^{i} \alpha_j \phi(\boldsymbol{x}_j)$
4: $\boldsymbol{\omega}_{i-1} \leftarrow$`append(`$\boldsymbol{\omega}_{i-1}$`, 0)`
// computations below are in i-dimensional space:
5: *intersect* $\leftarrow$`getIntersectionArea(`$\boldsymbol{\omega}_{i-1}, b_{i-1}, \boldsymbol{\omega}_i, b_i$`)`
6: $area_{i-1} \leftarrow$`getUnitSphericalCapArea(`$i, \boldsymbol{\omega}_{i-1}, b_{i-1}$`)`
7: $area_i \leftarrow$`getUnitSphericalCapArea(`$i, \boldsymbol{\omega}_i, b_i$`)`
8: *change* $\leftarrow area_{i-1} + area_i - 2 \cdot intersect$
9: *whole* $\leftarrow$`getUnitSphereArea(`i`)`
10: **return** *change*/*whole*

$\phi(\boldsymbol{x}_1), \ldots, \phi(\boldsymbol{x}_{i-1})$ in the feature space. In other words, each example adds a new dimension to the span[4] of example's projections. When computing the surface area difference of two consecutive models $\boldsymbol{\omega}_{i-1}$ and $\boldsymbol{\omega}_i$, we add a zero to $\boldsymbol{\omega}_{i-1}$ to make it of the same dimensionality with $\boldsymbol{\omega}_i$.

We summarize the procedure of computing the model change rate in the feature space at iteration i in Algorithm 5. After we obtain the model change rate, we can just check convergence using the long-established methods.

4.3 Optimizations

We further provide a suite of optimizations for uncertainty sampling, including those for reducing the time cost of retrieving the most uncertain sample from the database, and for reducing the time of running the final query over the database.

[4]It is well-known that the feature space of Gaussian kernel is infinite dimensional but the projections of finite examples span a finite dimensional subspace of the infinite dimensional space.

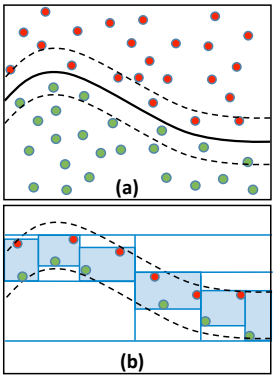


Figure 4.5: Decision tree based approximation of the non-linear decision boundary in the data space.

4.3.1 Sample Retrieval

A key performance goal in our work is to limit the cost of each iteration, including retrieving the most uncertain sample to label next, within a few seconds. Recent research [16] proposed to choose the sample closest to the current decision boundary of the SVM. However, finding the sample closest to the decision boundary from a large database is costly. Pre-computation to store the distance of each tuple to the decision boundary is not possible because the boundary changes in each iteration. As a result, existing implementations either pay the cost to scan the database, or sacrifice convergence by resorting to random sampling and among the random samples, choosing the one closest to the decision boundary.

Optimizing the retrieval of the most uncertain sample is challenging because the sample closest to the decision boundary is defined in the feature space, while sample retrieval is performed in the data space. There is no reverse mapping from the feature space to the data space. Below, we propose a decision tree based optimization for the sample retrieval problem without scanning the entire database.

Decision Tree Based Approximation. The classification boundary in the data space can be of arbitrary shapes, hence hard to be interpreted but can be approximated by the union of many disjunctive hyper-rectangles. As shown in Figure 4.5(a), the classification boundary is a solid black curve that cuts the data space into two regions: one filled with green positive

Algorithm 6 Decision Tree Based Approximation

1: Define a band that encloses the current SVM decision boundary.
2: Prepare a training dataset of synthetic grid points such that the points in the band are positive and otherwise are negative.
3: Feed the dataset to a decision-tree algorithm and train a decision tree.
4: Translate the leaves returned by the decision tree into a SQL query and send it to the backend database.
5: From all the results of the query, return the one that is closest to the SVM decision boundary as the sample to be used in the next iteration.

points and one with red negative points. The black curve can be approximated by the blue boxes in Figure 4.5(b). Each hyper-rectangle is a conjunction of conditions on (a subset of) dimensions in the data space, just like the leaves returned by the decision tree learning algorithm. This inspires us to leverage decision trees to approximate the classification boundary in the data space.

We summarize the main steps in Algorithm 6. Let us define the decision boundary (function) in the feature space as:

$$y(\boldsymbol{x}) = \boldsymbol{\omega}^T \phi(\boldsymbol{x}) + b = 0 \tag{4.6}$$

where $\boldsymbol{x}$ as a point in the data space, ϕ as the mapping function from the data space to the feature space, and $\boldsymbol{\omega}$ is the weight vector.

In step 1, we define a band in the data space as $\{\boldsymbol{x} |\ y(\boldsymbol{x}) \in [-\delta, \delta]\}$. In step 2, we enumerate synthetic grid points in the data space to find those residing in the band (positive points) and outside the band (negative points). In Step 3, we feed these points to build a decision tree on the data space dimensions. We then turn the decision tree to a database query (step 4) and among its output, find the exmple closest to the SVM decision boundary (step 5).

4.3.2 Final Result Retrieval

Once the exploration terminates upon convergence or by the user request, we obtain an SVM model as represented in Equation 4.6. The ultimate goal is to find all tuples in the

database D with positive predictions by this model, i.e., $\boldsymbol{x}$ such that $\boldsymbol{x} \in D$ and $y(\boldsymbol{x}) > 0$. To expedite the retrieval of the final results, we propose to build R-tree as the index over the database, and perform a top-down search in a depth-first fashion.

Branch and Bound. The unique aspect of the R-tree search is a solver-based branch and bound approach. Each R-tree node offers a hyper-rectangle as a minimum bounding box of all the data points reachable from this node. With an explicit decision function, $y(\boldsymbol{x})$ in Equation 4.6, we can compute an upper bound of $y(\boldsymbol{x})$ without visiting the descendent nodes. Instead, we can obtain it by solving the following constraint optimization problem.

$$\begin{aligned} \max_{\boldsymbol{x}} \quad & y(\boldsymbol{x}) \\ \text{s.t.} \quad & a_j \leq \boldsymbol{x}^{(j)} \leq b_j,\ j = 1, \ldots, d. \end{aligned} \tag{4.7}$$

where $[a_j, b_j]$ is the range of the tree node on the j-th dimension and $\boldsymbol{x}^{(j)}$ is the value of $\boldsymbol{x}$ on the j-th dimension. If the upper bound is already smaller than 0, we can prune the entire subtree rooted at this node. Since the positive results tend to be clustered and mark only a small portion of the database in practice, with such index structure, we may gain a significant improvement over scanning the entire database and running the model on each tuple.

4.4 Experimental Evaluation

We have implemented all of our proposed techniques in a Java-based prototype for data exploration, which connects to a PostgreSQL database. In this section, we evaluate our techniques in terms of accuracy (using the F1-score), convergence rate (the number of user labeled samples needed to reach an accuracy level), and efficiency (the execution time in each iteration and in final query retrieval). We also compare our system to two state-of-the-art systems on explore-by-example, AIDE [34, 35] and LifeJoin [26], as well as specific sampling methods from active learning [16].

Attributes	Query template	Selectivity
(*rowc*,*colc*)	**Q1.1:** $rowc > 662.5$ and $rowc < 702.5$ and $colc > 991.5$ and $colc < 1053.5$	0.1%
	Q1.2: $rowc > 617.5$ and $rowc < 747.5$ and $colc > 925$ and $colc < 1120$	0.97%
	Q1.3: $rowc > 480$ and $rowc < 885$ and $colc > 719$ and $colc < 1326$	9.43%
	Q2.1: $(rowc - 682.5)^2 + (colc - 1022.5)^2 < 29^2$	0.1%
	Q2.2: $(rowc - 682.5)^2 + (colc - 1022.5)^2 < 90^2$	0.98%
	Q2.3: $(rowc - 682.5)^2 + (colc - 1022.5)^2 < 280^2$	9.43%
(*ra*,*dec*)	**Q3.1:** $ra > 190$ and $ra < 200$ and $dec > 53$ and $dec < 57$	0.1%
	Q3.2: $ra > 180$ and $ra < 210$ and $dec > 50$ and $dec < 60$	0.95%
	Q3.3: $ra > 150$ and $ra < 240$ and $dec > 40$ and $dec < 70$	10.24%
(*rowc*,*colc*) + (*ra*,*dec*)	4-dimensional queries as a combination of the above (*rowc*,*colc*) and (*ra*,*dec*) queries, e.g., **Q2.2 + Q3.2**	varied

