EFFICIENT SUMMARIZATION TECHNIQUES FOR MASSIVE DATA

by

Jeffrey Jestes

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computing

School of Computing

The University of Utah

December 2013

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Jeffrey Jestes**

has been approved by the following supervisory committee members:

Feifei Li	, Chair	**7/30/2013** Date Approved
Ke Yi	, Member	**7/30/2013** Date Approved
Graham Cormode	, Member	**7/30/2013** Date Approved
Suresh Venkatasubramanian	, Member	**7/30/2013** Date Approved
Jeff M. Phillips	, Member	**7/30/2013** Date Approved

and by **Alan Davis**, Chair/Dean of

the Department/College/School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

We are living in an age where data are being generated faster than anyone has previously imagined across a broad application domain, including customer studies, social media, sensor networks, and the sciences, among many others. In some cases, data are generated in massive quantities as terabytes or petabytes. There have been numerous emerging challenges when dealing with massive data, including: (1) the explosion in *size* of data; (2) data have increasingly more *complex structures and rich semantics*, such as representing temporal data as a piecewise linear representation; (3) *uncertain data* are becoming a common occurrence for numerous applications, e.g., scientific measurements or observations such as meteorological measurements; (4) and data are becoming increasingly *distributed*, e.g., distributed data collected and integrated from distributed locations as well as data stored in a distributed file system within a cluster.

Due to the massive nature of modern data, it is oftentimes infeasible for computers to efficiently manage and query them exactly. An attractive alternative is to use data summarization techniques to construct data summaries, where even efficiently constructing data summaries is a challenging task given the enormous size of data. The data summaries we focus on in this thesis include the *histogram* and *ranking operator*. Both data summaries enable us to summarize a massive dataset to a more succinct representation which can then be used to make queries orders of magnitude more efficient while still allowing approximation guarantees on query answers. Our study has focused on the critical task of designing efficient algorithms to summarize, query, and manage massive data.

I dedicate this thesis to my family, who supported me each step of the way.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This thesis would not have been possible without my advisor, Feifei Li. I first met Feifei by attending his undergraduate database course at Florida State University. Feifei taught in a way which made me excited to learn more about database systems and data management in general. He was always happy to meet with me both inside and outside of office hours to discuss further details about topics discussed in class. He also exposed me to his research and showed me how fun, exciting, and fulfilling data management research can be. Without his guidance and recommendation to pursue a PhD in data management research, I would have never embarked on the long journey which is now coming to a close. The past five years have been extremely rewarding, and I would like to thank Feifei for opening my eyes to the world of research and for being my advisor who always gave me advice on topics of research and which directions to take when I was stuck or lost along the way.

I am honoured to have worked with some amazing researchers and would like to thank Feifei Li, Ke Yi, Graham Cormode, and Jeff M. Phillips for all of their insights, efforts, and advice during our collaborations. Each collaboration was a fun and exciting experience for me and would not have been possible without everyone's efforts and expertise. My collaborators have also taught me a great deal and I feel have enriched my skills and expertise as a researcher, as everyone challenged me to think in new ways to solve the problems on which we worked.

I thank Suresh Venkatasubramanian for his insightful comments on my future as a researcher as well as for his constructive comments and challenges on my talks in the data group, which I feel have made me a better presenter.

Finally, I thank the GAANN program for funding my first two years of research and the NSF for funding me during the final years of my research.

CHAPTER 1

INTRODUCTION

We are living in an age where data are being generated faster than anyone has previously imagined across a broad application domain, as indicated by Figure 1.1. Given the recent explosion of data we have witnessed within the past few years, it has become apparent that many algorithms designed for these applications simply will not scale to the huge amounts of data being generated. As we witness this explosion in data, there are numerous challenges which are continuously arising and changing the face of data management research. Some of the challenges we have observed and studied in this thesis include:

1) Coping with the ever-increasing *size* of data, for instance, which makes even simple queries, such as range queries, extremely costly and intractable.

2) Dealing with the emergence of increasingly *complex structures and rich semantics* of data; for instance, the MesoWest Project [5] represents various sensor measurements,

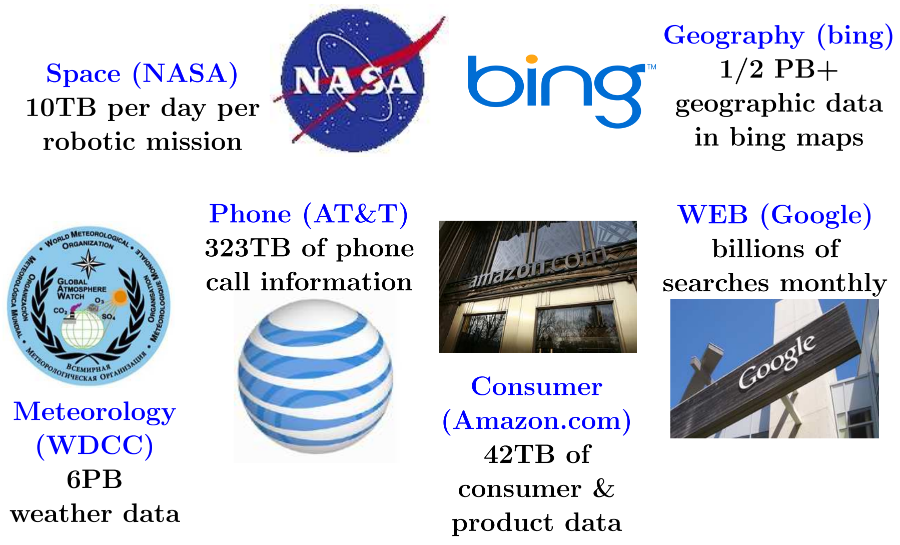

Space (NASA)
10TB per day per robotic mission
Geography (bing)
1/2 PB+ geographic data in bing maps
Phone (AT&T)
323TB of phone call information
WEB (Google)
billions of searches monthly
Meteorology (WDCC)
6PB weather data
Consumer (Amazon.com)
42TB of consumer & product data

Figure 1.1. Massive data spanning public and private sectors [1–4].

such as temperature, it has been continuously recording from weather towers across the United States since 1997 in a piecewise linear representation; there are clearly *rich semantics* one needs to capture with respect to the temporal aspect of such data.

3) Managing *uncertain data*, e.g., uncertainties arising in scientific measurements, such as temperature measurements collected by MesoWest, or observations such as the analysis of DNA and proteins.

4) Handling *distributed data*, such as distributed data collected and integrated in a sensor network as well as data stored in a distributed file system within a cluster.

With massive data, it is usually infeasible for computers to efficiently manage and query it exactly. An attractive alternative is to use data summaries. There are many possibilities when discussing data summaries, but for this thesis we focus on two types: (1) the *histogram*, which can be used to gain quick insights on massive data stored in relational databases or in a distributed file system within a cluster, (2) and *ranking operators*, where rankings are used to summarize a massive dataset to a more succinct dataset containing only the top-k most important records. An overview of the challenges faced with large data as well as the data summaries used to address these challenges in this thesis is illustrated in Figure 1.2. Our proposed summaries can make queries and data analytics tasks orders of magnitude more efficient while still allowing approximation guarantees on answers. Our PhD study has focused on the critical task of designing efficient algorithms, exploiting parallel and distributed settings when possible, to summarize and query massive data.

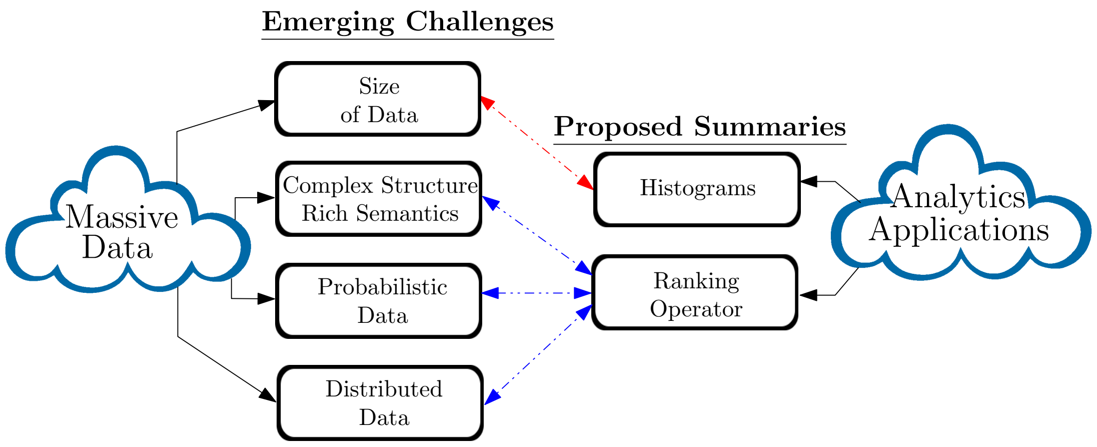


Figure 1.2. Massive data challenges and proposed data summaries.

1.1 Main Results of Our Research

Querying a massive dataset can require extensive amounts of resources, including I/Os, computation, and in some instances communication, and it can be unrealistic to issue queries over the scale of data we are seeing nowadays, especially when there are restrictions to resource utilization as well as *complex structures and rich semantics* in the data. The common scenario we are seeing is that data are first broken up into chunks and then stored in some form of distributed file system within a cluster, such as the Google File System [6] or Hadoop Distributed File System [7]. Then, data analytics, e.g., database queries, used by end-user applications are created as jobs and sent to the cluster for scheduling. Eventually, the cluster executes these data analytics tasks over the entire massive dataset. The primary problem with this approach is that it is extremely wasteful of resources within a cluster, I/Os, CPU time, communication, and can also generate a lot of unnecessary heat, which has become one of the biggest banes for data centers. Not to mention, queries may take an excessive amount of time before returning any results to users.

Users are often willing, or in some instances required, to trade some of the accuracy obtained in an exact solution for an approximate solution in order to save orders of magnitude in computation, I/Os, and communication [8–10]. One particular and useful way of obtaining approximate solutions is to first construct a data summary of the huge data in the cluster, as seen in Figure 1.3(a), which provides quality guarantees for a particular set of queries. A nice feature of data summaries is they typically are *independent* of the dataset size and can be defined to depend only on the desired *error* in the query, resulting in data summaries which are often only kilo- or megabytes in size (as we will see in Chapters 2-5). Building a data summary is usually a one-time cost, and serves as a surrogate (as seen in Figure 1.3(b)) for the original dataset residing in the cluster, allowing queries to be answered much faster and at the same time potentially reducing heat and resource utilization.

1.2 Summaries for Massive Data

As discussed above, and summarized in Figure 1.2, we identify four challenges emerging from massive data: (1) *size*; (2) *complex structure and rich semantics*; (3) *uncertain data*; (4) and *distributed data*. We propose the use of the *histogram* and *ranking operator* to deal with these emerging challenges and give a brief overview in this section of the results of our work in the following chapters.

We begin our study in Chapter 2 of how to manage massive data by first investigating the challenge of dealing with the ever-increasing *size* of data, e.g., such as weather measurements collected by MesoWest [5] since 1997. For many applications using relational data, obtaining

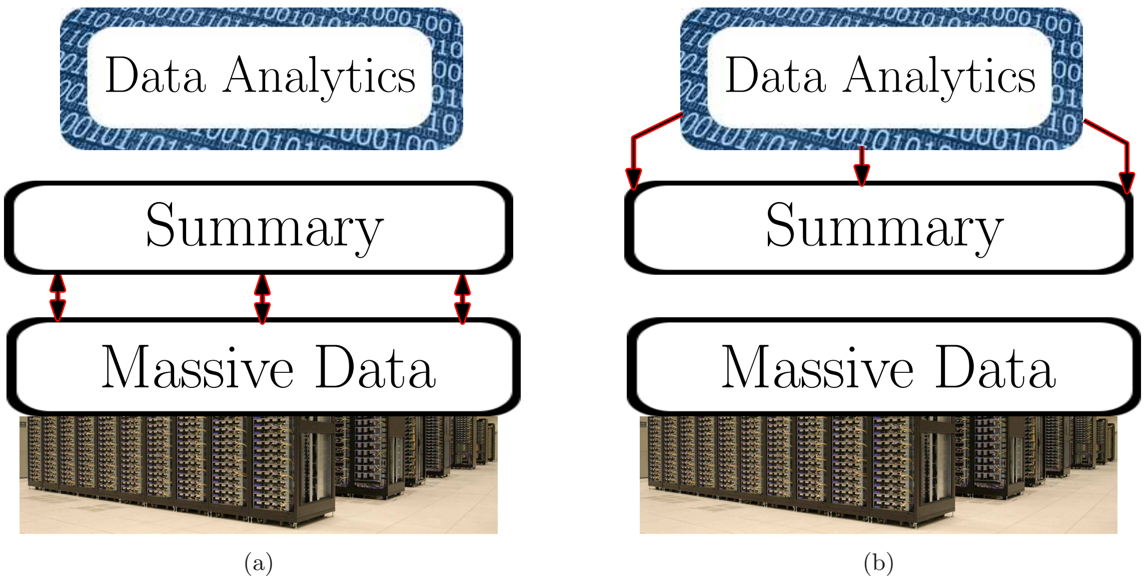


Figure 1.3. A summary-based approach for querying massive data: (a) building a summary and (b) querying the summary.

a compact and accurate summary of data is essential. Among various data summarization tools, histograms have proven to be particularly important and useful for summarizing data, and the wavelet histogram is one of the most widely used histograms [11]. Due to its simplicity, good accuracy, small size, and a variety of applications in data analysis, data visualization, query optimization, and its usefulness in approximating queries, wavelet histograms have been extensively studied [11–14]. Surprisingly, the problem of constructing wavelet histograms, in a scalable fashion, over truly massive relational data had not been studied.

We investigate the problem of building wavelet histograms efficiently on large data in parallel and distributed settings. We demonstrate straightforward adaptations of existing exact and approximate methods for building wavelet histograms in a parallel and distributed setting that are highly inefficient, in terms of both communication and computation. We design new algorithms for computing exact and approximate wavelet histograms and discuss their implementation in MapReduce. We illustrate our techniques in Hadoop, and compare to baseline solutions with extensive experiments performed in a heterogeneous Hadoop cluster of 16 nodes, using large real and synthetic datasets, up to hundreds of gigabytes. The results show significant (often orders of magnitude) improvement by our new algorithms.

Next, we turn our focus to a second challenge, the management of large data with *complex structure and rich semantics* in Chapter 3, namely how to efficiently manage and summarize temporal data represented in a piecewise linear representation. In particular, we are interested in an important query type typically performed over temporal data, e.g., given temporal data, such as sensors collecting temperature data over a long period of time as in the MesoWest project [5], a typical query is to find which sensors have the highest aggregate temperature in a given arbitrary time range.

With this particular query type in mind, we propose a novel ranking operator over temporal data to rank objects based on the aggregation of their scores in a query interval, which we dub the *aggregate* top-k query on temporal data. For example, return the top-10 weather stations having the highest average temperature from 10/01/2010 to 10/07/2010. Chapter 3 presents a comprehensive study to this problem by first designing exact solutions and then investigating techniques to summarize the data using both the ranking operator and indexing techniques to produce approximate solutions (the approximate solutions all have quality guarantees).

To study the effectiveness of our proposed techniques, we study real-world massive datasets from the MesoWest project [5], which contains temperature measurements from 26,383 distinct stations across the United States from Jan 1997 to Oct 2011, and the MemeTracker project [15], which had tracked 1.5 million distinct memes by 2012. We demonstrate an orders of magnitude query performance gain when using our data summaries over exact solutions which use I/O efficient indexing structures to query the original dataset.

We study the challenge of efficiently managing and querying *uncertain data* in Chapter 4. As data become increasingly large, uncertain data are becoming ubiquitous for many applications, such as applications collecting sensor data as well as many scientific applications. This was made apparent to us by observing individual sensor readings from the MesoWest [5] and SAMOS projects [16], which frequently reported uncertain measurements. In Chapter 4, we observe that recently, there have been several attempts to propose definitions and algorithms for ranking queries on probabilistic data, in order to summarize it. However, these lack many intuitive properties of a top-k over deterministic data.

Therefore, we task ourselves to define numerous fundamental properties, including *exact-k*, *containment*, *unique-rank*, *value-invariance*, and *stability*, which are satisfied by ranking queries on certain relational data. We argue these properties should also be carefully studied in defining ranking queries in probabilistic data, and fulfilled by definition for ranking uncertain data for most applications.

We propose intuitive new ranking definitions based on the observation that the ranks of a tuple across all possible worlds represent a well-founded rank distribution. We studied the ranking definitions based on the expectation, the median, and other statistics of this rank distribution for a tuple and derived the *expected rank, median rank and quantile rank* respectively. We are able to prove the expected rank, median rank and quantile rank satisfy all these properties for a ranking query. We provide efficient solutions in centralized settings to compute such rankings across major models of uncertain data, such as attribute-level and tuple-level uncertainty. A comprehensive experimental study shows the effectiveness of our approach.

In Chapter 5 we investigate the challenge of massive *distributed data*. In particular, we extend our study of *uncertain data* in Chapter 4 by investigating the problem of ranking *uncertain data* in a *distributed setting*. We observe that in many applications where uncertainty and fuzzy information arise, data are collected from multiple sources in distributed, networked locations, e.g., distributed sensor fields with imprecise measurements, multiple scientific institutes with inconsistency in their scientific data, or data generated and distributed within a cluster. Due to the desire for efficient queries with low latency coupled with resource restrictions such as limited communication bandwidth in a sensor network or heat restrictions in a cluster, a fundamental problem we face when constructing a summary in this setting is to reduce communication cost and computation costs to the extent possible. Given this insight, we design both communication and computation efficient algorithms which we show perform orders of magnitude better than the baseline technique over both real and synthetic data. After designing such algorithms, uncertain data which are either distributed in a sensor network or within a cluster can be efficiently summarized to only the top-k most important records for consumption by users.

In conclusion, in Chapters 2-5, we study four challenges arising from massive data, the ever-increasing *size* of data, emerging *complex structures and rich semantics* of data, *uncertain data*, and *distributed data*. In all cases, we propose novel summarization techniques to construct the *histogram* and *ranking operator* which allow queries and data analytics tasks to be answered orders of magnitude faster while still providing quality guarantees.

1.3 Outline of Thesis

The remainder of our thesis is structured as follows:

- First, we study the challenge of dealing with the ever-increasing *size* of data in Chapter 2 and propose novel techniques for constructing wavelet histograms over this massive data to summarize it.
- We then study the challenge of managing *complex structures and rich semantics* in

massive data in Chapter 3, in particular focusing on temporal data and how to answer aggregate range ranking queries over a piecewise linear representation of these data.

- Next, we change our focus to the challenge of *uncertain data* in Chapter 4 by observing its semantics and what this entails for massive data management. We propose novel querying techniques using the ranking operator, with our contribution being the median and quantile rank.
- We study the challenge of *distributed data* in Chapter 5 by extending our study of *uncertain data* in Chapter 4 to the *distributed and parallel* case, and propose novel techniques to use the *Expected Rank*, discussed in Chapter 4, to summarize it to the most important top-k records.
- Finally, we discuss some of our other works in Chapter 6 and conclude in Chapter 7

CHAPTER 2

BUILDING WAVELET HISTOGRAMS ON LARGE DATA

2.1 Introduction

One of the first challenges we face when dealing with massive data is the ever-increasing *size* of data, as argued in Chapter 1. In traditional relational database systems and many modern data management applications, an important useful summary for datasets is the *histogram* [18], and this compact data structure is becoming increasingly important as the *size* of data continues to grow. Given the importance of the histogram, in this chapter, we study how to efficiently construct histograms, in particular the *wavelet histogram* [11].

Suppose the keys of a dataset are drawn from finite domain $[u] = \{1, \cdots, u\}$. Broadly speaking, a histogram on the dataset is any compact, possibly lossy, representation of its frequency vector $\mathbf{v} = (\mathbf{v}(1), \ldots, \mathbf{v}(u))$, where $\mathbf{v}(x)$ is the number of occurrences of key x in the dataset. There are many different histograms depending on the form this compact representation takes. One popular choice is the wavelet histogram. Treating $\mathbf{v}$ as a signal, the wavelet histogram consists of the top-k wavelet coefficients of $\mathbf{v}$ in terms of their magnitudes (absolute values), for a parameter k. As most real-world distributions have few large wavelet coefficients with others close to zero, retaining only the k largest yields a fairly accurate representation of $\mathbf{v}$. Due to its simplicity, good accuracy, compact size, and a variety of applications in data analysis, data visualization, query optimization, and approximating queries, wavelet histograms have been extensively studied. Efficient algorithms are well known for building a wavelet histogram on offline data [11, 12] and for dynamically maintaining it in an online or streaming [12–14] fashion.

As data continue to increase in *size*, it is becoming infeasible to store it in a single database or on a single machine. It has become increasingly common to store this massive data in clusters and, currently, one of the most popular cluster frameworks for managing

*The work in this chapter appears in [17] ©2012 VLDB Endowment. Reprinted by Permission.

and querying large data is MapReduce. In this chapter, we use the MapReduce framework to illustrate our ideas of how the wavelet histogram can be efficiently constructed in a distributed and parallel fashion, although our techniques can be extended to other parallel and distributed settings as well.

MapReduce has become one of the most popular cluster-based frameworks for storing and processing massive data, due to its excellent scalability, reliability, and elasticity [19]. Datasets stored and processed in MapReduce, or any large data management platform, are usually enormous, ranging from tens of gigabytes to terabytes [19, 20]. Hence, in many applications with such massive data, obtaining a compact accurate summary of a dataset is important. Such a summary captures essential statistical properties of the underlying data distribution, and offers quick insight on the gigantic dataset, provided we can compute it efficiently. For example, in the MapReduce framework, this allows other MapReduce jobs over the same dataset to better partition the dataset utilizing its histogram, which leads to better load-balancing in the MapReduce cluster [19].

In this chapter, we study how to efficiently build wavelet histograms for large data in MapReduce. We utilize *Hadoop* [7], an open-source realization of MapReduce, to demonstrate our ideas, which should extend to any other MapReduce implementation, as well as other similar parallel and distributed platforms. We measure the efficiency of all algorithms in terms of *end-to-end running time* (affected by the computation and IO costs) and *intracluster communication* (since network bandwidth is also scarce in large data centers running MapReduce [19], whose usage needs to be optimized). Note that communication cost might not be significant when running only one particular MapReduce job (this is often the case); however, in a busy data center/cluster where numerous jobs might be running simultaneously, the aggregated effect from the total communications of these jobs is still critical.

We show straightforward adaptations of both exact and approximate wavelet histogram construction methods from traditional data management systems and data mining fields to MapReduce clusters are highly inefficient, mainly since data are stored in a distributed file system, e.g., the Hadoop Distributed File System (HDFS).

2.1.1 Contributions

We propose novel exact and approximation algorithms demonstrated in MapReduce clusters, in particular Hadoop, which outperform straightforward adaptations of existing methods by several orders of magnitude in performance. Specifically, we:

- present a straightforward adaptation of the exact method in Hadoop, and a new exact method that can be efficiently instantiated in MapReduce in Section 2.3;
- show how to apply existing, sketch-based approximation algorithms in Hadoop, and discuss their shortcomings. We design a novel random sampling scheme to compute approximate wavelet histograms efficiently in Hadoop in Section 2.4;
- conduct extensive experiments on large (up to 400GB) data-sets in a heterogeneous Hadoop cluster with 16 nodes in Section 2.5. The experimental results demonstrate convincing results that both our exact and approximation methods have outperformed their counterparts by several orders of magnitude.

We also introduce necessary background on MapReduce, Hadoop, and wavelet histograms in Section 2.2, survey related work in Section 2.6, and conclude in Section 2.7.

2.2 Preliminaries

2.2.1 Wavelet basics

Suppose each record in the dataset has a key drawn from domain $[u] = \{1, \cdots, u\}$, and we want to build a wavelet histogram on the keys. Define the frequency vector as $\mathbf{v} = (\mathbf{v}(1), \ldots, \mathbf{v}(u))$ where $\mathbf{v}(x)$ is the number of occurrences of key x in the dataset. The idea of building a histogram using wavelets is to consider $\mathbf{v}$ as a signal and apply a wavelet transformation. For most applications, one usually adopts the simplest Haar wavelet basis [11–14,21], which is defined as follows. We first average values pairwise to obtain the *average coefficients*, i.e., $[(\mathbf{v}(2)+\mathbf{v}(1))/2, (\mathbf{v}(4)+\mathbf{v}(3))/2, \ldots, (\mathbf{v}(u)+\mathbf{v}(u-1))/2]$. We also retain the average difference of the pairwise values, i.e., $[(\mathbf{v}(2)-\mathbf{v}(1))/2, \ldots, (\mathbf{v}(u)-\mathbf{v}(u-1))/2]$, which are called the *detail coefficients*. Clearly, given these vectors, one can reconstruct the original signal $\mathbf{v}$ exactly. We recursively apply this pairwise averaging and differencing process on the *average coefficients vector* until we reach the *overall average* for $\mathbf{v}$. The Haar *wavelet coefficients* of $\mathbf{v}$ are given by the overall average, followed by the detail coefficients in a binary tree, as shown by example in Figure 2.1, where the leaf level of the tree (level $\ell = \log u$) is the original signal. To preserve the energy of the signal ($\mathbf{v}$'s L_2 norm), one must multiply coefficients in level ℓ by a scaling factor $\sqrt{u/2^{\ell}}$.

This transformation is lossless as we can reconstruct $\mathbf{v}$ exactly from all u wavelet coefficients. However, the main reason wavelets are popular and powerful in signal processing is, for most real-world signals $\mathbf{v}$, most of its wavelet coefficients are near zero. Thus, if for a parameter k we keep only the k wavelet coefficients of largest magnitude while assuming others

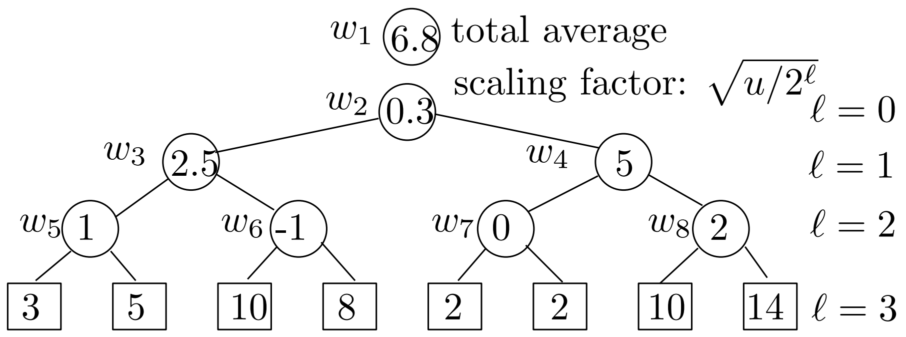


Figure 2.1. Wavelet coefficients.

are zero, we can still reconstruct the original signal reasonably well. Since *energy* is preserved under the Haar wavelet transform after scaling, i.e., $\|\mathbf{v}\|_2^2 = \sum_{i=1}^{u} \mathbf{v}(i)^2 = \sum_{i=1}^{u} w_i^2$, keeping the k wavelet coefficients of largest magnitude minimizes energy loss for all k-term wavelet representations of $\mathbf{v}$ [14]. The best k-term wavelet representation can be computed efficiently in a centralized setting [11]: Assuming entries in frequency vector $\mathbf{v}$ are given in order, one can compute all wavelet coefficients bottom-up in $O(u)$ time. Then, using a priority queue of size k, we can find the k coefficients of largest magnitude in one pass over all u coefficients, taking time $O(u \log k)$.

Another method to compute wavelet coefficients, especially in streaming settings, is to use wavelet basis vectors. The first wavelet basis vector is $\psi_1 = [1, \ldots, 1]/\sqrt{u}$. To define the other $u - 1$ basis vectors, we first introduce, for $j = 1, \ldots, \log u$ and $k = 0, \ldots, 2^j - 1$, the vector $\phi_{j,k}(l) = 1$ for $k(u/2^j) + 1 \le l \le k(u/2^j) + u/2^j$, and 0 elsewhere. For $j = 0, \ldots, \log u - 1$ and $k = 0, \ldots, 2^j - 1$, we define the ith wavelet basis vector for $i = 2^j + k + 1$ as $\psi_i = (-\phi_{j+1,2k} + \phi_{j+1,2k+1})/\sqrt{u/2^j}$, where $\sqrt{u/2^j}$ is a scaling factor. The wavelet coefficients are the dot products of $\mathbf{v}$ with these wavelet basis vectors, i.e., $w_i = \langle \mathbf{v}, \psi_i \rangle$, for $i = 1, \ldots, u$; see an illustration of this process in Table 2.1.

Wavelets provide a compact approximation of a data distribution and the wavelet histogram serves a variety of data analysis tasks such as range selectivity estimation [11], approximating queries [22], and many other data mining applications [23–25]. As we are concerned with constructing a best k-term wavelet histogram, we will not talk about its use, which has already been well studied [11].