Table 4.1: Query templates (selectivity is reported on 1% dataset with 1,918,287 tuples)

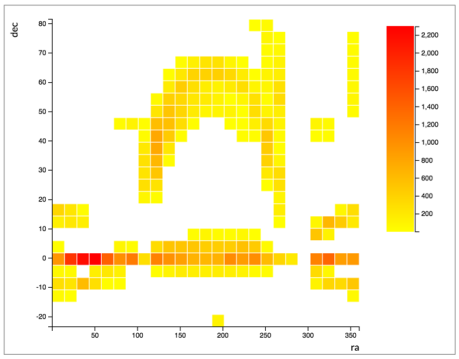


Figure 4.6: (*ra*, *dec*) distribution

Datasets: We evaluate our techniques using the "PhotoObjAll" table, which contains 510 attributes, from the Sloan Digital Sky Survey (SDSS) with data release 8[6]. The table contains the full photo metric catalog quantities for SDSS imaging, one entry per detection. We downloaded around 192 million tuples. For experiment purposes, we generated tables by random sampling the base table with different sampling ratios, 0.03%, 1%, 10%. After loading to PostgreSQL, the sizes were 300MB, 9991MB, and 98GB, respectively. B+ tree indexes on the key attribute *objid* were pre-built to facilitate example (i.e., tuple) retrieval. Since data exploration usually operates on a sampled dataset that fits in memory, we used the 1% dataset, which is the largest that fits in memory, as our default data exploration space.

[6]http://www.sdss3.org/dr8/

User Interest Queries: We extracted a set of queries from the SDSS query release 8 to represent true user interests[7], as shown in Table 4.1. These user interest queries allow us to run simulation of user exploration sessions: we precompute the answer set of each query, then run a data exploration session as described in the previous sections; during each iteration, when the active learning algorithm presents a new sample to be labeled, we consult the query answer set to decide whether to give a positive or negative label.

When choosing queries in our experiments, we consider the following factors : (1) pattern: queries can be linear or non-linear, (2) varied query selectivities, and (3) varied query dimensionalities. The queries are summarized in Table 4.1: On ($rowc$, $colc$), i.e., the row and column center positions, data are roughly evenly distributed, and we have two groups of queries, one for the linear pattern (Q1) and one for the non-linear (Q2). On (ra, dec), i.e., the right-ascension and declination in the spherical coordinate system, represent workloads with skewed data (see Figure 4.6). Within each group, we consider three selectivities: 0.1%, 1% and 10%, which we believe covers the general settings in real applications. We combine queries on ($rowc$, $colc$) and (ra, dec) to obtain 4-dimensional queries with varied selectivities, e.g., combining Q2.2 on ($rowc$, $colc$) and Q3.3 on (ra, dec) will result in a query on ($rowc$, $colc$, ra, dec) with selectivity 0.1%.

We conduct 10 runs for each query and in each run, one positive sample and one negative sample that are randomly selected will be fed to the system as initial samples.

Servers: Our experiments were run on five identical servers, each with 12-cores, 2× Intel(R) Xeon(R) CPU X5650 @2.66GHz, 64GB memory, JVM 1.7.0 on CentOS 6.6.

4.4.1 Sample Retrieval Methods

We first use the general active learning framework (without making convex assumptions) and evaluate our sample retrieval methods, the decision tree and solver methods ([72]), for finding the sample closest to the SVM decision boundary in each iteration. For comparison,

[7]http://skyserver.sdss.org/dr8/en/help/docs/realquery.asp

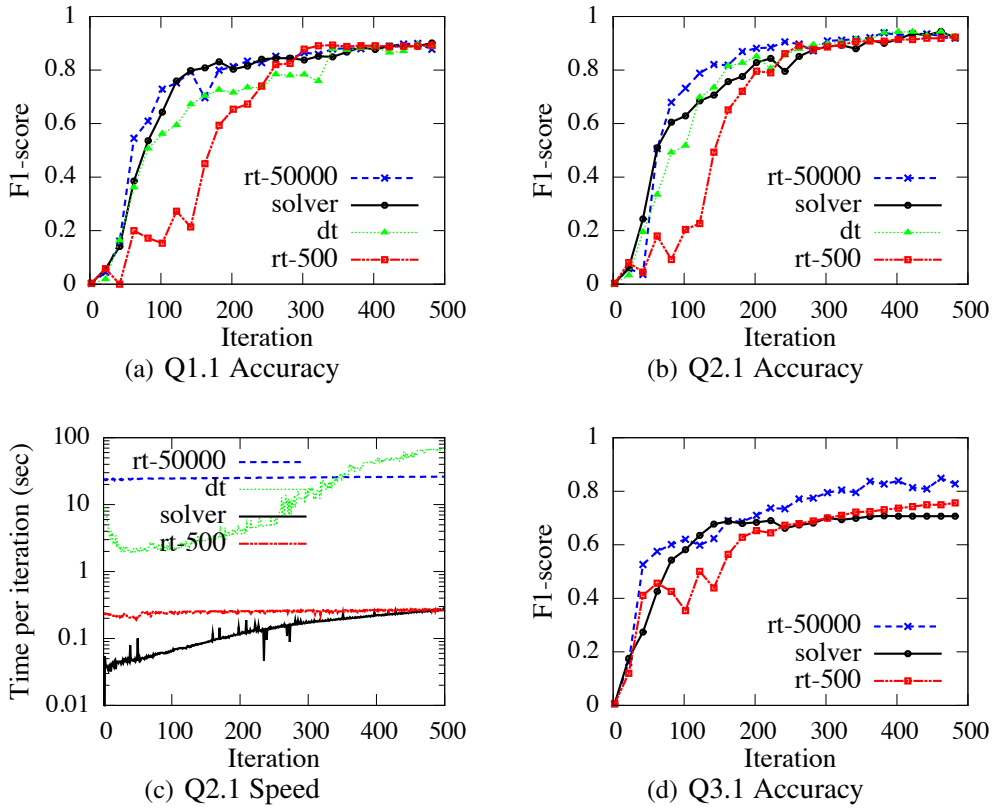
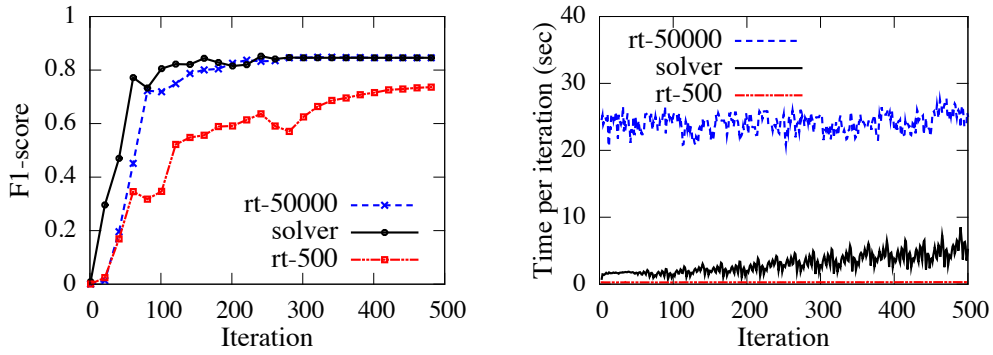


Figure 4.7: The accuracy and response time of various example acquisition methods on various workloads.

we also include the best sampling method reported for active learning [16]. This method retrieves a fixed number L of random samples at each iteration and among them chooses the one closest to the decision boundary, denoted as x^*. L is chosen under the condition that x^* is among the top p% closest instances in the original dataset with probability q. We varied L from 500 to 50,000, whose corresponding p% and q values are (1%, 0.993) and (0.01%, 0.993). We call these methods "random-top-500" to "random-top-50k". We made consistent observations that for queries with selectivity 1% and 10%, all techniques have marginal difference in terms of accuracy. Therefore, we show the results for queries with 0.1%, considered as harder queries, in Figure 4.7 and 4.8.

Expt 1 (*rowc*, *colc*): First consider Figure 4.7(b) and 4.7(c), both for Q2.1 with a nonlinear pattern in Table 4.1. We can see that random-top-500 is much less accurate



(a) Q2.2+Q3.2 Accuracy (b) Q2.2+Q3.2 Speed

Figure 4.8: The accuracy and response time of various example acquisition methods on various workloads.

compared to random-top-50000, especially in early iterations. This is because for highly-selective queries, it is hard to hit a good and informative sample among a set of only 500 samples. However, random-top-500 only takes 0.25 second per iteration while random-top-50k takes around 25 seconds due to much more samples retrieved from the database per iteration. The accuracy of the decision tree method and the solver method lies in between, closer to random-top-500k. The time per iteration is around 2 seconds for the solver method and does not increase with iterations; the average time of the decision tree method is 2.86 seconds over the first 200 iterations but increases fast after some point. **Overall, the solver method finds the best tradeoff between accuracy and response time**. The accuracy trends of various methods for Q1.1 with the linear pattern are very similar, as shown in Figure 4.7(a). We omit the time plot for the same reason.

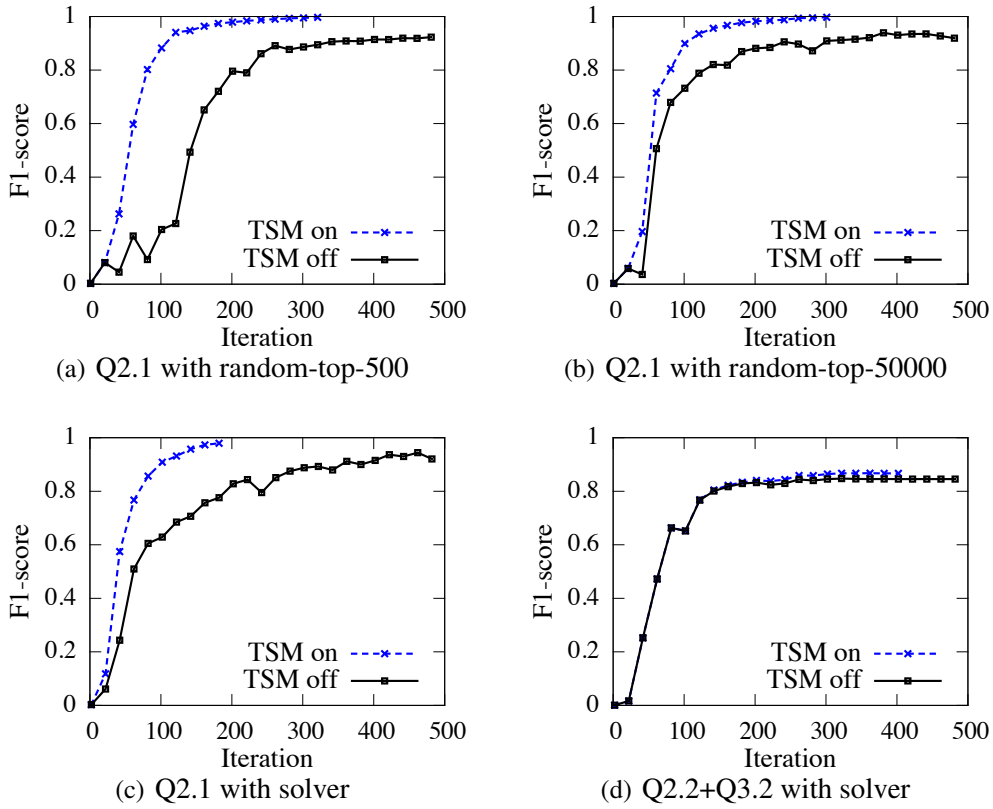
Expt 2 (*ra*, *dec*): Since the decision-tree-based method is always observed inferior to the solver method, we omit it in the following experiments. Figure 4.7(d) shows the accuracy result of Q3.1, which is on a skewed dataset. The solver method still approximates well random-top-50k, especially in the early iterations, while the response time stays slow, very similar to Figure 4.7(c).

Expt 3 (*rowc*, *colc*, *ra*, *dec*): We next combine the *rowc-colc* query of 1% selectivity with a *ra-dec* query of 1% selectivity. As Figure 4.8(a) and Figure 4.8(b) show, the solver method approximates random-top-50k for accuracy and random-top-500 for response time, hence achieving a good tradeoff between them.

4.4.2 Uncertainty Sampling for Convex Queries

We next consider the convex properties of queries. We compare our uncertainty sampling algorithm, Algorithm 3, to default active learning work, Algorithm 2, for any given example retrieval method. These algorithms are labeled as "TSM on" and "TSM off" in Figure 4.9 and 4.10. The x-axis is the iteration (conversation) number; at each iteration, the user provides a label for one example. The system is automatically terminated after 500 examples labeled either by the user or TSM.

Expt 4: Accuracy for Q2.1 with different example retrieval methods are shown in Figure 4.9(a)-4.9(c). At any iteration, TSM has a better accuracy than the baseline active learning algorithm; the F1-score of the latter does not even reach 0.99 with 500 iterations while TSM only requires around 200-300 iterations to obtain at least 0.99 F1-score. **Overall, with TSM turned on, the user is expected to label much fewer examples while achieving higher accuracy.** The reasons are: (1) TSM keeps track of regions (i.e., the positive and negative regions) where labels are certain based on the existing labeled examples, and only requires the user to label an example if it falls in the uncertain region. As exploration proceeds, TSM requires less user labeling because the uncertain region shrinks over time. (2) SVM itself can make wrong predictions for points in the positive and negative regions, while TSM will not (see Proposition 4.2.1 and 4.2.2), which is guaranteed to bring a better accuracy. Accuracy for a four-dimensional query combining Q2.2 and Q3.2 is shown in Figure 4.9(d). Compared with Q2.1, the increase in the dimensionality makes it harder to form effective certain regions for TSM, so TSM does not help to label any example retrieved by the solver method in early iterations and the F1-score is almost identical. But eventually



(a) Q2.1 with random-top-500 (b) Q2.1 with random-top-50000

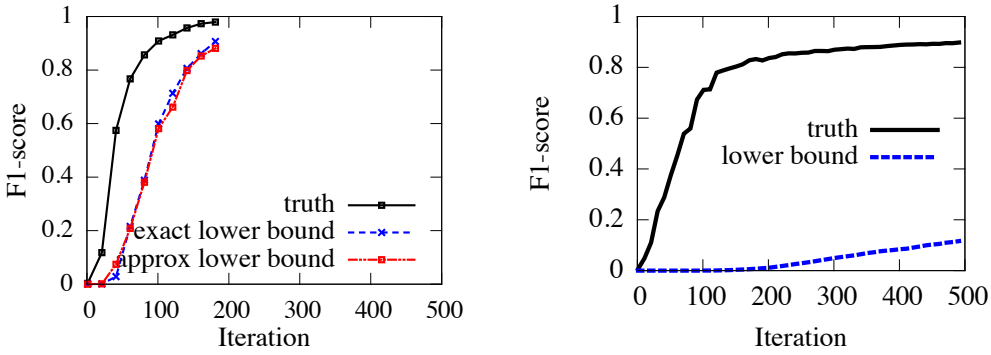
(c) Q2.1 with solver (d) Q2.2+Q3.2 with solver

Figure 4.9: Model accuracy with TSM turned on and off for various workload.

the user only needs to label around 400 examples, which is a 20% reduction on the user effort. It is worth noting that **our TSM technique works with any retrieval methods.**

Expt 5: We next study the effectiveness of the lower bound given by TSM. We compute both the exact lower bound and the approximate lower bound based on 16588[8] samples. The results on Q2.1 is shown in Figure 4.10(a). It can be seen that the blue and red lines are very close and both are indeed lower bounds for the black line. There is a gap between the truth and the lower bound but eventually the lower bound converge to the true F1-score: when the true F1-score is 0.99, the lower bound is around 0.96. The same observation is made in Q1.1. The results on Q2.2+Q3.2 is shown in Figure 4.10(b). The lower bound provided by TSM is not as tight as that for the 2-dimensional workload, as we discussed in the last

[8]Derived from Theorem 4.2.2 when $\lambda = 0.99$ and $\delta = 0.01$.



(a) Q2.1 with solver (b) Q2.2+Q3.2 with random-top-50000

Figure 4.10: The effectiveness of lower bound provided by TSM.