Wavelet histograms also extend to multidimensional signals or datasets. Consider the two-dimensional case where keys are drawn from two-dimensional domain $[u]^2$, defining a two-dimensional frequency array $\mathbf{v} = (\mathbf{v}(x, y)), 1 \le x, y \le u$. A 2D wavelet transform first applies a standard 1D wavelet transform to each row of $\mathbf{v}$. Then, using the 1D wavelet

Table 2.1. Coefficients by wavelet basis vectors

$\frac{(\mathbf{v}(1)+\mathbf{v}(2)+\mathbf{v}(3)+\mathbf{v}(4)+\mathbf{v}(5)+\mathbf{v}(6)+\mathbf{v}(7)+\mathbf{v}(8))}{2\sqrt{2}}$							
$\frac{(\mathbf{v}(5)+\mathbf{v}(6)+\mathbf{v}(7)+\mathbf{v}(8))-(\mathbf{v}(1)+\mathbf{v}(2)+\mathbf{v}(3)+\mathbf{v}(4))}{2\sqrt{2}}$							
$\frac{(\mathbf{v}(3)+\mathbf{v}(4))-(\mathbf{v}(1)+\mathbf{v}(2))}{2}$				$\frac{(\mathbf{v}(7)+\mathbf{v}(8))-(\mathbf{v}(5)+\mathbf{v}(6))}{2}$			
$\frac{\mathbf{v}(2)-\mathbf{v}(1)}{\sqrt{2}}$		$\frac{\mathbf{v}(4)-\mathbf{v}(3)}{\sqrt{2}}$		$\frac{\mathbf{v}(6)-\mathbf{v}(5)}{\sqrt{2}}$		$\frac{\mathbf{v}(8)-\mathbf{v}(7)}{\sqrt{2}}$	
$\mathbf{v}(1)$	$\mathbf{v}(2)$	$\mathbf{v}(3)$	$\mathbf{v}(4)$	$\mathbf{v}(5)$	$\mathbf{v}(6)$	$\mathbf{v}(7)$	$\mathbf{v}(8)$

coefficients as inputs, we apply a second round of 1D wavelet transforms to each column of the array. This process can be similarly extended to d dimensions.

2.2.2 Hadoop basics

For this chapter, we assume Hadoop's default file system HDFS. A cluster using HDFS consists of multiple DataNodes, for storing file system data, and a single master node designated as the NameNode which oversees all file operations and maintains all file meta-data. A file in HDFS is split into *data chunks*, 64MB in size by default, which are allocated to DataNodes by the NameNode. Chunks are typically replicated to multiple DataNodes, based on the file replication ratio, to increase data availability and fault tolerance. In this chapter and many other studies where fault tolerance is not the main subject of interest, the replication ratio is set to 1 and machine failure is not considered. The MapReduce core consists of one master JobTracker task and many TaskTracker tasks. Typical configurations run the JobTracker and NameNode on the same machine, called the *master*, and run TaskTracker and DataNode tasks on other machines, called *slaves*.

Typical MapReduce jobs consist of three phases: Map, Sort-and-Shuffle, and Reduce. The user may specify m, the desired number of Mapper tasks, and r, the number of Reducer tasks before starting the job. Next, we look at the three phases in detail.

In the Map phase, the m Mappers run in parallel on different TaskTrackers over different logical portions of an input file, called *splits*. Splits typically, but not always, correspond to physical data chunks. Hadoop allows users to specify the InputFormat for a file, which determines how splits are created and defines a RecordReader for reading data from a split.

After splits have been formed, the JobTracker assigns each available Mapper a split to process. By default, the scheduler attempts to schedule Data-Local Mappers by assigning a Mapper a locally stored split. There are also cases which call for Non-Data-Local Mappers, i.e., when a node is idle and has no local split to process. Then, a MapRunner is started which obtains a RecordReader and invokes the Map function for each record in the split. A Mapper then maps input key-value pairs (k_1, v_1) from its split to intermediate key-value pairs (k_2, v_2). As a Mapper proceeds, it maintains an in-memory buffer of the (k_2, v_2); for each distinct k_2 a list of values, $list(v_2)$, is maintained. When the buffer fills to threshold, pairs are partitioned, sorted, and optionally processed by the Combine function, which outputs locally aggregated (k_2, v_2) pairs (aggregation on v_2s with the same key k_2). Pairs are then written to their corresponding logical partitions on the local disk. The partitions are defined by a Partition function, typically a hash function like $hash(k_2) \mod r$, which determines the Reducer task that will process a particular k_2 later. When the Mapper ends, all emitted (k_2, v_2) have been partitioned, sorted (w.r.t. k_2), and optionally combined. One can also define a Close interface which executes at the end of the Mapper.

In the Shuffle-and-Sort Phase, each Reducer copies all (k_2, v_2) for which it is responsible (as designated by the Partition function) from all DataNodes. It then sorts all received (k_2, v_2) by k_2 so all occurrences of key k_2 are grouped together. An external sort is needed if the (k_2, v_2) do not fit in memory.

After all (k_2, v_2) are collected and sorted, a Reducer iterates over all its (k_2, v_2). For each distinct key k_2, the Reducer passes all corresponding v_2 values to the Reduce function. Then, the Reduce function produces a final key-value pair (k_3, v_3) for every intermediate key k_2. As in the Map phase, one can implement a Close interface which is executed at the end of the Reducer. An example of a typical MapReduce job appears in Figure 2.2.

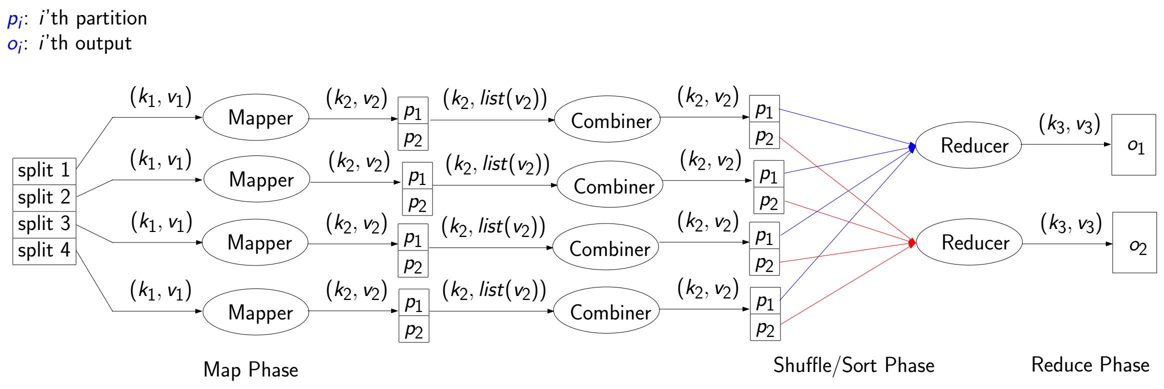


Figure 2.2. An example MapReduce job with $m = 4$ and $r = 2$.

2.3 Exact Computation

2.3.1 Baseline solutions

Let n be the total number of records in the entire dataset, where each record has a key drawn from key domain $[u]$. Note either $n \gg u$ or $n \ll u$ is possible. Recall in Hadoop the n records are partitioned into m splits, processed by m Mappers, possibly on different machines, which emit intermediate key-value pairs for processing by Reducers. Thus, one baseline solution to compute the wavelet representation is to compute, for each split $j = 1, \ldots, m$, its local frequency vector $\mathbf{v}_j$, and emit a $(x, \mathbf{v}_j(x))$ pair for each key x in the split.

After all local frequencies are computed, a Reducer can aggregate the local frequencies producing the overall frequency vector $\mathbf{v} = \sum_{j=1}^{m} \mathbf{v}_j$, where $\mathbf{v}_j(x) = 0$ if key x does not appear in the jth split. Finally, we compute the best k-term wavelet representation of $\mathbf{v}$ using the centralized algorithm (e.g., [11]).

We observe that each wavelet coefficient $w_i = \langle \mathbf{v}, \psi_i \rangle$ can be written as

$$w_i = \left\langle \sum_{j=1}^{m} \mathbf{v}_j, \psi_i \right\rangle = \sum_{j=1}^{m} \langle \mathbf{v}_j, \psi_i \rangle,$$

i.e., w_i is the summation of the corresponding local wavelet coefficients of frequency vectors for the m splits. Then, an alternate approach to compute the exact wavelet coefficients is to compute, for each split $j = 1, \ldots, m$ its local frequency vector $\mathbf{v}_j$. The local coefficients $w_{i,j} = \langle \mathbf{v}_j, \psi_i \rangle$ are computed for each split's local frequency vector $\mathbf{v}_j$ and a $(i, w_{i,j})$ pair is emitted for each nonzero $w_{i,j}$. The Reducer can then determine the exact w_i as $\sum_{j=1}^{m} w_{i,j}$ where $w_{i,j} = 0$ if the Reducer does not receive a $w_{i,j}$ from the jth split. After computing all complete w_i the Reducer selects the best k-term wavelet representation, i.e., by selecting the top-k coefficients of largest absolute value.

A big drawback of the baseline solutions is they generate too many intermediate key-value pairs, $O(mu)$ of them to be precise. This consumes too much network bandwidth, which is a scarce resource in large data clusters shared by many MapReduce jobs [19].

2.3.2 A new algorithm

Since w_i is the summation of the corresponding local wavelet coefficients of frequency vectors for the m splits, if we first compute the local coefficients $w_{i,j} = \langle \mathbf{v}_j, \psi_i \rangle$, the problem is essentially a distributed top-k problem. For the standard distributed top-k problem, it is assumed there are N total items and m distributed sites. Each of the m distributed sites has a local score $r_j(x)$ for item x. The goal is to obtain the top-k items at a coordinator which have the largest aggregate scores $r(x) = \sum_{j=1}^{m} r_j(x)$. The standard distributed top-k problem assumes all local "scores" are nonnegative, while in our case, wavelet coefficients can be *positive and negative*, and we want to find the top-k aggregated coefficients of *largest absolute value* (magnitude). Negative scores and finding largest absolute values are a problem for existing top-k algorithms such as Three-Phase Uniform Threshold (TPUT) and others [26–28], as they use a "partial sum" to prune items which cannot be in the top-k. That is, if we have seen (at the coordinator, e.g., in the MapReduce model the coordinator is a single reducer responsible for producing the global top-k items) t local scores for an item out of m total local scores, we compute a partial sum for it assuming its other $m - t$ scores are zero. When we see k such partial sums, we use the kth largest partial sum as a threshold, denoted τ, to prune other items: If an item's local score is always below τ/m at all sites, it can be pruned as it cannot get a total score larger than τ to get in the top-k. If there are negative scores and when the goal is to find largest absolute values, we cannot compute such a threshold as unseen scores may be very negative.

We next present a distributed algorithm which handles positive and negative scores (coefficients) and returns the top-k aggregated scores of largest magnitude. The algorithm is based on algorithm TPUT [27], and can be seen as interleaving two instances of TPUT. As TPUT, our algorithm requires three rounds. For an item x, $r(x)$ denotes its aggregated score and $r_j(x)$ is its score at node j.

2.3.2.1 Round 1

Each node first emits the k highest and k lowest (i.e., most negative) scored items. For each item x seen at the coordinator, we compute a lower bound $\tau(x)$ on its total score's magnitude $|r(x)|$ (i.e., $|r(x)| \geq \tau(x)$), as follows. We first compute an upper bound $\tau^+(x)$ and a lower bound $\tau^-(x)$ on its total score $r(x)$ (i.e., $\tau^-(x) \leq r(x) \leq \tau^+(x)$): If a node sends out the score of x, we add its exact score to $\tau^+(x)$ and $\tau^-(x)$. Otherwise, for $\tau^+(x)$, we add the kth highest score this node sends out and for $\tau^-(x)$ we add the kth lowest score. Then, we set $\tau(x) = 0$ if $\tau^+(x)$ and $\tau^-(x)$ have different signs and $\tau(x) = \min\{|\tau^+(x)|, |\tau^-(x)|\}$ otherwise. Doing so ensures $\tau^-(x) \leq r(x) \leq \tau^+(x)$ and $|r(x)| \geq \tau(x)$. Now, we pick the kth largest $\tau(x)$, denoted as T_1. This is a threshold for the magnitude of the top-k items.

2.3.2.2 Round 2

A node j next emits all local items x having $|r_j(x)| > T_1/m$. This ensures an item in the true top-k in magnitude must be sent by at least one node after this round, because if an item is not sent, its aggregated score's magnitude can be no higher than T_1. Now, with more scores available from each node, we refine the upper and lower bounds $\tau^+(x)$, $\tau^-(x)$, hence $\tau(x)$, as previously for each item $x \in R$, where R is the set of items ever received. If a node did not send the score for some x, we can now use T_1/m (resp. $-T_1/m$) for computing $\tau^+(x)$ (resp. $\tau^-(x)$). This produces a new better threshold, T_2 (calculated in the same way as computing T_1 with improved $\tau(x)$'s), on the top-k items' magnitude.

Next, we further prune items from R. For any $x \in R$, we compute its new threshold $\tau'(x) = \max\{|\tau^+(x)|, |\tau^-(x)|\}$ based on refined upper and lower bounds $\tau^+(x), \tau^-(x)$. We delete item x from R if $\tau'(x) < T_2$. The final top-k items must be in the set R.

2.3.2.3 Round 3

Finally, we ask each node for the scores of all items in R. Then, we compute the aggregated scores exactly for these items, from which we pick the k items of largest magnitude. A simple optimization is, in Round 2, to not send an item's local score if it is in the local top-k/bottom-k sets, even if $|r_j(x)| > T_1/m$; these scores were sent in Round 1. Also in Round 3, a node can send an item's local score only if it was not sent to the coordinator in previous rounds (using simple local bookkeeping).

2.3.2.4 Communication issues

Our algorithm makes novel extensions to TPUT [27] to support both positive and negative scores at distributed sites. The original TPUT algorithm has no nontrivial bound on communication cost and its worst- and best-case bounds carry over to our algorithm. In the worst case, the pruning of our algorithm may be completely ineffective and all local scores $r_j(x)$ may need to be sent to the coordinator to compute the global top-k aggregate scores $r(x)$; however, we see this is not the case in practice through extensive experimental results in Section 2.5. In the best case, our algorithm will only need to communicate $O(mk)$ local scores.

2.3.3 Implementation details

At a high-level, for a dataset in HDFS with m splits, we assign one Mapper task per split and each Mapper acts as a distributed node. We use one Reducer as the coordinator. We implement our three-round algorithm in three rounds of MapReduce in Hadoop. To be consistent across rounds, we identify each split with its unique offset in the original input file. Two technical issues must be dealt with when implementing the algorithm in Hadoop.

First, the algorithm is designed assuming the coordinator and distributed nodes are capable of bidirectional communication. However, in MapReduce, data normally flow in one direction, from Mappers to Reducers. In order to have two-way communication, we utilize two Hadoop features: the Job Configuration and Distributed Cache. The Job Configuration is a small piece of information communicated to every Mapper and Reducer task during task initialization. It contains some global configuration variables for Mappers and Reducers. The Job Configuration is good for communicating a small amount of information. If large amounts of data must be communicated, we use Hadoop's Distributed Cache feature. A file can be submitted to the master for placement into the Distributed Cache. Then, Distributed Cache content is replicated to all slaves during the MapReduce job initialization.