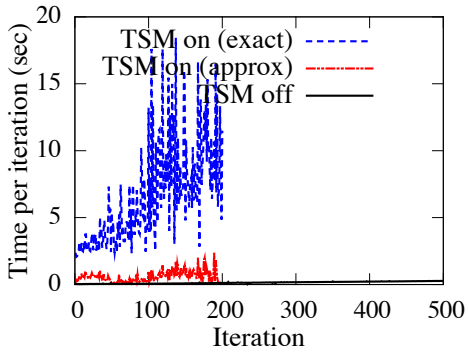


Figure 4.11: The time cost of turning on TSM.

experiment. For d-dimensional workloads, each polytope needs at least $d+1$ examples to form and each, so it requires a lot more labeled examples for larger d in order to shrink the uncertain region.

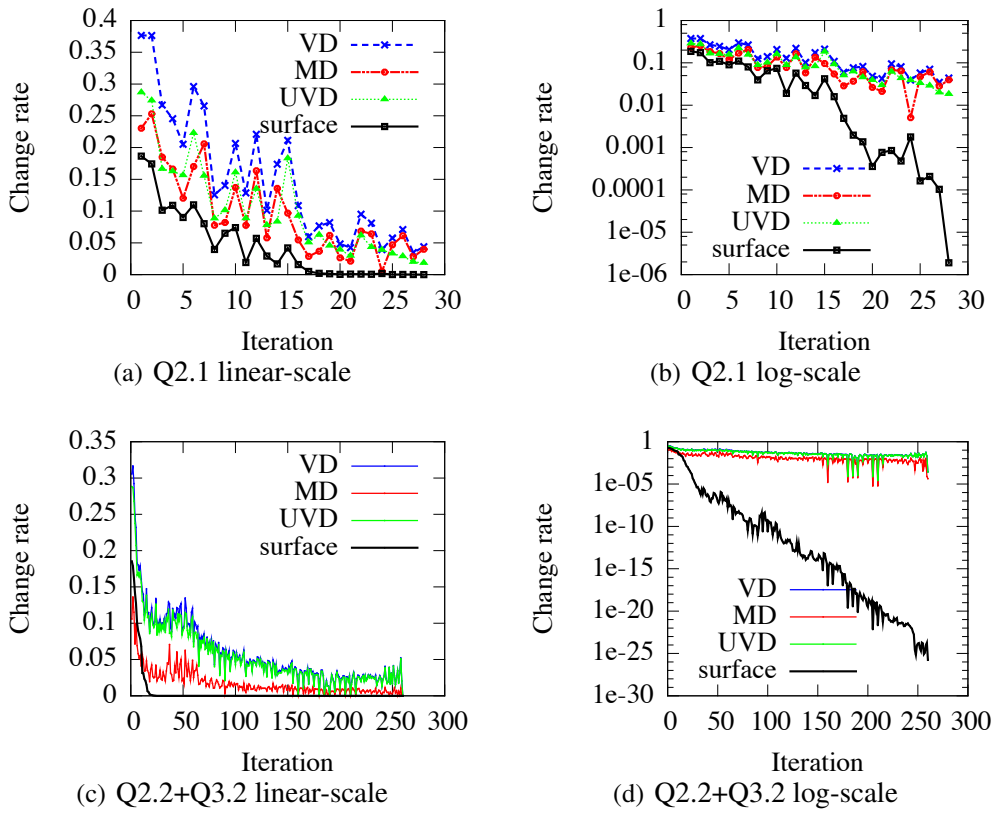
Expt 6: The response time per iteration for Q2.1 with TSM turned on and off is shown in Figure 4.11. With TSM on, we do see overheads. The overhead increases with iteration because as exploration proceeds, TSM will have better knowledge about the query so it can help the user to label more samples and feed the labeled samples to the learning module. We can also see that with the approximation the running time is much reduced without sacrificing the accuracy according to Figure 4.10(a).

4.4.3 Model Change Rate

Expt 7: In Section 4.2.2, we define model change rate based on the surface area in the feature space. In this experiment, we compare it with other choices on both two-dimensional workload (Q2.1) and four-dimensional workload (Q2.2+Q3.2). In Figure 4.12, we considered the following alternatives with ω_i and ω_{i+1} being the ω's in two consecutive iterations: $||\omega_{i+1} - \omega_i||/(||\omega_{i+1}|| + ||\omega_i||)$, $(||\omega_{i+1}|| - ||\omega_i||)/(||\omega_{i+1}|| + ||\omega_i||)$, and $||(\omega_{i+1}/||\omega_{i+1}|| - \omega_i/||\omega_i||)||/2$, denoted as VD (vector difference), MD (mode difference) and UVD (unit vector difference), respectively. Note that all four metrics are in [0, 1]. As we can see from the plot, our surface-area-based model change rate fluctuates less (Figure 4.12(a) and Figure 4.12(c)) than the other three. Less fluctuation is desirable when selecting a metrics for detecting model convergence. The threshold for surface-area-based metric should be set much lower than others as it decreases faster (Figure 4.12(b) and Figure 4.12(d)). With an appropriate threshold, model change rate defined on surface area is the best metrics among the four to detect model convergence.

4.4.4 Compare to Alternative Systems

We finally compare our system to two alternative systems for explore-by-example. (1) **AIDE** [34, 35] uses decision trees as the classification model. If a query pattern is non-linear, it uses a disjunction of conjunctive linear predicates (or a collection of hyper-rectangles in the data space) to approximate the pattern. We obtained the source code from the authors. (2) **LifeJoin** [26] reports a method, named "hybrid", as its best performing method. At each iteration, this method uses all the labeled samples to train a collection of weak-learners (error-free regarding the training data), extracts basic predicates from these learners, and train a **linear** SVM over these predicates. Then the linear SVM is used to seek the next sample for labeling, which is the one closest to the SVM boundary. The final retrieval method collects the support vectors of the final SVM, uses it as a training set to build a decision tree, and converts the positive class of the decision tree to a query to retrieve



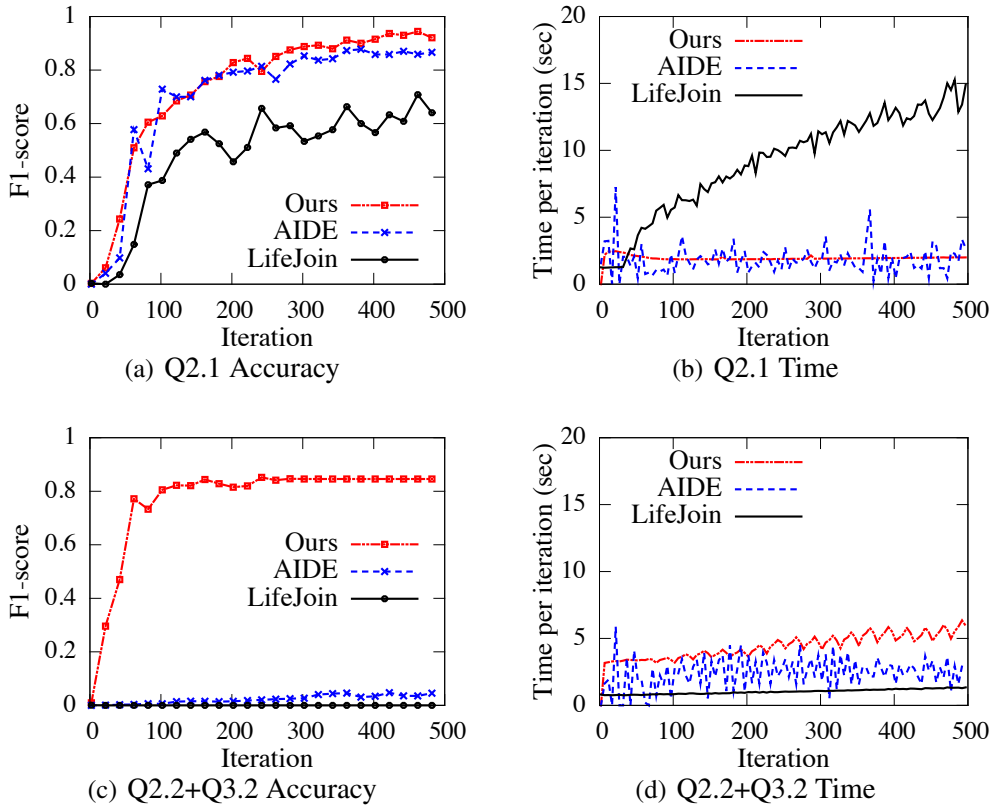
(a) Q2.1 linear-scale (b) Q2.1 log-scale

(c) Q2.2+Q3.2 linear-scale (d) Q2.2+Q3.2 log-scale

Figure 4.12: Model change rate defined on different metrics.

all the objects. We reimplemented the LifeJoin with two modifications: we used Random Forest (RF) with overfitted decision trees to build the weak learns as RF is a better known approach than a program synthesizer for this purpose, and we used our sample retrieval method to find the one closest to the SVM boundary, avoiding scanning the entire dataset. We tried to make parameters consistent with those recommended in the paper, including the number of weak learners used (10) and the number of basic features (on the order of hundreds). We run all experiments up to 500 user-labeled samples.