Second, the distributed nodes and the coordinator in the algorithm need to keep persistent state across three rounds. To do so, at the end of a Mapper task handling an input split, via its Close interface, we write all necessary state information to an HDFS file with a file name identifiable by the split's id. When this split is assigned to a Mapper in a subsequent round, the Mapper can then restore the state information from the file. Note Hadoop always tries to write an HDFS file locally if possible, i.e., state information is usually saved on the same machine holding the split, so saving state information in an HDFS file incurs almost no extra communication cost. For the Reducer which acts as the coordinator, since there is no split associated to it, we choose to customize the JobTracker scheduler so the Reducer is always executed on a designated machine. Thus, the coordinator's state information is saved locally on this machine.

We dub our new algorithm, tailored for Hadoop as discussed above, *Hadoop wavelet top-k*, or *H-WTopk* for short. Below, we detail how we implement all three rounds of H-WTopk in Hadoop.

2.3.3.1 H-WTopk MapReduce Round 1

In Round 1, a Mapper first computes local frequency vector $\mathbf{v}_j$ for split j by using a hashmap to aggregate the total count for each key encountered as the records in the split are scanned. After $\mathbf{v}_j$ is constructed, we compute its wavelet coefficients in the Close interface of the Mapper. Since the number of nonzero entries in $\mathbf{v}_j$, denoted as $|\mathbf{v}_j|$, is typically much smaller than u, instead of running the $O(u)$-time algorithm of [11], we use the $O(|\mathbf{v}_j| \log u)$-time and $O(\log u)$-memory algorithm of [13]. During the computation, we also keep two priority queues to store the top-k and bottom-k wavelet coefficients.

After all coefficients for split j have been computed, the Mapper emits an intermediate key-value pair $(i, (j, w_{i,j}))$ for each of the top-k and bottom-k wavelet coefficients $w_{i,j}$ of the split. In the emitted pairs, the Mapper marks the kth highest and the kth lowest coefficients using $(i, (j+m, w_{i,j}))$ and $(i, (j+2m, w_{i,j}))$, respectively. Finally, the Mapper saves all unemitted coefficients as state information to an HDFS file associated with split j.

After the Map phase, the Reducer receives the top-k and bottom-k wavelet coefficients from all the splits, $2km$ of them in total. We denote by R the set of distinct indices of the received coefficients. For each index $i \in R$, The Reducer passes the corresponding $(j, w_{i,j})$s received to a Reduce function, which adds up these $w_{i,j}$s, forming a partial sum $\widehat{w}_i$ for w_i. Meanwhile we construct a bit vector F_i of size m such that $F_i(j) = 0$ if $w_{i,j}$ has been received and $F_i(j) = 1$ if not. While examining the $(j, w_{i,j})$s in the Reduce function, if we encounter a marked pair, we remember it so the kth highest and the kth lowest coefficient from each split j, denoted as $\tilde{w}_j^+$ and $\tilde{w}_j^-$, can be obtained.

After we have all partial sums $\widehat{w}_i$ for all $i \in R$, and $\tilde{w}_j^+, \tilde{w}_j^-$ for all j, we compute upper bound τ_i^+ (resp. lower bound τ_i^-) on w_i, by adding $\sum_{j=1}^{m} F_i(j)\tilde{w}_j^+$ (resp. $\sum_{j=1}^{m} F_i(j)\tilde{w}_j^-$) to $\widehat{w}_i$. Then we obtain a lower bound τ_i on $|w_i|$, hence T_1, as described in Section 2.3. Finally, we save tuple $(i, \widehat{w}_i, F_i)$ for all $i \in R$, and T_1 as state information in a local file on the Reducer machine.

2.3.3.2 H-WTopk MapReduce Round 2

To start Round 2, T_1/m is first set as a variable in the Job Configuration. In this round, for the Map phase, we define an alternate InputFormat so a Mapper does not read an input split at all. Instead, a Mapper simply reads state information, i.e., all wavelet coefficients not sent in Round 1, one by one. For any $w_{i,j}$ such that $|w_{i,j}| > T_1/m$, a Mapper emits the pair $(i, (j, w_{i,j}))$.

The Reducer first reads tuple $(i, \widehat{w}_i, F_i)$ for all $i \in R$ from the local file written in Round 1. For each i, it passes all corresponding $(j, w_{i,j})$s received in this round to a Reduce function. Now, we update partial sum $\widehat{w}_i$ by adding these new coefficients, and update F_i correspondingly. We also refine upper bound τ_i^+ (resp. lower bound τ_i^-) as $\tau_i^+ = \widehat{w}_i + \|F_i\|_1 \cdot T_1/m$ (resp. $\tau_i^- = \widehat{w}_i - \|F_i\|_1 \cdot T_1/m$), where $\|F_i\|_1$ denotes the number of 1s in F_i.

With the updated τ_i^+, τ_i^-, we obtain a new T_2, which can be used to prune indices from R as described in Section 2.3. Lastly, the Reducer writes updated $\widehat{w}_i$ for all $i \in R$ in a local file, and the set of candidate indices R in an HDFS file.

2.3.3.3 H-WTopk MapReduce Round 3

In Round 3, the master reads R from HDFS and adds it to the Distributed Cache. Like in Round 2, the Mappers still do not read the input splits. During initialization, each

Mapper reads R from the distributed cache. Then, it reads from the state file storing the wavelet coefficients. For any $w_{i,j}$ it checks if $i \in R$ and $|w_{i,j}| \leq T_1/m$. If so, it means it has not been communicated to the Reducer yet, and thus we emit $(i, (j, w_{i,j}))$.

On the Reducer side, similar to Round 2, the Reducer first reads R and $\widehat{w}_i$ for all $i \in R$'s from the local file. Then for each i, the Reduce function adds all newly received $w_{i,j}$'s to $\widehat{w}_i$, yielding accurate w_i. Finally, we return the top-k coefficients w_i of largest magnitude for $i \in R$ as the best k-term representation for $\mathbf{v}$.

2.3.4 Multidimensional wavelets

It is straightforward to extend our algorithms to build multidimensional wavelet histograms. Consider the two-dimensional case. Recall in this case frequency vector $\mathbf{v}$ is a 2D array. A 2D wavelet transform applies two rounds of 1D wavelet transforms on the rows and then the columns of $\mathbf{v}$. Since each wavelet transform is a linear transformation, the resulting 2D wavelet coefficients are still linear transformations of $\mathbf{v}$. So if we apply a 2D wavelet transform to each split, any 2D wavelet coefficient is still a summation of corresponding 2D coefficients of all splits. Thus, we can still run the modified TPUT algorithm to find the top-k coefficients of largest magnitude as before.

2.4 Approximate Computation

We observe that the exact computation of the best k-term wavelet representation in Hadoop is expensive. Although our improved algorithm avoids emitting all local frequency vectors, it could still be expensive due to the following: (1) The (modified) TPUT algorithm could still send out a lot of communication, though better than sending all local frequency vectors; (2) it needs 3 rounds of MapReduce, which incurs a lot of overhead; and (3) most importantly, every split needs to be scanned to compute local frequency vector $\mathbf{v}_j$ and compute local wavelet coefficients $w_{i,j}$. This motivates us to explore approximation algorithms which compute a k-term wavelet representation which may not be the best one, but still approximates the underlying data distribution reasonably well.

There are many design choices for approximate computation of wavelets. Here are some natural attempts: (i) We can replace TPUT with an approximate top-k algorithm [28, 29], after appropriate modification to handle negative scores. This resolves issue (1) but not (2) and (3). (ii) We can approximate local wavelet coefficients of each split using a sketch as in [13, 14], and then send out and combine the sketches, due to the property that these sketches are linearly combinable. This resolves issues (1) and (2), but not (3), as computing a sketch still needs to scan the data once. (iii) Lastly, a generic approach is random sampling, that is, we take a random sample of the keys and construct the wavelets on the sample, as the sample approximates the underlying data distribution well for a sufficiently large sample size. Then, a wavelet representation can be constructed on the frequency vector of the sample.

Among the possibilities, only (iii) resolves all three issues simultaneously. It requires only one round, clearing issue (2). It also avoids reading the entire dataset, clearing issue (3). However, it may result in a lot of communication, as it is well known to approximate each (global) frequency $\mathbf{v}(x)$ with a standard deviation of εn (recall n is the number of records in the entire dataset), a sample of size $\Theta(1/\varepsilon^2)$ is required [30]. More precisely, for a sample probability $p = 1/(\varepsilon^2 n)$ (a sample of expected size $pn = 1/\varepsilon^2$), one can show $\widehat{\mathbf{v}}(x) = \mathbf{s}(x)/p$ is an unbiased estimator of $\mathbf{v}(x)$ with standard deviation $O(\varepsilon n)$ for any x, where $\mathbf{s}$ is the frequency vector of the sample. After that, we construct a wavelet representation on the estimated frequency vector $\widehat{\mathbf{v}}$. As n is the size of the entire dataset, which is usually extremely large (for MapReduce clusters), ε needs to be fairly small for $\widehat{\mathbf{v}}$ to approximate $\mathbf{v}$ well, usually on the order of 10^{-4} to 10^{-6}. The total communication cost of this *basic sampling* method is $O(1/\varepsilon^2)$, even with one-byte keys, this corresponds to 100MB to 1TB of data being emitted to the network!

A straightforward improvement is to summarize the sampled keys of a split before emitting them, which is actually used as a simple optimization for executing any MapReduce job [19]. We aggregate the keys with the Combine function, that is, if the split is emitting c pairs $(x, 1)$ for the same key, they are aggregated as one pair (x, c). This optimization indeed reduces communication cost, but its effectiveness highly depends on the data distribution; in the worst case it may not reduce the communication at all.

A slightly better idea is to ignore those sampled keys with low frequencies in a split, which we denote as the *improved sampling* algorithm. More precisely, we only send out a sampled key x and its sampled count $\mathbf{s}_j(x)$ if $\mathbf{s}_j(x) \geq \varepsilon t_j$, where t_j is the total number of sampled records in split j. Thus, the overall error in the total count of a sampled key x from all splits is at most $\sum_{j=1}^{m} \varepsilon t_j = \varepsilon pn = 1/\varepsilon$, which translates into an $(1/\varepsilon)/p = \varepsilon n$ error in the estimated frequency $\widehat{\mathbf{v}}(x)$. Thus, it adds another εn to the standard deviation, which is still $O(\varepsilon n)$. Note that the total number of key-value pairs sent out by one split is at most $t_j/(\varepsilon t_j) = 1/\varepsilon$. Hence, the total communication of this approach is at most $O(m/\varepsilon)$, which improves upon sending all the samples since usually we have $m \ll 1/\varepsilon$. However, an undesired consequence is $\widehat{\mathbf{v}}(x)$ will not be unbiased any more: $\mathbf{E}[\widehat{\mathbf{v}}(x)]$ could be εn away from $\mathbf{v}(x)$, since this method ignores all the small sample counts $\mathbf{s}_j(x) < \varepsilon t_j$.

Below we detail a new, *two-level sampling* idea, which produces an unbiased estimator $\widehat{\mathbf{v}}(x)$ for $\mathbf{v}(x)$ with standard deviation $O(\varepsilon n)$ as in the basic random sampling algorithm, while improving communication cost to $O(\sqrt{m}/\varepsilon)$. The idea is to obtain an unbiased estimator $\widehat{\mathbf{s}}(x)$ of $\mathbf{s}(x)$, instead of sending all $\mathbf{s}_j(x)$s to compute $\mathbf{s}(x)$ exactly. We then use $\widehat{\mathbf{s}}(x)$ to produce $\widehat{\mathbf{v}}(x)$. We perform another level of sampling on the local frequency vector $\mathbf{s}_j$ of sampled keys for each split j. Specifically, we sample each key x in the sample with probability $\min\{\varepsilon\sqrt{m}\cdot\mathbf{s}_j(x), 1\}$. More precisely, for any x with $\mathbf{s}_j(x) \geq 1/(\varepsilon\sqrt{m})$, we emit the pair $(x, \mathbf{s}_j(x))$; for any x with $0 < \mathbf{s}_j(x) < 1/(\varepsilon\sqrt{m})$, we sample it with a probability proportional to $\mathbf{s}_j(x)$, i.e., $\varepsilon\sqrt{m}\cdot\mathbf{s}_j(x)$, and emit the pair $(x$, NULL$)$ if it is sampled (for an example please see Figure 2.3). Note that in two-level sampling, we do not throw away sampled items with small frequencies completely, as is done in the *improved sampling* method. Rather, these items are still given a chance to survive in the second-level sample, by sampling them proportional to their frequencies relative to the threshold $1/\varepsilon\sqrt{m}$ (which is established from our analysis below).

Next, we show how to construct from the emitted pairs from all splits, an unbiased estimator $\widehat{\mathbf{s}}(x)$ of $\mathbf{s}(x)$ for any key $x \in [u]$ with standard deviation at most $1/\varepsilon$. As $\mathbf{s}(x) = \sum_{j=1}^{m}\mathbf{s}_j(x)$, we add up all sample count $(x, \mathbf{s}_j(x))$ pairs received for x. They do not introduce any error, and we denote this partial sum as $\rho(x)$. If a split has $\mathbf{s}_j(x) < 1/(\varepsilon\sqrt{m})$, the mapper processing the split will not emit $\mathbf{s}_j(x)$, but simply emit $(x$, NULL$)$ if x is sampled. Suppose we receive M such pairs for x. Then, our estimator is

$$\widehat{\mathbf{s}}(x) = \rho(x) + M/(\varepsilon\sqrt{m}) \tag{2.1}$$

(for an example of how we compute $\widehat{\mathbf{s}}(x)$ at the reducer please see Figure 2.4).

Theorem 2.1. *$\widehat{\mathbf{s}}(x)$ is an unbiased estimator of $\mathbf{s}(x)$ with standard deviation at most $1/\varepsilon$.*

Proof of Theorem 2.1: Without loss of generality, assume in the first m' splits $\mathbf{s}_j(x) < 1/(\varepsilon\sqrt{m})$. Write M as $M = \sum_{j=1}^{m'} X_j$ where $X_j = 1$ if x is sampled in split j and 0 otherwise. Each X_j is an independent Bernoulli trial, so

$$\mathbf{E}[X_j] = \varepsilon\sqrt{m}\cdot\mathbf{s}_j(x), \text{ and}$$

$$\mathbf{Var}[X_j] = \varepsilon\sqrt{m}\cdot\mathbf{s}_j(x)(1-\varepsilon\sqrt{m}\cdot\mathbf{s}_j(x)) \leq \varepsilon\sqrt{m}\cdot\mathbf{s}_j(x). \tag{2.2}$$

Thus, we have

$$\mathbf{E}[M] = \sum_{j=1}^{m'}\varepsilon\sqrt{m}\cdot\mathbf{s}_j(x) = \varepsilon\sqrt{m}(\mathbf{s}(x)-\rho(x)), \tag{2.3}$$

i.e., $\mathbf{E}[\widehat{\mathbf{s}}(x)] = \mathbf{s}(x)$ combining (2.1) and (2.3).

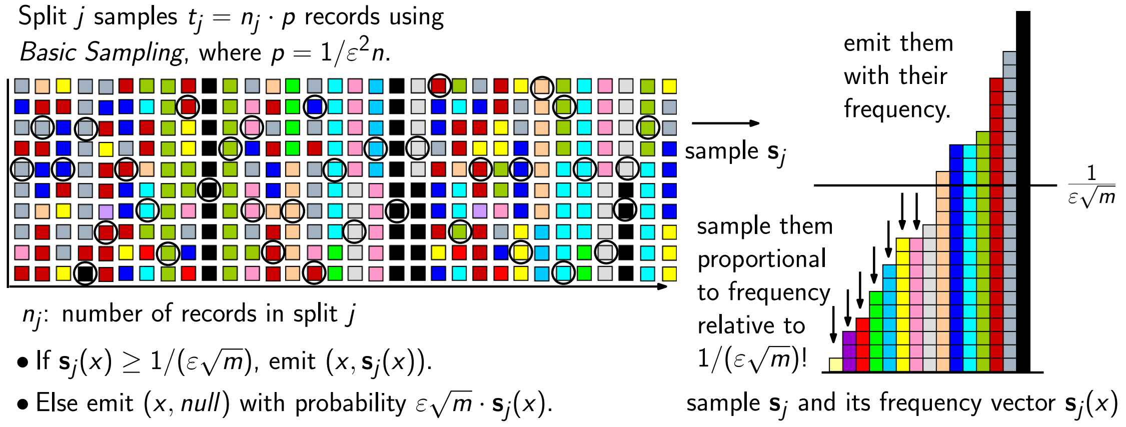

Split j samples $t_j = n_j \cdot p$ records using
Basic Sampling, where $p = 1/\varepsilon^2 n$.
n_j: number of records in split j
• If $\mathbf{s}_j(x) \geq 1/(\varepsilon\sqrt{m})$, emit $(x, \mathbf{s}_j(x))$.
• Else emit $(x, null)$ with probability $\varepsilon\sqrt{m} \cdot \mathbf{s}_j(x)$.
sample $\mathbf{s}_j$
emit them with their frequency.
$\frac{1}{\varepsilon\sqrt{m}}$
sample them proportional to frequency relative to $1/(\varepsilon\sqrt{m})$!
sample $\mathbf{s}_j$ and its frequency vector $\mathbf{s}_j(x)$

Figure 2.3. Two-level sampling at mapper.

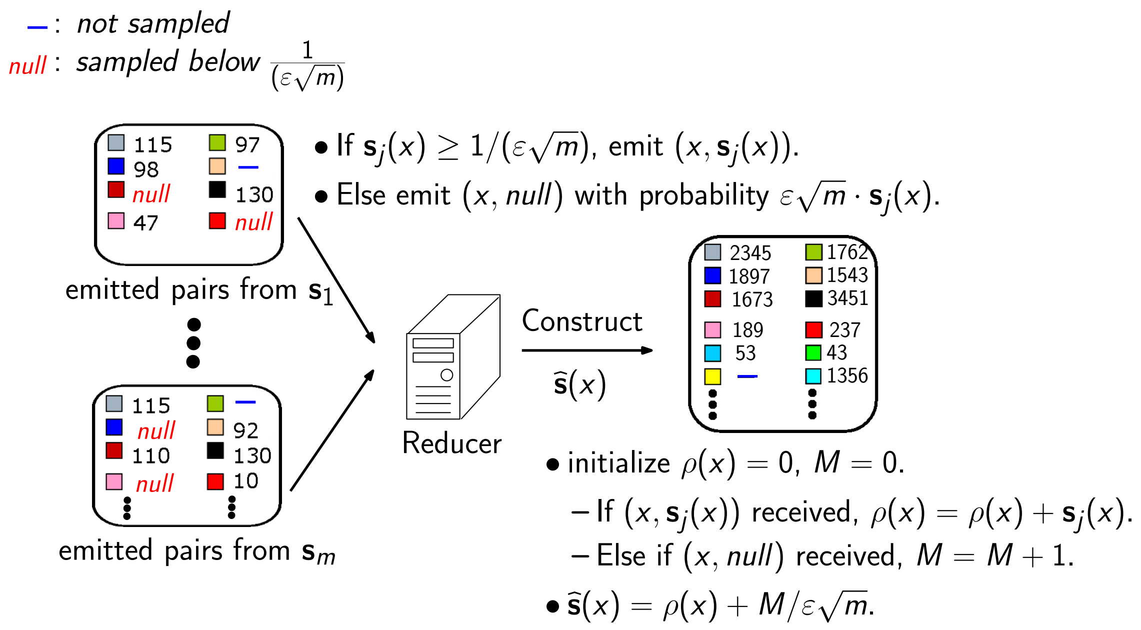

—: *not sampled*
null: *sampled below* $\frac{1}{(\varepsilon\sqrt{m})}$
115
97
98
—
null
130
47
null
emitted pairs from $\mathbf{s}_1$
• If $\mathbf{s}_j(x) \geq 1/(\varepsilon\sqrt{m})$, emit $(x, \mathbf{s}_j(x))$.
• Else emit $(x, null)$ with probability $\varepsilon\sqrt{m} \cdot \mathbf{s}_j(x)$.
115
—
null
92
110
130
null
10
emitted pairs from $\mathbf{s}_m$
Reducer
Construct
$\hat{\mathbf{s}}(x)$
2345
1762
1897
1543
1673
3451
189
237
53
43
—
1356
• initialize $\rho(x) = 0$, $M = 0$.
– If $(x, \mathbf{s}_j(x))$ received, $\rho(x) = \rho(x) + \mathbf{s}_j(x)$.
– Else if $(x, null)$ received, $M = M + 1$.
• $\hat{\mathbf{s}}(x) = \rho(x) + M/\varepsilon\sqrt{m}$.

Figure 2.4. Two-level sampling at reducer.

Next, from (2.2), we have

$$\mathbf{Var}[M] = \sum_{j=1}^{m'} \mathbf{Var}[X_j] = \varepsilon\sqrt{m} \cdot \sum_{j=1}^{m'} \mathbf{s}_j(x). \tag{2.4}$$

Since each $\mathbf{s}_j(x) \leq 1/(\varepsilon\sqrt{m})$, $\mathbf{Var}[M]$ is at most m'. Thus, the variance of $\widehat{\mathbf{s}}(x)$ is $\mathbf{Var}[M/(\varepsilon\sqrt{m})] = \mathbf{Var}[M]/(\varepsilon^2 m)$. So $\mathbf{Var}[\widehat{\mathbf{s}}(x)] \leq m'/(\varepsilon^2 m) \leq 1/\varepsilon^2$, namely, the standard deviation is at most $1/\varepsilon$. ■

From $\widehat{\mathbf{s}}(x)$, we can estimate $\mathbf{v}(x)$ as $\widehat{\mathbf{v}}(x) = \widehat{\mathbf{s}}(x)/p$ (recall that $p = 1/(\varepsilon^2 n)$ is the sampling probability of the first level random sample in each split). It will be an unbiased estimator of $\mathbf{v}(x)$ with standard deviation $(1/\varepsilon)/p = \varepsilon n$.

Corollary 2.1. *$\widehat{\mathbf{v}}(x)$ is an unbiased estimator of $\mathbf{v}(x)$ with standard deviation at most εn.*

Corollary 2.1 gives a bound on the error of the estimated frequencies. Below, we also analyze the error in the computed wavelet coefficients. Consider the coefficient $w_i = \langle \mathbf{v}, \psi_i \rangle$, where $\psi_i = (-\phi_{j+1,2k} + \phi_{j+1,2k+1})/\sqrt{u/2^j}$ is the corresponding wavelet basis vector (see discussion in Section 2.2.1). From the estimated frequency vector $\widehat{\mathbf{v}}$, we estimate w_i as $\widehat{w}_i = \langle \widehat{\mathbf{v}}, \psi_i \rangle$. Since $\widehat{\mathbf{v}}(x)$ for every x is unbiased, $\widehat{w}_i$ is also an unbiased estimator of w_i. Recall that $\psi_i(x) = -1, +1$ for $x = 2ku/2^{j+1} + 1, \ldots, (2k+2)u/2^{j+1}$, so the variance of $\widehat{w}_i$ is

$$\begin{aligned}
\mathbf{Var}[\widehat{w}_i] &= \frac{2^j}{u} \sum_{x=2ku/2^{j+1}+1}^{(2k+2)u/2^{j+1}} \mathbf{Var}[\widehat{\mathbf{v}}(x)] \\
&= \frac{2^j}{u} \sum_{x=2ku/2^{j+1}+1}^{(2k+2)u/2^{j+1}} \mathbf{Var}[\widehat{\mathbf{s}}(x)]/p^2 \\
&= \frac{2^j}{u} \sum_{x=2ku/2^{j+1}+1}^{(2k+2)u/2^{j+1}} \mathbf{Var}[M]/(\varepsilon^2 m p^2) \\
&\leq \frac{2^j n}{um} \sum_{x=2ku/2^{j+1}+1}^{(2k+2)u/2^{j+1}} \varepsilon\sqrt{m}\mathbf{s}(x) \qquad \text{(by (2.4))} \\
&= \frac{\varepsilon 2^j n}{u\sqrt{m}} \sum_{x=2ku/2^{j+1}+1}^{(2k+2)u/2^{j+1}} \mathbf{s}(x).
\end{aligned} \tag{2.5}$$

Note that $\sum_{x=2ku/2^{j+1}+1}^{(2k+2)u/2^{j+1}} \mathbf{s}(x)$ is just the total number of keys covered by the wavelet basis vector. This discussion leads to the next result:

Theorem 2.2. *The two-level sampling method provides an unbiased estimator $\widehat{w}_i$ for any wavelet coefficient w_i, and the variance of $\widehat{w}_i$ is bounded by* (2.5).

Finally, it remains to bound its communication cost.

Theorem 2.3. *The expected total communication cost of our two-level sampling algorithm is $O(\sqrt{m}/\varepsilon)$.*

Proof of Theorem 2.3: The expected total sample size of first-level sampling is $pn = 1/\varepsilon^2$. Thus, there are at most $(1/\varepsilon^2)/(1/(\varepsilon\sqrt{m})) = \sqrt{m}/\varepsilon$ keys with $\mathbf{s}_j(x) \geq 1/(\varepsilon\sqrt{m})$ across all splits. These keys must be emitted for second-level sampling. For any key x in any split j with $\mathbf{s}_j(x) < 1/(\varepsilon\sqrt{m})$, we emit it with probability $\varepsilon\sqrt{m} \cdot \mathbf{s}_j(x)$, so the expected total number of sampled keys for this category is

$$\sum_j \sum_x \varepsilon\sqrt{m} \cdot \mathbf{s}_j(x) \leq \varepsilon\sqrt{m} \cdot 1/\varepsilon^2 = \sqrt{m}/\varepsilon.$$

So the total number of emitted keys is $O(\sqrt{m}/\varepsilon)$. ■

Consider typical values: $m = 10^3, \varepsilon = 10^{-4}$ and 4-byte keys. Basic sampling emits $1/\varepsilon^2 \approx$ 400MB; improved sampling emits at most $m/\varepsilon \approx$ 40MB; while two-level sampling emits about $\sqrt{m}/\varepsilon \approx$ 1.2MB of data—a 330-fold or 33-fold reduction, respectively!

Remark: In our second-level sampling, the sampling probability depends on the frequency, so that "important" items are more likely to be sampled. This falls into the general umbrella of "importance sampling" [31], and has been used for frequency estimation on distributed data [32,33]. However, its application to wavelet histograms and the corresponding variance analysis are new.

2.4.1 Multidimensional wavelets

Our algorithm extends to constructing multidimensional wavelet histograms naturally. In d dimensions, frequency vector $\mathbf{v}$ is a d-dimensional array, and frequency array $\mathbf{s}$ of a random sample of the dataset still approximates $\mathbf{v}$. So the problem boils down to how well $\mathbf{s}$ approximates $\mathbf{v}$ (note our two-level sampling algorithm does not affect the approximation error of the sample). However, because data are usually sparse in higher dimensions, the quality of the sample may not be as good as in one dimension. In fact, the standard deviation of the estimated frequency for any $\mathbf{v}(x)$ (x is now a cell in $[u]^d$) from a sample of size $O(1/\varepsilon^2)$ is still $O(\varepsilon n)$, but due to the sparsity of the data, all the $\mathbf{v}(x)$s may be small, so the relative error becomes larger. This is, unfortunately, an inherent problem with sparse data: if all $\mathbf{v}(x)$s are small, say 0 or 1, then random sampling, and in general any sublinear method, cannot possibly achieve small relative errors [34]. One remedy is to lower the granularity of the data, i.e., project the data to a smaller grid $[u/t]^d$ for some appropriate t so as to increase the density of the data.

2.4.2 System issues

Among the three general approximation strategies mentioned at the beginning of Section 2.4, implementing the approximate TPUT methods (such as KLEE [28]) in Hadoop requires at least three rounds of MapReduce, which involves too much overhead for just approximating a wavelet histogram. Wavelet sketches can be easily implemented in Hadoop. The idea is to run one Mapper per split, which builds a local wavelet sketch for the split and emits the nonzero entries in the sketch to the Reducer. The Reducer then combines these m sketches and estimates the top-k coefficients from the combined sketch. There are two wavelet sketches in the literature: the AMS sketch [13, 35] and the GCS sketch [14]. The latter was shown to have better performance, so we choose it to implement in Hadoop. There are some technical details in optimizing its implementation in Hadoop, which we omit here.

The third strategy, random sampling, clearly has better performance as it avoids scanning the entire dataset and is also easy to implement in Hadoop. Our two-level sampling algorithm in addition achieves very low communication cost. We detail how we address some system issues, overcome the challenges, and implement two-level sampling in Hadoop in the following.

2.4.3 Implementation details of TwoLevel-S

We will have m Mappers, one per input split, and 1 Reducer. The first issue is how to randomly read records from an input split. The default Hadoop RecordReader in InputFile format is designed to sequentially scan an input split. Hence, we define our own InputFile format *RandomInputFile*, assuming each record in the input files has a fixed size. The RandomInputFile defines a custom RecordReader, called RandomRecordReader, which can randomly sample records from an input split. A straightforward implementation is to simply seek to a random offset in the split when the Mapper requests the next record, but this requires seeking offset locations in both directions. Instead, we implement it as follows.

When the RandomRecordReader is first initialized, it determines n_j, the number of records in the split. Next, it randomly selects pn_j offsets in the split, where $p = 1/(\varepsilon^2 n)$ is the sample probability of the first-level sampling, and stores them in a priority queue Q sorted in ascending order.