Figure 4.13 shows the results for both 2D and 4D workloads. The main observations are: (1) For the 2D nonlinear query (Q2.1), our system and AIDE are similar, while LifeJoin is significantly worse in both accuracy and per-iteration time. (2) For the 4D query that combines linear and nonlinear predicates (Q2.2+Q3.2), the accuracy of AIDE and LifeJoin



(a) Q2.1 Accuracy (b) Q2.1 Time

(c) Q2.2+Q3.2 Accuracy (d) Q2.2+Q3.2 Time

Figure 4.13: Compare our system to AIDE and LifeJoin in accuracy and per-iteration time for 2D and 4D query workloads.

drops to below 10%, while our system achieves 85% with the per-iteration time within a few seconds. (3) Our per iteration time is consistently around two seconds for Q2.1 and slowly increases to five seconds for Q2.2+Q3.2. In contrast, AIDE fluctuates over the course for both workloads; LifeJoin increases to 15 seconds quickly for Q2.1 and does not quite work for Q2.2+Q3.2 so per iteration is consistently below two seconds.

Finally, Table 4.2 shows the final result retrieval after 500 iterations and compares the three systems in both accuracy and running time. Again, LifeJoin suffers from low accuracy, using either its decision-tree based final retrieval method or running the SVM model over the database (in parentheses). AIDE loses accuracy for workloads beyond 2D. AIDE's model can be transformed into SQL queries, so the final result retrieval is generally very fast. Its accuracy on Q1.1 is the best as Q1.1 is a linear pattern query, which is not challenging for

Query	Metrics	LifeJoin	AIDE	Ours (B&B)
Q1.1	F-score (%)	7.13 (45.12)	95.8	88.0
	retrieval time (s)	0.683	0.013	79.7
Q2.1	F-score (%)	48.81 (58.76)	86.5	93.9
	retrieval time (s)	0.338	0.018	104.3
Q2.2+Q3.2	F-score (%)	0.02 (0.007)	4.6	84.9
	retrieval time (s)	2.575	0.088	207.2
Query	**Metrics**	**Ours (scan)**	**Ours (B&B, 20x)**	**Ours (scan, 20x)**
Q1.1	F-score (%)	88.0	88.06	88.1
	retrieval time (s)	1009.2	314.75	20184.9
Q2.1	F-score (%)	92.3	93.15	93.15
	retrieval time (s)	1023.8	340.73	20481.2
Q2.2+Q3.2	F-score (%)	84.6	81.34	81.34
	retrieval time (s)	1039.7	942.02	20794.1

Table 4.2: Compare to AIDE and LifeJoin for the final retrieval (after 500 iterations).

its decision-tree based sample retrieval. However, it loses to our system on non-linear 2D query and 4D query. Our system with branch-and-bound final result retrieval maintains high accuracy while having a modest final retrieval time of a few minutes. We also tested scalability on 20x data. As shown in the table, branch-and-bound retrieval scales sub-linearly due to the tree structure and early branch pruning while scanning scales linearly.

4.5 Related Work

Data Exploration. *Faceted search* iteratively recommends query attributes for drilling down into structured databases, but the user is often asked to provide attribute values until the desired tuple(s) are returned [76, 77, 55] or provides an"interestingness" measure and its threshold [32]. Semantic windows [54] are pre-defined multidimensional shape-based and content-based predicates that a user can explore. Its utility is restricted to the case that such patterns exactly suit the user interest. To speed up interactive exploration, adaptive tree indexes are built [108] on parts of the data that the user has actually queried, rather than on all the data upfront. The work [67] specifically focuses on iterative "linear algebra programs" and proposes techniques based on matrix factorization to make incremental

view maintenance substantially cheaper than re-evaluation. Most recent work has proposed a model to interpret the variability of likely queries in a workload [37], and dynamic prefetching of data tiles for interactive visualization [9].

Query by Example is a specific framework for data exploration. Earlier work on QBE focused on a visualization front-end that aims to minimize the user effort to learn the SQL syntax [51, 64, 70, 107]. Recent work [66] proposes exemplar queries which treat a *query* as a sample from the desired result set and retrieve other tuples based on similarity metrics, but for graph data only. The work [85] considers data warehouses with complex schemas and aims to learn the minimal project-join queries from a few example tuples efficiently. It does not consider selection with complex predicates, which is the main focus of our work. The work [53] helps users construct join queries for exploring relational databases, and [60] does so by asking the user to determine whether a given output table is the result of her intended query on a given input database. These works are relevant yet orthogonal to our active learning based approach.

Query formulation has been surveyed in [25]. The closest to our work is LifeJoin [26], which we described and compared in our performance study. Query By Output (QBO) [95] takes the output of some query on a database, and aims to construct an alternative query such that running these two queries on the database are instance-equivalent. Das Sarma et al. [31] studied the complexity of finding a query that describes the relationships between a database and an existing view. Dataplay [2] provides a graphical interface for users to directly construct and manipulate query trees, assuming that users already have knowledge about quantified queries. In [104], the user is asked to provide both the input and output relations, with a small number of sample tuples in each, in order to synthesize a SQL query consistent with the sample tuples. In summary, our work takes an active-learning approach with new sampling algorithms and theory on convergence.

Active Learning. [94, 83] provide a theoretical motivation on selecting which examples to request next using the notion of a version space.The main idea is to select instances

whose corresponding hyperplanes in parameter space split the current version space into two equal parts as much as possible hence achieve the fastest convergence. However, the convergence speed is unknown. Related to our work is a lower bound on the probability of misclassification error on the unlabeled training set based on a large deviation theory [19]. However, the calculation of the lower bound relies on user labeling of an additional sampled subset from the unlabeled pool, which is not required in our work. A recent set of papers [38, 47, 48, 49] offered probabilistic bounds for the classification error and sample complexity. Our work differs from these in that 1) we focus on F1-score, which suits selective user interest queries (imbalanced classes in classification), 2) our lower bound is deterministic. Note that different performance metrics (F measure versus classification error) lead to different relative performances of the active learning methods. Since the user interest exploration is naturally an imbalanced problem, i.e., the true user interest query selects way less than 50% database objects, F measure is a more suitable measure because it emphasizes the accuracy regarding the positive class (i.e., objects in the query answer set). Recent work [74] focuses on preference learning by pairwise comparison on structured entities. It uses linear SVM and the way to select the next example is similar to [94].

There are also a set of stopping criteria proposed for active learning. Schohn and Cohn [79] developed a heuristic stopping rule that labeling stops when the examples in the margin of the SVM have all been labeled. It does not give any indication of classification error at convergence. Four simple stopping criteria based on confidence estimation over the unlabeled data pool and the label consistency between consecutive training rounds of active learning have been presented [106]. Fu and Yang use as the stopping criterion a measure on whether SVM's separating hyperplane lies in a low density region [40]. Vlachos defines the confidence of the SVM classifier as the sum of the decision margins for the instances of a test set and stops the active learning process when the confidence reaches its peak [100]. Recent work in NLP [14] compares successive model predictions on a set of examples that do not need to be labeled and choose a proper set size as well as a cut-off value on

a measure of "agreement" . Another study [68] focuses on the committee-based active learning and proposes to stop active learning when the Selection Agreement (decision on the most informative example selected for the next iteration) is no less than the Validation Set Agreement (decision on the validation of the current classifier on an unannotated dataset). The above techniques are based on various heuristics, while our work provides stronger results, namely, provable lower bounds on the F1 accuracy measure, and uses the lower bound as the stopping criteria for active learning.