Afterwords, every time the RandomRecordReader is invoked by the Mapper to retrieve the next record from the split, it seeks to the record indicated by the next offset, and retrieves the record there. We continue this process iteratively until all pn_j random records have been obtained. Note in Section 2.4, we assume coin-flip sampling for sake of simpler analysis; here we use sampling without replacement. It has been observed coin-flip sampling and sampling without replacement behave almost the same for most sampling-based methods [30], and we observe this is also true for our sampling-based approaches.

Using RandomInputFile as the InputFile format, two-level sampling can be implemented in one round of MapReduce, as follows.

2.4.3.1 Map phase

During initialization of the Map phase, we specify n and ε in the Job Configuration. With the RandomRecordReader, the MapRunner reads the pn_j random records one at a time and invokes the Map function for each record, which simply maintains aggregated counts for keys of the sampled records. After the MapRunner has processed all sampled records, the Mapper's Close routine is called. It iterates over all sampled keys and checks their aggregate counts. If $\mathbf{s}_j(x) \geq 1/(\varepsilon\sqrt{m})$, we emit the pair $(x, \mathbf{s}_j(x))$. Otherwise, we emit $(x, 0)$ with probability $\varepsilon\sqrt{m} \cdot s_j(x)$.

2.4.3.2 Reduce phase

For each key x, the Reducer passes all corresponding $(x, \mathbf{s}_j(x))$ or $(x, 0)$ pairs to the Reduce function, which computes the estimated $\widehat{\mathbf{v}}(x)$ as described in Section 2.4. After all keys are processed by the Reducer, its Close method is invoked, where approximate wavelet coefficients are computed from approximate global frequency vector $\widehat{\mathbf{v}}$. In the end, we emit (i, w_i) pairs for the top-k approximate coefficients (with the k largest magnitudes).

2.4.3.3 Remarks

In our discussion so far, our RandomRecordReader assumes fixed length records. However, it is easy to extend it to support variable length records as well. Instead, assume records of variable length end with a delimiter character or byte sequence (e.g., a new line character). As a preprocessing step, the offsets of all records within a split j are computed and stored as fixed-length integer values in an HDFS file associated with split j. This is easily accomplished with a single scan over all splits, which is a one-time cost. The RandomRecordReader will now read from two files, the record offset file for split j and split j. The RandomRecordReader first generates pn_j random offsets within the record offset file, reads the associated record offsets, and inserts them in a priority queue Q. Note, the number of records n_j within split j are easy to determine based on the size of the record offset file associated with split j since each offset is a fixed-length integer value. The

RandomRecordReader then proceeds to process offsets from Q one at a time by seeking to an offset within split j and scanning forward until it finds the record delimiter.

In the implementation of our exact and sampling methods, we choose to do the final processing in the close method, instead of using the combiner. This is a technicality due to the default behavior of Hadoop, which runs the COMBINE function continuously while keys are being processed to save memory buffer space (leads to fewer disk writes). On the other hand, the close method is guaranteed to run only once when all keys have been processed.

2.5 Experiments

We implement all algorithms in Hadoop and empirically evaluate their performance, in both end-to-end running time and communication cost. For the exact methods, we denote the baseline solution of sending all local frequency vectors (the $\mathbf{v}_j$s of all splits) in Section 2.3 as *Send-V*, the baseline solution of sending the local wavelet coefficients (the $w_{i,j}$s of all splits) in Section 2.3 as *Send-Coef*, and our new algorithm as *H-WTopk* (meaning "Hadoop wavelet top-k"). For the approximate algorithms, we denote the basic sampling method as *Basic-S*, the improved sampling method as *Improved-S*, and the two-level sampling method as *TwoLevel-S*. Note *Improved-S* is based on the same idea as *Basic-S*, but offers strictly better performance, which we derived in Section 2.4. Given this fact, we choose to utilize *Improved-S* as the default competitor of *TwoLevel-S*. We also implement the sketch-based approximation method as discussed in Section 2.4. We use the GCS-sketch which is the state-of-the-art sketching technique for wavelet approximations [14]. We denote this method as *Send-Sketch*. We did not attempt to modify the approximate TPUT methods (such as KLEE [28]) to work with negative values and adapt them to MapReduce, since they generally require multiple rounds and scanning the entire datasets, which will be strictly worse than other approximation methods.

2.5.1 Setup and datasets

All experiments are performed on a heterogeneous Hadoop cluster running the latest stable version of Hadoop, version 0.20.2. The cluster consists of 16 machines with four different configurations: (1) 9 machines with 2GB of RAM and one Intel Xeon 5120 1.86GHz CPU, (2) 4 machines with 4GB of RAM and one Intel Xeon E5405 2GHz CPU, (3) 2 machines with 6GB of RAM and one Intel Xeon E5506 2.13GHz CPU, and (4) 1 machine with 2GB of RAM and one Intel Core 2 6300 1.86GHz CPU. The Hadoop NameNode and TaskTracker run on the same machine with configuration (2) and we select one of the machines of configuration (3) to run the (only) Reducer. We configure Hadoop to use

300GB of hard drive space on each slave and allocate 1GB memory per Hadoop daemon. We have one TaskTracker and one DataNode daemon running on each slave, and a single NameNode and JobTracker daemon on the master. All machines are directly connected to a 100Mbps switch.

For the datasets, clearly, the determining parameters are n, the total number of records, which corresponds to the size of the input file, and u, the domain size, as well as the skewness. Note it only makes sense to use a dataset which is at least tens of gigabytes and has a domain size on the order of 2^{20}. Otherwise a centralized approach would work just fine, and the overhead of running MapReduce could actually lead to worse performance [19].

That said, for real datasets, we test all algorithms on the *WorldCup* [36] dataset, which is the access logs of 92 days from the 1998 World Cup servers, a total of approximately 1.35 billion records. Each record consists of 10 4-byte integer values including month, day, and time of access as well as the client id, object id, size, method, status, and accessed server. We assign to each record a 4-byte identifier *clientobject*, which uniquely identifies a distinct client id and object id pairing. The object id uniquely identifies a URL referencing an object stored on the World Cup servers, such as a page or image. The pairing of the client id and the object id is useful to analyze the correlation between clients and resources from the World Cup servers, under the same motivation as that in the more common example of using the (src ip, dest ip) pairing in a network traffic analysis scenario. There are approximately 400 million distinct client id object id combinations, so the domain of this key value is approximately 2^{29}, i.e., $u = 2^{29}$. We store *WorldCup* in binary format, and in total, the stored dataset is 50GB.

To model the behavior of a broad range of real large datasets, we also generate datasets following the Zipfian distribution (since most real datasets, e.g., the *clientobject* in *WorldCup*, are skewed with different levels of skewness), with various degrees of skewness α, as well as different u and n. We randomly permute keys in a dataset to ensure the same keys do not appear contiguously in the input file. Each dataset is stored in binary format and contain records with only a 4-byte integer key. Unless otherwise specified, we use the Zipfian dataset as our default dataset to vigorously test all approaches on a variety of parameters on large-scale data.

We vary α in $\{0.8, 1.1, 1.4\}$ and $\log_2 u$ in { 8, 11, 14, 17, 20, 23, 26, 29, 32 }. We vary input file size from 10GB to 200GB, resulting in different n from 2.7 to 54 billion. We vary the size of a record from 4-bytes to 100kB. For all algorithms, we use 4-byte integers to represent $\mathbf{v}(x)$ in a Mapper and 8-byte integers in a Reducer. We represent wavelet coefficients and sketch entries as 8-byte doubles.

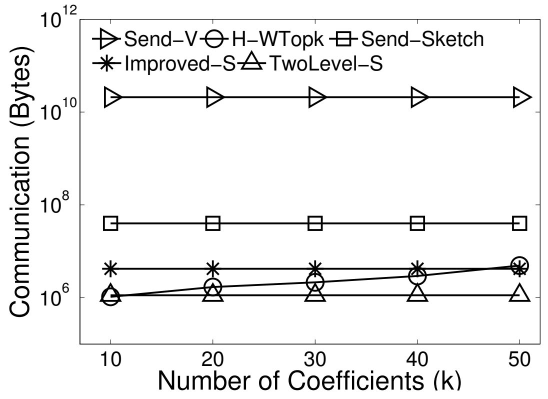
For all experiments, we vary one parameter while keeping the others fixed at their default values. Our default α is 1.1 and $\log_2 u$ is 29. The default dataset size is 50GB (so the default n is 13.4 billion). The default record size is 4-bytes. We compute the best k-term wavelet histogram with $k = 30$ by default, which also varies from 10 to 50. The default split size β is 256MB, which varies from 64MB to 512MB. Note that the number of splits is $m = 4n/(1024^2\beta)$ (so the default m is 200). We also simulate a live MapReduce cluster running in a large data center where typically multiple MapReduce jobs are running at the same time, which share the network bandwidth. Thus, the default available network bandwidth is set to 50% (i.e., 50Mbps) but we also vary it from 10% to 100%. Note, we omit the results for *Send-Coef* on all experiments except for varying the domain u of the input dataset as it performs strictly worse than *Send-V* for other experiments.

The exact methods have no parameters to tune. For *Send-Sketch*, we use a recommended setting for the GCS-sketch from [14], where each sketch is allocated 20KB$\cdot \log_2 u$ space. We use GCS-8 which has the overall best per-item update cost and a reasonable query time to obtain the final coefficients. We also did the following optimizations: First, for each split, we compute the local frequency vector $\mathbf{v}_j$, and then insert the keys into the sketch so we update the sketch only once for each distinct key. Second, we only send nonzero entries in a local sketch to the Reducer. For the two sampling methods, the default ε is 10^{-4}, and we vary it from 10^{-5} to 10^{-1}.

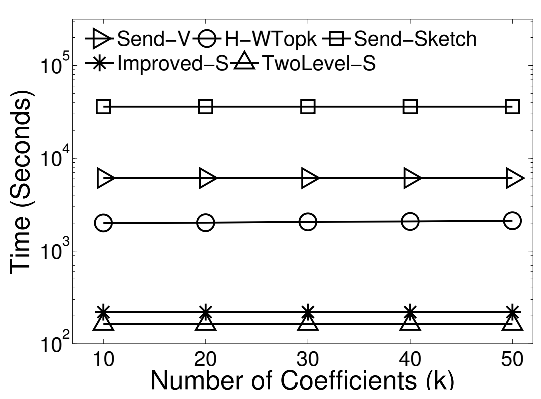
2.5.2 Results on varying k

We first study the effect of k, i.e., the size of the wavelet histogram to be computed. Figure 2.5 shows the effect of varying k on the communication cost and running time of all algorithms. The results show k has little impact on performance, except for the communication cost of *H-WTopk*. This is expected, as *Send-V* (resp. *Send-Sketch*) always compute and send out all local frequency vectors (resp. their sketches). The sampling methods are also unaffected by k as the sampling rate is solely determined by m and ε. However, *H-WTopk*'s communication cost is closely related to k, as it determines thresholds T_1 and T_2 for pruning items.

For the exact methods, *H-WTopk* outperforms *Send-V* by orders of magnitude, in both communication and running time. It also outperforms *Send-Sketch*, which is an approximate method. The two sampling algorithms are clearly the overall winners. Nevertheless, among the two, in addition to a shorter running time, *TwoLevel-S* reduces communication to 10%–20% compared to *Improved-S*. Recall our analysis indicates an $O(\sqrt{m})$-factor reduction



(a)



(b)

Figure 2.5. Cost analysis vary k: effect on (a) communication and (b) running time.

from *Improved-S* to *TwoLevel-S*; but this assumes arbitrary input data. Due to the skewness of the Zipfian data distribution, *Improved-S* actually combines many keys into one key-value pair, and thus typically does not reach its $O(m/\varepsilon)$ upper bound on communication. Overall, the sampling algorithms have impressive performance: On this 50GB dataset, *TwoLevel-S* incurs only 1MB communication and finishes in less than 3 minutes. In contrast, *Send-Sketch* takes about 10 hours (most time is spent updating local sketches), *Send-V* about 2 hours (mostly busy communicating data), and *H-WTopk* 33 minutes (scanning inputs plus overhead for 3 rounds of MapReduce).

We must ensure the efficiency gain of the sampling methods does not come with a major loss of quality. Thus, we examine the sum of squared error (SSE) between the frequency vector reconstructed from the wavelet histogram and that of the original dataset. The results are shown in Figure 2.6. Since *Send-V* and *H-WTopk* are exact methods, they represent the best possible reconstruction using any k-term wavelet representation. So

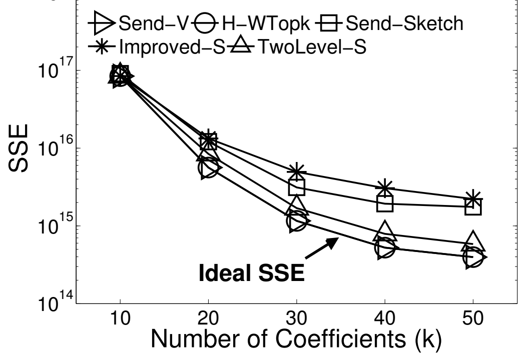


Figure 2.6. SSE: vary k.

their curves are identical in Figure 2.6 and represent the ideal error for measuring the accuracy of the approximation methods. Clearly, when k increases, the SSEs of all methods decrease. Among the three approximate methods, *TwoLevel-S* returns wavelet histograms which come very close to the ideal SSE. *Improved-S* has the worst SSE, as it is not an unbiased estimator for $\mathbf{v}$, and the gap from the ideal SSE widens as k gets larger, as it is not good at capturing the details of the frequency vector. *Send-Sketch*'s SSE is between *TwoLevel-S* and *Improved-S*. Even though the SSE looks large in terms of absolute values, it is actually quite small considering the gigantic dataset size. When $k \geq 30$, the SSE is less than 1% of the original dataset's energy.

2.5.3 Varying ε

Next, we explore the impact of ε on all sampling methods, by varying it from 10^{-5} to 10^{-1} in Figure 2.7. In all cases, *TwoLevel-S* consistently achieves significantly better accuracy than *Improved-S*, as the first is an unbiased estimator of $\mathbf{v}$ while the latter is not. Both methods have larger SSEs when ε increases, with $\varepsilon = 10^{-4}$ achieving a reasonable balance between the SSE and efficiency (to be shown next), hence it is chosen as the default. Figure 2.8 shows all sampling methods have higher costs when ε decreases (from right to left). In all cases, *TwoLevel-S* has significantly lower communication cost than *Improved-S*, as seen in Figure 2.8(a). In addition, as shown in Figure 2.8(b), it has a lower running time than *Improved-S*. In a busy data center where network bandwidth is shared by many concurrent jobs, the savings in communication by *TwoLevel-S* will prove to be critical and the gap for the running time will widen even more.

In what follows, we omit the results on SSEs when we vary the other parameters, as they have less impact on the SSEs of various methods, and the relative trends on SSEs for

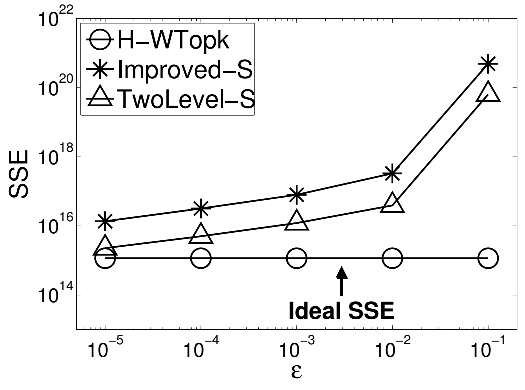
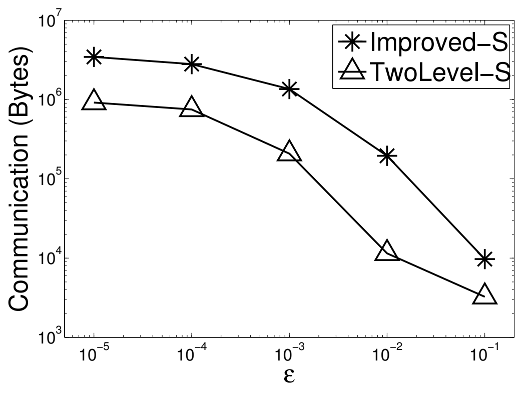
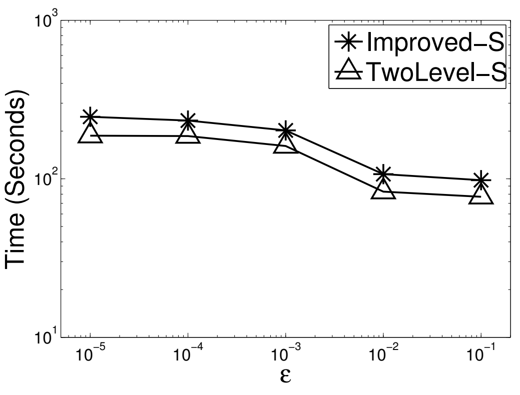


Figure 2.7. SSE: vary ε.



(a)



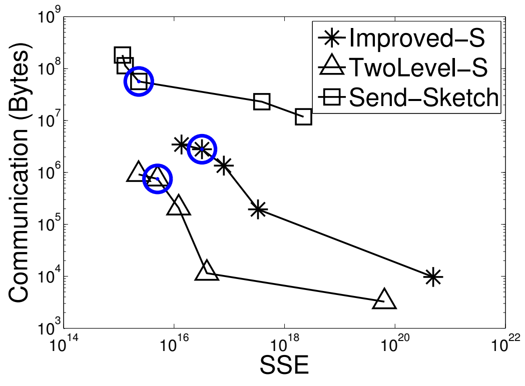
(b)

Figure 2.8. Cost analysis vary ε: effect on (a) communication and (b) running time.

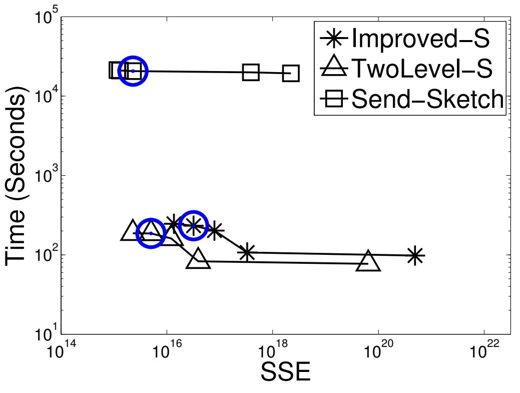
all methods are always similar to those reported in Figures 2.6 and 2.7.

2.5.4 Comparing SSE

For the next experiment, we analyze the communication and computation overheads of all approximation algorithms to achieve a similar SSE in Figure 2.9, where the defaults of all algorithms are circled. In Figure 2.9(a), we see that the communication cost increases as the SSE decreases for all algorithms. *TwoLevel-S* achieves the best SSE to communication cost, and communicates at least an order of magnitude less than *Improved-S* and two orders of magnitude less than *Send-Sketch* to achieve a similar SSE. Among the algorithms, *TwoLevel-S* is the most efficient in terms of running time, achieving a similar SSE to *Send-Sketch* in orders of magnitude less time and approximately 2-3 times less time than *Improved-S*, as



(a)



(b)

Figure 2.9. SSE versus (a) communication and (b) running time.

shown in Figure 2.9(b). These results also indicate the sketch size selected at 20kB * $\log_2(u)$ is most competitive against the sampling-based algorithms, justifying our choice for using it as the default value for the GCS-sketch.

2.5.5 Varying dataset size n

Next, we analyze the scalability of all methods by varying n, or equivalently the dataset size. Note as n increases, so does m, the number of splits. This explains the general trends in Figure 2.10 for both communication and running times. There are two points worth pointing out. First, *TwoLevel-S* outperforms *Improved-S* by a larger margin in terms of communication cost for larger datasets due to the $O(\sqrt{m})$-factor difference, which becomes more than one order of magnitude when the data becomes 200GB. Second, the increase in m leads to longer running times of all methods, but the two sampling algorithms are much less affected. The reason is the sampling algorithms mainly have two kinds of running time costs: overheads associated with processing each split (i.e., Mapper initialization), which linearly depends on m, and sampling overheads where the sample size is always $\Theta(1/\varepsilon^2)$, which is independent of n. The net effect of these costs is a slow growth in running time. Overall, *H-WTopk* and *TwoLevel-S* are clearly the best exact and approximate methods, respectively.

2.5.6 Varying record size

In Figure 2.11 we analyze the effect varying the record size has on the performance of all algorithms. We fix the number of records as 4,194,304 (which is the number of records when the total dataset reaches 400GB with 100kB per record) for the default Zipfian dataset,

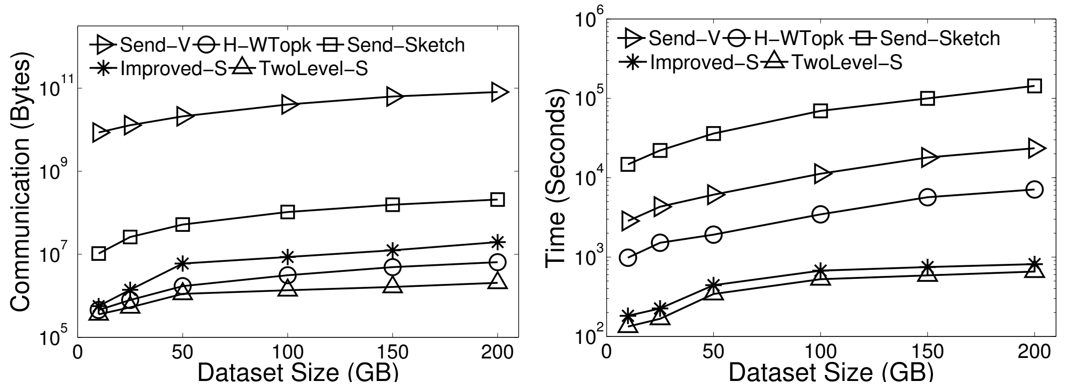
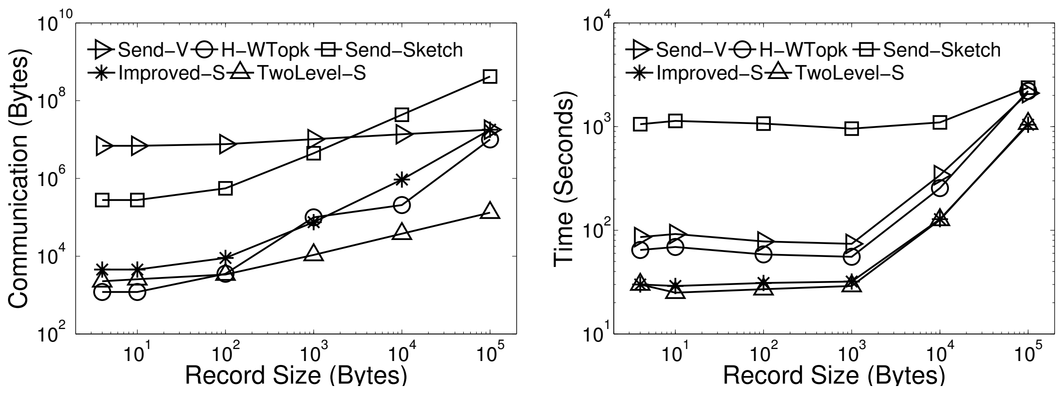


Figure 2.10. Cost analysis vary n: effect on (a) communication and (b) running time.



(a) (b)

Figure 2.11. Cost analysis vary record size: effect on (a) communication and (b) running time.

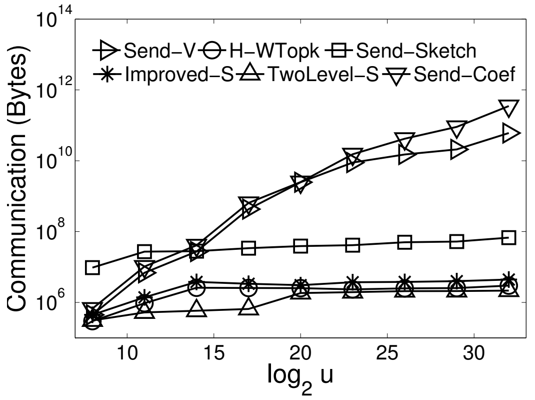
and vary the size of a record from 4 bytes (key only) to 100kB, which corresponds to a dataset size of 16MB to 400GB consisting of 1 and 1600 splits, respectively. We see the communication cost increases for all methods as the record size increases. This makes sense since increasing the number of splits has a negative impact to all of their communication costs. Nevertheless, even with 1600 splits when the record size is 100kB *H-WTopk* still communicates less than *Send-V*; and *TwoLevel-S* still outperforms the other algorithms by orders of magnitude with respect to communication.

The running time of all algorithms also increases as the record size increases, while the total number of records is fixed. This is not surprising due to several factors when the record size increases: 1) all algorithms communicate more data; 2) there are much more splits than the number of slaves in our cluster; 3) the IO cost becomes higher. Note that regardless of the record size *H-WTopk* still performs better than *Send-V*. We also note that the clear winner is *TwoLevel-S* with a running time roughly an order of magnitude better than *Send-V*. Finally, the performance gap between *H-WTopk* and *Send-V*, as well as the gap between *TwoLevel-S* and *Improved-S*, are not as significant as in other experiments. This is mostly due to the small number of records (only 4 million, in contrast to 13.4 billion in the default zipfian dataset and 1.35 billion in the WorldCup dataset) we have to use in this study, which is constrained by the number of records we can accommodate for the maximum record size (100kB), while still keeping the total file size under control (400GB when each record becomes 100kB).

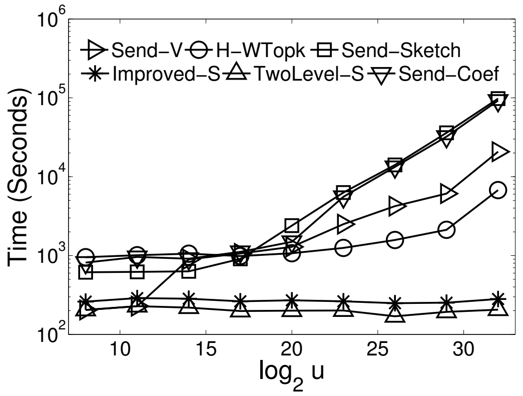
2.5.7 Varying domain size u

We next examine how u affects all algorithms in Figure 2.12. Note as we increase u while keeping n fixed, the tail of the data distribution gets longer while frequent elements get slightly less frequent. Figure 2.12(a) shows that this affects *Send-V*, which is obvious as each local frequency vector $\mathbf{v}_j$ gets more entries. We note that *Send-V* performs better than *Send-Coef* for all tested domains. *Send-Coef* reduces the computational burden at the reducer by performing the wavelet transform in parallel over the local frequency vectors. However, the results indicate that the potential savings from computing the wavelet transform in parallel is canceled out by the increase in communication and computation cost of *Send-Coef* over *Send-V*. The overheads in *Send-Coef* are caused by the fact that the number of local wavelet coefficients grows linearly to the domain size, regardless of the size of each split and how many records a local split contains. Thus, with the increasing domain size, the communication cost and the overall running time of this approach quickly degrade. Indeed, the total nonzero local wavelet coefficients are almost always much greater than the total number of keys in the local frequency vector with a nonzero frequency. Since *Send-V* always results in less communication and computation overheads than *Send-Coef*, we use *Send-V* as our default baseline algorithm for all other experiments.

In terms of running time, larger u makes all methods slower except the sampling-based algorithms. *Send-V*, *Send-Coef*, *H-WTopk*, and *Send-Sketch* all more or less linearly depend on u: *Send-V* and *Send-Coef* are obvious; *H-WTopk* needs $O(u)$ time to compute the wavelet transformation for each $\mathbf{v}_j$; while *Send-Sketch* needs to make $O(u)$ updates to the sketch. The two sampling algorithms are not affected as their sample size is independent of u.



(a)



(b)

Figure 2.12. Cost analysis vary u: effect on (a) communication and (b) running time.

2.5.8 Varying split size β

Figure 2.13 shows the effect of varying the split size β from 64MB to 512MB while keeping n fixed. The number of splits m drops for larger split sizes (varying from 800 to 100 for the 50GB dataset). Hence, the communication cost of all algorithms drop with a larger split size. This is essentially the opposite of Figure 2.10(a) where we increase n (hence m) for a fixed split size. The difference is, for *Send-V*, the communication is not reduced as much, since as the split gets larger, there are more distinct keys in each split, which cancels some benefit of a smaller m.

The running times of all methods reduce slightly as well for larger split size, because *Send-V* has less communication overhead, *H-WTopk* has to perform less local wavelet transformations, and *Send-Sketch* has less updates to the local sketches. For the two sampling algorithms, although their sample size does not depend on m, the communication (hence the cost of the Reducer who needs to process all the incoming messages) reduces as m gets smaller.

All these seem to suggest we should use a split size as large as possible. However, there is a limit on the split size, constrained by the available local disk space (so that a split does not span over multiple machines, which would incur significant communication cost when processing such a split). In addition, larger split sizes reduce the granularity of scheduling and increase the overhead of failure recovery. On our cluster with 16 machines, these issues do not manifest. But on large clusters with thousands of machines, the split size should not be set too large. So the typical split size as recommended by most works in the literature (e.g., [37–39]) is either 128MB or 256MB.