4.6 Conclusion

In this chapter, we presented the design of a new database service for data exploration by example. We devised new uncertainty sampling algorithms with formal results, and a series of optimizations to improve performance. Our main results are: (1) For convex patterns, our TSM algorithm is proved and experimentally tested to offer a lower-bound for F1-score, and reduces the user labeling effort. On 2-dimensional workloads, when TSM is applied, the user only needs to label 20% of the examples needed using traditional active learning without the TSM technique to achieve the same accuracy. On 4-dimensional workloads, the number is 80%. We also devise an approximation technique to reduce the overhead of applying TSM hence can provide interactive experience for users. Although TSM assumes convex query pattern and no-noise labeling and only works significantly well with low dimensionality, it is worth noting that TSM works with any sample retrieval method and provides a good foundation for follow-on work which has successfully relaxed the convex assumption and factorized a high-dimensional exploration problem into a collection of low-dimensional problems. (2) For general workloads, we define model change rate based on surface area difference in the feature space, which can be integrated into detecting model convergence. (3) We propose a series of optimizations, namely, decision-tree-based sample retrieval and branch-and-bound method for final result retrieval, to improve performance. (4) Our system

significantly outperforms AIDE and LifeJoin, two alternative systems, in accuracy while achieving desired efficiency for interactive exploration.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

5.1 Thesis Summary

Data management problems are crucial in large-scale scientific applications, where uncertainty presents in both data and user queries. In this thesis, I have proposed novel techniques to provide efficient query processing for the following three problems: data uncertainty in the relational model, data uncertainty in the array model, and data exploration by example under query uncertainty.

Data Uncertainty in Relational Databases. I proposed techniques to optimize probabilistic threshold query processing on continuous uncertain data by (*i*) expediting selections by reducing dimensionality of integration and using faster filters, (*ii*) expediting joins using new indexes, and (*iii*) using dynamic, per-tuple based planning that considers both cost and selectivity of operators. Results using the SDSS benchmark show significant performance gains over a state-of-the-art indexing technique and its threshold query optimizer.

Data Uncertainty in Array Databases. I proposed a number of storage and evaluation schemes for the *Subarray* operator, in particular, the *store-multiple* scheme, and building on that, the subarray-based join (SBJ) for the *Structure-Join* operator. The 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).

Data Exploration by Example. I proposed the design of a new database service for data exploration by examples. The main results include: (1) the three-set metric for convex queries can reduce the user labeling effort and lower-bound the F1-score hence can be used as an accuracy-based system stopping criterion, (2) the change rate metric can be generally applied to detect model convergence over iterations and is therefore another practical system stopping criterion, (3) the solver-based approximation expedites the example acquisition while still providing a high-quality example for labeling in each iteration, (4) the efficiency of the final result retrieval is much improved by applying indexes, and (5) the new system significantly outperforms AIDE and LifeJoin, two alternative systems, in accuracy while achieving desired efficiency for interactive exploration.

5.2 Future Work

There are some other exciting directions related to the topics of this thesis for further study.

Noisy User Feedback. Over the course of data exploration, a user may provide inaccurate feedback. The current TSM approach permanently converts part of the uncertain region in the data space to a positive region or a negative region based on each single feedback. In order to yield an accurate model, a detector of inaccurate region conversion could be added. A possible direction is to choose data points in previously converted regions, and ask additional user feedback of the data points. The additional feedback can serve for two purposes: (1) Detect an inaccurate user feedback by conflicts in user interest, and (2) correct a user feedback with high confidence based on majority voting. An interesting problem to solve is how to choose minimum additional data points to effectively detect and correct inaccurate feedback. Besides inaccurate user feedback, changing in user interest also poses an issue. A similar detector could be used to detect interest changes of the previous feedback. However, how to take time as a factor in feedback correction to capture the most recent user interest remains an open problem.

Dimensionality Reduction. Data exploration starts with all possible relevant attributes. Such high dimensionality requires more user feedback to generate an accurate model. As more user feedback are collected in the interactive process, opportunities to identify and prune irrelevant attributes can be further explored in future work.

Semi-supervised Learning. The current data exploration approach adopts supervised learning over user feedback. When user interest aligns with the clustering properties of data, adopting semi-supervised learning techniques can utilize unlabeled data to learn user interest more quickly.

Combining Data Uncertainty and Query Uncertainty. In the presence of both data uncertainty and query uncertainty, some high-level questions remain open to answer and I will discuss in two cases: (1) When user interest is the same across all possible worlds, some existing work on training a classifier on uncertain data [12, 102, 52] applies. How do we integrate data uncertainty with active learning and our TSM technique? (2) When user interest varies with different possible worlds, it is a problem of higher complexity. Those are challenging problems to solve in future work.

BIBLIOGRAPHY

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

[2] Abouzied, Azza, Hellerstein, Joseph M., and Silberschatz, Avi. Playful query specification with dataplay. *Proc. VLDB Endow. 5*, 12 (Aug. 2012), 1938–1941.

[3] Agarwal, Pankaj K., Aronov, Boris, Har-Peled, Sariel, Phillips, Jeff M., Yi, Ke, and Zhang, Wuzhou. Nearest-neighbor searching under uncertainty ii. In *PODS* (2013).

[4] Agarwal, Pankaj K., Efrat, Alon, Sankararaman, Swaminathan, and Zhang, Wuzhou. Nearest-neighbor searching under uncertainty. In *PODS* (2012).

[5] Antova, Lyublena, Jansen, Thomas, Koch, Christoph, and Olteanu, Dan. Fast and simple relational processing of uncertain data. In *ICDE* (2008), pp. 983–992.

[6] Arge, Lars, Samoladas, Vasilis, and Yi, Ke. Optimal external memory planar point enclosure. *Algorithmica 54*, 3 (May 2009), 337–352.

[7] Avnur, Ron, and Hellerstein, Joseph M. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA* (2000), Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, Eds., ACM, pp. 261–272.

[8] Barber, C. Bradford, Dobkin, David P., and Huhdanpaa, Hannu. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw. 22*, 4 (Dec. 1996), 469–483.

[9] Battle, Leilani, Chang, Remco, and Stonebraker, Michael. Dynamic prefetching of data tiles for interactive visualization. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, ACM, pp. 1363–1375.

[10] Beckmann, Norbert, Kriegel, Hans-Peter, Schneider, Ralf, and Seeger, Bernhard. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990* (1990), Hector Garcia-Molina and H. V. Jagadish, Eds., ACM Press, pp. 322–331.

[11] Benjelloun, Omar, Sarma, Anish Das, Halevy, Alon Y., and Widom, Jennifer. Uldbs: Databases with uncertainty and lineage. In *VLDB* (2006), pp. 953–964.

[12] Bi, Jinbo, and Zhang, Tong. Support vector classification with input data uncertainty. In *Advances in Neural Information Processing Systems 17*, L. K. Saul, Y. Weiss, and L. Bottou, Eds. MIT Press, 2005, pp. 161–168.

[13] Bizarro, Pedro, Babu, Shivnath, DeWitt, David, and Widom, Jennifer. Content-based routing: different plans for different data. In *Proceedings of the 31st international conference on Very large data bases* (2005), VLDB '05, VLDB Endowment, pp. 757–768.

[14] Bloodgood, Michael, and Vijay-Shanker, K. A method for stopping active learning based on stabilizing predictions and the need for user-adjustable stopping. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning* (Stroudsburg, PA, USA, 2009), CoNLL '09, Association for Computational Linguistics, pp. 39–47.

[15] Bohm, C., Pryakhin, A., and Schubert, M. The gauss-tree: Efficient object identification in databases of probabilistic feature vectors. In *Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on* (April 2006), pp. 9–9.

[16] Bordes, Antoine, Ertekin, Seyda, Weston, Jason, and Bottou, Léon. Fast kernel classifiers with online and active learning. *J. Mach. Learn. Res. 6* (Dec. 2005), 1579–1619.

[17] Boyd, Stephen, and Vandenberghe, Lieven. *Convex Optimization*. Cambridge University Press, 2004.

[18] Brown, Paul G. Overview of SciDB: large scale array storage, processing and analysis. In *SIGMOD Conference* (2010), pp. 963–968.

[19] Campbell, Colin, Cristianini, Nello, and Smola, Alex J. Query learning with large margin classifiers. In *Proceedings of the Seventeenth International Conference on Machine Learning* (San Francisco, CA, USA, 2000), ICML '00, Morgan Kaufmann Publishers Inc., pp. 111–118.

[20] Chaudhuri, Surajit, and Shim, Kyuseok. Optimization of queries with user-defined predicates. In *ACM Transactions on Database Systems* (1997), pp. 87–98.

[21] Cheng, Reynold, Kalashnikov, Dmitri V., and Prabhakar, Sunil. Evaluating probabilistic queries over imprecise data. In *SIGMOD Conference* (2003), pp. 551–562.

[22] Cheng, Reynold, Singh, Sarvjeet, and Prabhakar, Sunil. U-dbms: a database system for managing constantly-evolving data. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases* (2005), VLDB Endowment, pp. 1271–1274.

[23] Cheng, Reynold, Singh, Sarvjeet, Prabhakar, Sunil, Shah, Rahul, Vitter, Jeffrey Scott, and Xia, Yuni. Efficient join processing over uncertain data. In *CIKM* (2006), pp. 738–747.

[24] Cheng, Reynold, Xia, Yuni, Prabhakar, Sunil, Shah, Rahul, and Vitter, Jeffrey Scott. Efficient indexing methods for probabilistic threshold queries over uncertain data. In *VLDB* (2004), pp. 876–887.

[25] Cheung, Alvin, and Solar-Lezama, Armando. Computer-assisted query formulation. *Found. Trends Program. Lang. 3*, 1 (June 2016), 1–94.

[26] Cheung, Alvin, Solar-Lezama, Armando, and Madden, Samuel. Using program synthesis for social recommendations. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management* (New York, NY, USA, 2012), CIKM '12, ACM, pp. 1732–1736.

[27] Cramér, H. *Mathematical Methods of Statistics*. Princeton Mathematical Series. Princeton University Press, 1999.

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

[29] Cudré-Mauroux, Philippe, Kimura, Hideaki, Lim, Kian-Tat, Rogers, Jennie, Simakov, Roman, Soroush, Emad, Velikhov, Pavel, Wang, Daniel L., Balazinska, Magdalena, Becla, Jacek, DeWitt, David J., Heath, Bobbi, Maier, David, Madden, Samuel, Patel, Jignesh M., Stonebraker, Michael, and Zdonik, Stanley B. A demonstration of scidb: A science-oriented dbms. *PVLDB 2*, 2 (2009), 1534–1537.

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

[31] Das Sarma, Anish, Parameswaran, Aditya, Garcia-Molina, Hector, and Widom, Jennifer. Synthesizing view definitions from data. In *Proceedings of the 13th International Conference on Database Theory* (New York, NY, USA, 2010), ICDT '10, ACM, pp. 89–103.

[32] Dash, Debabrata, Rao, Jun, Megiddo, Nimrod, Ailamaki, Anastasia, and Lohman, Guy. Dynamic faceted search for discovery-driven analysis. In *CIKM* (2008).

[33] Diao, Yanlei, Dimitriadou, Kyriaki, Li, Zhan, Liu, Wenzhao, Papaemmanouil, Olga, Peng, Kemi, and Peng, Liping. AIDE: an automatic user navigation system for interactive data exploration. *PVLDB 8*, 12 (2015), 1964–1967.

[34] Dimitriadou, Kyriaki, Papaemmanouil, Olga, and Diao, Yanlei. Explore-by-example: an automatic query steering framework for interactive data exploration. In *SIGMOD Conference* (2014), pp. 517–528.

[35] Dimitriadou, Kyriaki, Papaemmanouil, Olga, and Diao, Yanlei. AIDE: an active learning-based approach for interactive data exploration. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* (2016). Accepted for publication.

[36] Duggan, Jennie, and Stonebraker, Michael. Incremental elasticity for array databases. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, ACM, pp. 409–420.

[37] Ebenstein, Roee, Kamat, Niranjan, and Nandi, Arnab. Fluxquery: An execution framework for highly interactive query workloads. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, ACM, pp. 1333–1345.

[38] El-Yaniv, Ran, and Wiener, Yair. Active learning via perfect selective classification. *J. Mach. Learn. Res. 13*, 1 (Feb. 2012), 255–279.

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

[40] Fu, Chunjiang, and Yang, Yupu. Low density separation as a stopping criterion for active learning svm. *Intelligent Data Analysis 19*, 4 (July 2015), 727–741.

[41] Ge, Tingjian. Join queries on uncertain data: Semantics and efficient processing. In *ICDE* (2011).

[42] Ge, Tingjian, Grabiner, David, and Zdonik, Stanley B. Monte carlo query processing of uncertain multidimensional array data. In *ICDE* (2011), pp. 936–947.

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

[44] Geoffrey, McLachlan, and David, Peel. *Finite Mixture Models*. Wiley-Interscience, 2000.

[45] Gray, Jim, Szalay, Alexander S., Thakar, Aniruddha R., Fekete, Gyorgy, O'Mullane, William, Nieto-Santisteban, María A., Heber, Gerd, and Rots, Arnold H. There goes the neighborhood: Relational algebra for spatial data search. *CoRR cs.DB/0408031* (2004).

[46] Grünbaum, Branko. Convex polytopes. In *Convex Polytopes*, 2 ed. Springer-Verlag New York, 2003.

[47] Hanneke, Steve. Rates of convergence in active learning. *Ann. Statist. 39*, 1 (02 2011), 333–361.

[48] Hanneke, Steve. Theory of disagreement-based active learning. *Found. Trends Mach. Learn. 7*, 2-3 (June 2014), 131–309.

[49] Hanneke, Steve. Refined error bounds for several learning algorithms. *J. Mach. Learn. Res. 17*, 1 (Jan. 2016), 4667–4721.

[50] Hellerstein, Joseph M., Koutsoupias, Elias, Miranker, Daniel P., Papadimitriou, Christos H., and Samoladas, Vasilis. On a model of indexability and its bounds for range queries. *J. ACM 49*, 1 (Jan. 2002), 35–55.

[51] Jacobs, B. E., and Walczak, C. A. A Generalized Query-by-Example Data Manipulation Language Based on Database Logic. *IEEE Transactions on Software Engineering 9*, 1 (1983), 40–57.

[52] Jebara, Tony, Kondor, Risi, and Howard, Andrew. Probability product kernels. *J. Mach. Learn. Res. 5* (Dec. 2004), 819–844.

[53] Kahng, Minsuk, Navathe, Shamkant B., Stasko, John T., and Chau, Duen Horng Polo. Interactive browsing and navigation in relational databases. *Proc. VLDB Endow. 9*, 12 (Aug. 2016), 1017–1028.

[54] Kalinin, Alexander, Cetintemel, Ugur, and Zdonik, Stan. Interactive data exploration using semantic windows. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, ACM, pp. 505–516.

[55] Kamat, Niranjan, Jayachandran, Prasanth, Tunga, Kathik, and Nandi, Arnab. Distributed Interactive Cube Exploration. In *ICDE* (2014).

[56] Kersten, M., Zhang, Y., Ivanova, M., and Nes, N. Sciql, a query language for science applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases* (New York, NY, USA, 2011), AD '11, ACM, pp. 1–12.

[57] Kimura, Hideaki, Madden, Samuel, and Zdonik, Stanley B. Upi: A primary index for uncertain databases. *PVLDB 3*, 1 (2010), 630–637.

[58] Kurose, James F., Lyons, Eric, McLaughlin, David, Pepyne, David, Philips, Brenda, Westbrook, David, and Zink, Michael. An end-user-responsive sensor network architecture for hazardous weather detection, prediction and response. In *AINTEC* (2006), pp. 1–15.

[59] Lee, Yongjae, and Kim, Woo Chang. Concise formulas for the surface area of the intersection of two hyperspherical caps. Tech. rep., KAIST technical report IE-TR-2014-01, 2014.

[60] Li, Hao, Chan, Chee-Yong, and Maier, David. Query from examples: An iterative, data-driven approach to query construction. *Proc. VLDB Endow. 8*, 13 (Sept. 2015), 2158–2169.

[61] Li, Jian, and Deshpande, Amol. Ranking continuous probabilistic datasets. *PVLDB 3*, 1 (2010), 638–649.

[62] Liu, Wenzhao, Diao, Yanlei, and Liu, Anna. An analysis of query-agnostic sampling for interactive data exploration. Tech. rep., University of Massachusetts Amherst, 2016. Technical report UM-CS-2016-003.

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

[64] McLeod, Dennis. The translation and compatibility of SEQUEL and Query by Example. In *Proceedings of the 2nd International Conference on Software Engineering* (1976), ICSE '76.

[65] Mercer, J. Functions of positive and negative type, and their connection with the theory of integral equations. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences 209*, 441-458 (1909), 415–446.

[66] Mottin, Davide, Lissandrini, Matteo, Velegrakis, Yannis, and Palpanas, Themis. Exemplar queries: Give me an example of what you need. *Proc. VLDB Endow. 7*, 5 (Jan. 2014), 365–376.

[67] Nikolic, Milos, ElSeidy, Mohammed, and Koch, Christoph. Linview: Incremental view maintenance for complex analytical queries. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, ACM, pp. 253–264.

[68] Olsson, Fredrik, and Tomanek, Katrin. An intrinsic stopping criterion for committee-based active learning. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning* (Stroudsburg, PA, USA, 2009), CoNLL '09, Association for Computational Linguistics, pp. 138–146.

[69] Olteanu, Dan, and Huang, Jiewen. Secondary-storage confidence computation for conjunctive queries with inequalities. SIGMOD '09, ACM, pp. 389–402.

[70] Özsoyoglu, Gültekin, and Wang, Huaqing. Example-Based Graphical Database Query Languages. *Computer 26*, 5 (1993), 25–38.

[71] Peng, Liping, Diao, Yanlei, and Liu, Anna. Optimizing probabilistic query processing on continuous uncertain data. *PVLDB 4* (2011).

[72] Peng, Liping, Huang, Enhui, Xing, Yuqing, Liu, Anna, and Diao, Yanlei. Uncertainty sampling and optimization for interactive database exploration. In *UMass Technical Report, http://www.cs.umass.edu/yanlei/explore2017.pdf* (2007).

[73] Qi, Yinian, Jain, Rohit, Singh, Sarvjeet, and Prabhakar, Sunil. Threshold query optimization for uncertain data. In *SIGMOD Conference* (2010), pp. 315–326.

[74] Qian, Li, Gao, Jinyang, and Jagadish, H. V. Learning user preferences by adaptive pairwise comparison. *Proc. VLDB Endow. 8*, 11 (July 2015), 1322–1333.

[75] Ravishanker, Nalini, and Dey, Dipak. *A first course in linear model theory*. CRC, 2002.

[76] Roy, Senjuti Basu, Wang, Haidong, Das, Gautam, Nambiar, Ullas, and Mohania, Mukesh. Minimum-effort driven dynamic faceted search in structured databases. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management (CIKM)* (2008).

[77] Roy, Senjuti Basu, Wang, Haidong, Nambiar, Ullas, Das, Gautam, and Mohania, Mukesh. Dynacet: Building dynamic faceted search systems over databases. In *International Conference on Data Engineering (ICDE)* (2009).

[78] Ruttenberg, Brian E., and Singh, Ambuj K. Indexing the earth movers distance using normal distributions. In *Proceedings of the VLDB Endowment, Vol. 5, No. 3* (2012), pp. 205–216.

[79] Schohn, Greg, and Cohn, David. Less is more: Active learning with support vector machines. In *Proceedings of the Seventeenth International Conference on Machine Learning* (San Francisco, CA, USA, 2000), ICML '00, Morgan Kaufmann Publishers Inc., pp. 839–846.

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

[81] Sellis, Timos K., Roussopoulos, Nick, and Faloutsos, Christos. The r+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1987), VLDB '87, Morgan Kaufmann Publishers Inc., pp. 507–518.

[82] Sen, Prithviraj, Deshpande, Amol, and Getoor, Lise. Exploiting shared correlations in probabilistic databases. In *VLDB* (2008).

[83] Settles, Burr. *Active Learning (Synthesis Lectures on Artificial Intelligence and Machine Learning)*, vol. 6. Morgan Claypool.

[84] Shawe-Taylor, John, and Cristianini, Nello. Section 2.2.3. kernel-defined nonlinear feature mappings. In *Kernel Methods for Pattern Analysis*. Cambridge University Press, New York, NY, USA, 2004.

[85] Shen, Yanyan, Chakrabarti, Kaushik, Chaudhuri, Surajit, Ding, Bolin, and Novik, Lev. Discovering queries based on example tuples. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, ACM, pp. 493–504.

[86] Singh, Sarvjeet, Mayfield, Chris, Shah, Rahul, Prabhakar, Sunil, Hambrusch, Susanne E., Neville, Jennifer, and Cheng, Reynold. Database support for probabilistic attributes and tuples. In *ICDE* (2008), pp. 1053–1061.

[87] Steinwart, Ingo, Hush, Don, and Scovel, Clint. An explicit description of the reproducing kernel hilbert spaces of gaussian rbf kernels. Tech. rep., IEEE Trans. Inform. Theory, 2005.

[88] Stonebraker, Michael, Bear, Chuck, Çetintemel, Ugur, Cherniack, Mitch, Ge, Tingjian, Hachem, Nabil, Harizopoulos, Stavros, Lifter, John, Rogers, Jennie, and Zdonik, Stanley B. One size fits all? part 2: Benchmarking studies. In *CIDR* (2007), pp. 173–184.

[89] Stonebraker, Michael, Becla, Jacek, DeWitt, David J., Lim, Kian-Tat, Maier, David, Ratzesberger, Oliver, and Zdonik, Stanley B. Requirements for science data bases and scidb. In *CIDR* (2009).

[90] Stonebraker, Michael, Brown, Paul, Poliakov, Alex, and Raman, Suchi. The architecture of scidb. In *SSDBM* (2011), pp. 1–16.

[91] Suciu, Dan, Connolly, Andrew, and Howe, Bill. Embracing uncertainty in large-scale computational astrophysics. In *Proceedings of the 3rd International Workshop on Management of Uncertain Data (MUD)* (2009).

[92] Szalay, Alexander S., Kunszt, Peter Z., Thakar, Ani, Gray, Jim, Slutz, Donald R., and Brunner, Robert J. Designing and mining multi-terabyte astronomy archives: The sloan digital sky survey. In *SIGMOD Conference* (2000), pp. 451–462.

[93] Tao, Yufei, Xiao, Xiaokui, and Cheng, Reynold. Range search on multidimensional uncertain data. *ACM Trans. Database Syst. 32*, 3 (Aug. 2007).

[94] Tong, Simon, and Koller, Daphne. Support vector machine active learning with applications to text classification. *J. Mach. Learn. Res. 2* (Mar. 2002), 45–66.

[95] Tran, Quoc Trung, Chan, Chee-Yong, and Parthasarathy, Srinivasan. Query by output. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2009), SIGMOD '09, ACM, pp. 535–548.

[96] Tran, Thanh T. L., McGregor, Andrew, Diao, Yanlei, Peng, Liping, and Liu, Anna. Conditioning and aggregating uncertain data streams: Going beyond expectations. *PVLDB 3*, 1 (2010), 1302–1313.

[97] Tran, Thanh T. L., Peng, Liping, Diao, Yanlei, McGregor, Andrew, and Liu, Anna. Claro: modeling and processing uncertain data streams. *VLDB J. 21*, 5 (2012), 651–676.

[98] Tran, Thanh T. L., Peng, Liping, Li, Boduo, Diao, Yanlei, and Liu, Anna. Pods: a new model and processing algorithms for uncertain data streams. In *SIGMOD Conference* (2010), pp. 159–170.

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

[100] Vlachos, Andreas. A stopping criterion for active learning. *Comput. Speech Lang. 22*, 3 (July 2008), 295–312.

[101] Wang, Daisy Zhe, Michelakis, Eirinaios, Garofalakis, Minos, and Hellerstein, Joseph. Bayesstore: Managing large, uncertain data repositories with probabilistic graphical models. In *VLDB* (2008).

[102] Wang, Ximing, and Pardalos, Panos M. A survey of support vector machines with uncertainties. *Annals of Data Science 1*, 3 (Dec 2014), 293–309.

[103] Zäschke, Tilmann, Zimmerli, Christoph, and Norrie, Moira C. The ph-tree: A space-efficient storage structure and multi-dimensional index. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, ACM, pp. 397–408.

[104] Zhang, S., and Sun, Y. Automatically synthesizing sql queries from input-output examples. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Nov 2013), pp. 224–234.

[105] Zhang, Ying, Kersten, Martin, and Manegold, Stefan. Sciql: Array data processing inside an rdbms. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, ACM, pp. 1049–1052.

[106] Zhu, Jingbo, Wang, Huizhen, Hovy, Eduard, and Ma, Matthew. Confidence-based stopping criteria for active learning for data annotation. *ACM Trans. Speech Lang. Process. 6*, 3 (Apr. 2010), 3:1–3:24.

[107] Zloof, Moshé M. Query-by-example: operations on hierarchical data bases. In *Proceedings of the June 7-10, 1976, national computer conference and exposition* (1976), AFIPS '76, pp. 845–853.

[108] Zoumpatianos, Kostas, Idreos, Stratos, and Palpanas, Themis. Indexing for interactive exploration of big data series. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, ACM, pp. 1555–1566.