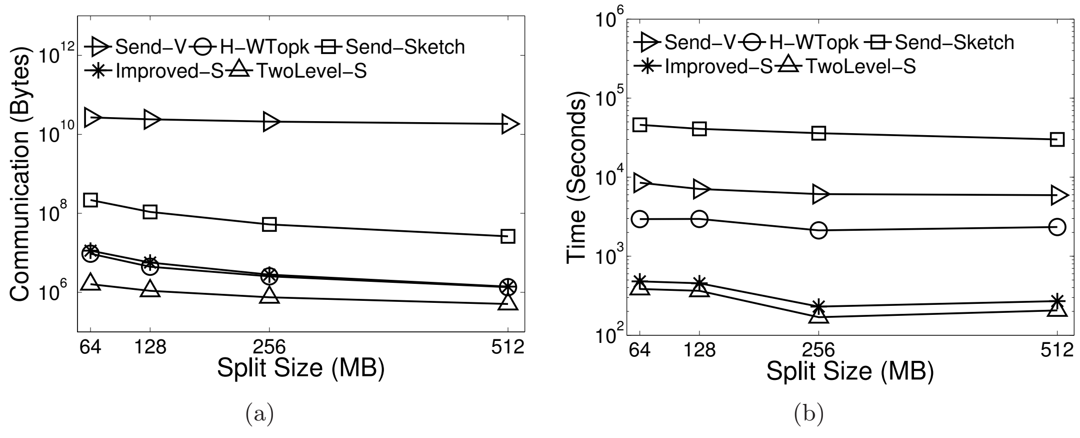


Figure 2.13. Cost analysis vary split size β: effect on (a) communication and (b) running time.

2.5.9 Varying data skewness α

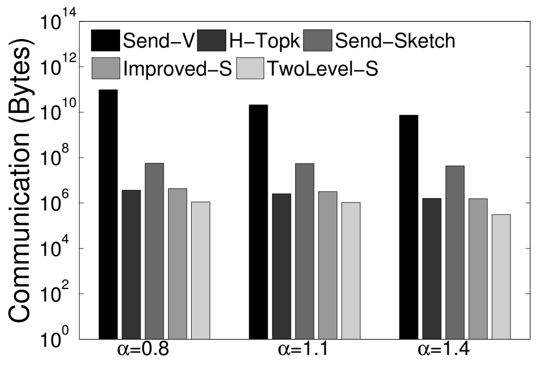
We also study the effect of data skewness α, with α as $0.8, 1.1, 1.4$ and show results in Figure 2.14 and 2.15. When data are less skewed, each split has more distinct key values. As a result, the communication cost of *Send-V* is higher, leading to higher running time. The running time of *Send-Sketch* becomes more expensive as more local sketch updates are necessary. The communication and running time of other methods have little changes. The SSE is analyzed in Figure 2.15. All methods' SSE seem to improve on less skewed data. Nevertheless, *TwoLevel-S* consistently performs the best among all approximation methods.

2.5.10 Varying bandwidth B

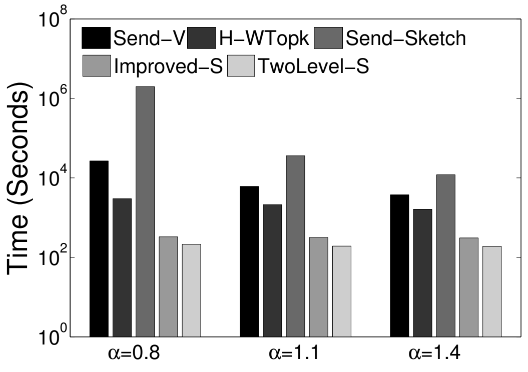
Finally, Figure 2.16 shows the effect the bandwidth B has on the running time of all methods, by varying it from 10% to 100% of the full network bandwidth which is 100Mbps. The communication cost of all algorithms are unaffected by B. *Send-V* enjoys an almost linear reduction in running time when B increases as transmitting data dominates its running time. Other methods see a slight reduction in their respective running times.

2.5.11 WorldCup dataset

Figure 2.17 analyzes the performance of all algorithms on *WorldCup* using default k, ε, β, and B values, in which we attempt to compute the best k-term wavelet representation over the *clientobject* attribute. Notice in Figure 2.17(a) the communication trends for all algorithms are similar to our previous observations. We note the *WorldCup* dataset is approximately 50GB with almost 2^{29} distinct *clientobject* values, which are the defaults used for the Zipfian datasets. *Send-V*'s communication cost is dependent on two primary factors:



(a)



(b)

Figure 2.14. Cost analysis vary skewness α: effect on (a) communication and (b) running time.

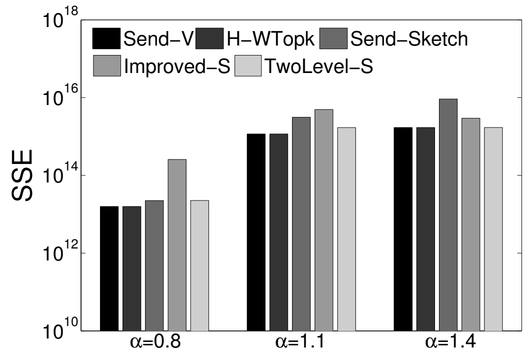


Figure 2.15. Vary α SSE.

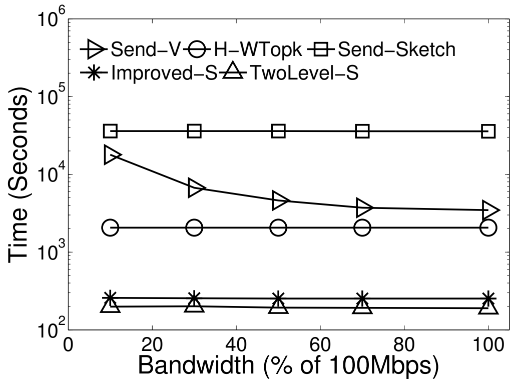
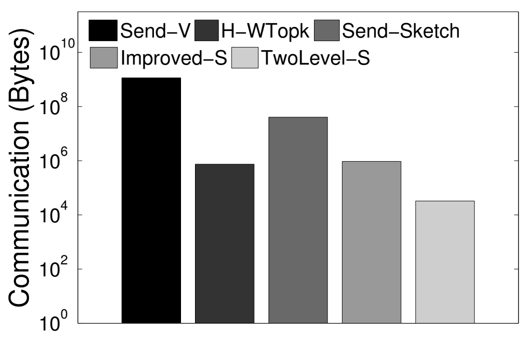
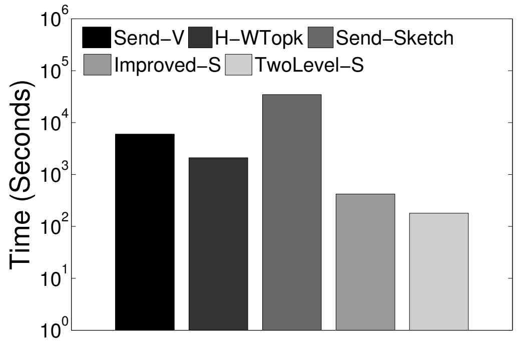


Figure 2.16. Vary B.



(a)



(b)

Figure 2.17. Cost analysis WorldCup dataset: effect on (a) communication and (b) running time.

the skewness of the data and the total number of distinct values. As the data become more skewed, *Send-V* can leverage on the Combine function to reduce communication. However, as we see in Figure 2.17(a), *Send-V* requires roughly the same amount of communication as for the Zipfian datasets. This indicates that by varying α, u, and n for the Zipfian datasets, we can approximate the distribution of real large datasets fairly well.

In Figure 2.17(b), we observe the running times of all approaches on *WorldCup*. *Send-V*'s running time is mainly dependent on its communication cost. The data communicated are about the same as the default Zipfian dataset so it is not surprising *Send-V* preforms similarly on the *WorldCup* dataset. We would like to note *TwoLevel-S* saves almost 2 orders of magnitude and *H-WTopk* saves about 0.5-1 order of magnitude over *Send-V*, indicating our algorithms are effective on large real datasets.

We observe the SSE on *WorldCup* in Figure 2.18. The relative performance of various algorithms is similar to the previously observed trends for Zipfian datasets in Figure 2.15.

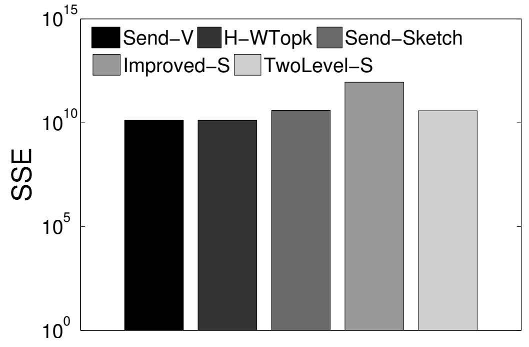


Figure 2.18. SSE on WorldCup.

We also analyze the communication and running time of all algorithms versus the SSE on *WorldCup* in Figure 2.19. The trends are again similar to that in Figure 2.9 for Zipfian datasets. Notice the *Send-Sketch* method achieves a similar SSE, with at least an order of magnitude more communication and orders of magnitude more computation overheads than other methods. We observe that *TwoLevel-S* achieves the best overall SSE to communication cost, requiring approximately an order of magnitude less communication than other methods. In addition, *TwoLevel-S* is also 2-3 times or orders of magnitude faster than other methods to achieve a similar SSE.

2.5.12 Experimental conclusion

These extensive results reach the clear conclusion that *H-WTopk* is the choice if we wish to find exact top-k wavelet coefficients, outperforming the baseline exact method *Send-V* by several orders of magnitude in communication, and 0.5-1 order of magnitude in running time; when approximation is allowed, *TwoLevel-S* is the best method. Not only

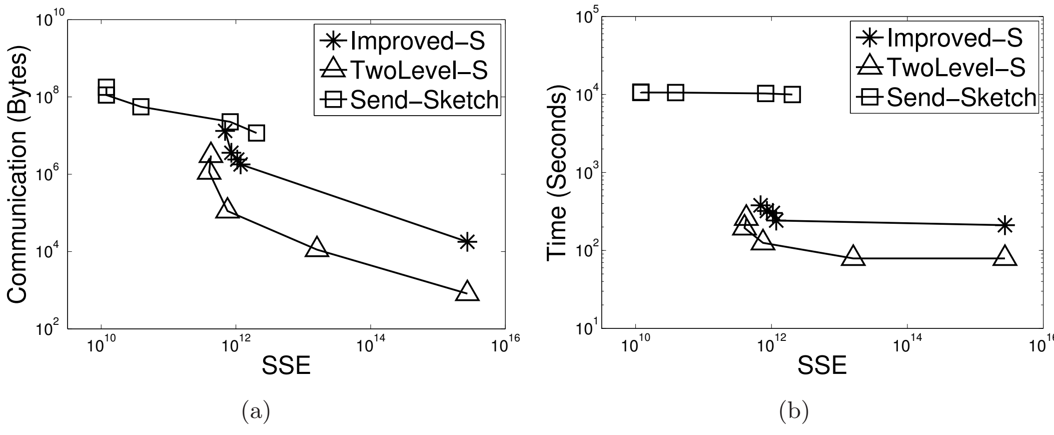


Figure 2.19. SSE on WorldCup: versus (a) communication and (b) running time.

does it offer the cleanest solution, but it also achieves an SSE nearly as good as exact methods for a tiny fraction of their communication cost and running time. In addition, it achieves the best overall communication and running time to achieve an SSE similar to other sampling and sketching techniques. It produces an approximate wavelet histogram of high approximation quality for 200GB data of domain size of 2^{29} in less than 10 minutes with only 2MB communication!

2.6 Related Work

The wavelet histogram and wavelet analysis, introduced to data management for selectivity estimation by Matias et al. [11], has quickly emerged as a widely used tool in databases, data mining, and data analysis [22–25]. Matias et al. have also studied how to dynamically maintain the wavelet histograms under updates [12]. Gilbert et al. [13] extended the construction of the wavelet histogram to streaming data, using the AMS sketch [35]. Cormode et al. [14] then improved the efficiency of the sketch with the Group-Count Sketch (GCS).

Many types of histograms exist. Poosala et al. [18] presented an excellent discussion on the properties of various histograms. How to efficiently build other types of histograms for large data in MapReduce is an intriguing open problem we plan to investigate.

Since its introduction [19], MapReduce has quickly become a primary framework for processing massive data. It represents the trend of going towards parallel and distributed processing on shared-nothing commodity clusters [20, 40, 41]. Significant effort has been devoted to improving the efficiency, the functionality and query processing in MapReduce, e.g., Amazon EC2 [42], HadoopDB [43], Hadoop++ [44], Hadoop [7], MapReduce Online [45], and many others [46]. To the best of our knowledge, efficient construction of wavelet histograms in MapReduce has not been studied.

Our work is also related to finding distributed top-k and frequent items. The best exact method for distributed top-k is TPUT [27]. However, it (and other methods, e.g., [28]) does not support finding the aggregates with the largest absolute values over positive and negative value sets. Our exact algorithm shares the basic principle in distributed query processing, however, comes with novel designs in order to work for wavelets in MapReduce. The approximate distributed top-k query has been studied in [28] and many others. However, they also only support nonnegative scores and require multiple rounds, which introduce considerable overhead in the MapReduce framework. As such, we did not attempt to adapt them as approximation methods. Instead, for approximation methods, we focus on algorithms that require only one round of MapReduce. The most related works are methods for finding heavy hitters from distributed datasets [33]. However, they are not tailored for the MapReduce environment, and use complicated heuristics that are hard to implement efficiently in Hadoop. There is also a study on finding the top-k largest valued items in MapReduce [47], where each item has a *single total score*, which is clearly different from (and does not help) our case.

2.7 Closing Remarks

As the *size* of data continues to explode, they are increasingly being stored and processed in parallel and distributed platforms, such as MapReduce clusters, and this chapter studies how to summarize these massive data using wavelet histograms. We designed both exact and approximation methods in MapReduce, which significantly outperform the straightforward adaptations of existing methods to MapReduce. Our methods are also easy to implement, in particular the two-level sampling method, making them appealing in practice.

Data summarization is an important technique for analyzing large relational data. The wavelet histogram is merely one representative, and there are many other types of summaries we may consider, such as other kinds of histograms (e.g., the V-optimal histogram [48]), and various sketches and synopses. Another open problem is how to incrementally maintain the summary when the data stored in the MapReduce cluster are being updated. Finally, data summarization in MapReduce, and any parallel and distributed platform, is also an intellectually challenging problem, requiring a good blend of algorithmic techniques and system building.

CHAPTER 3

RANKING LARGE TEMPORAL DATA

3.1 Introduction

In this chapter, we turn our focus to the emerging challenge of increasingly *complex structure and rich semantics* of data; in particular, we focus on temporal data and how to efficiently query and summarize it using the *ranking operator*.

Temporal data have important applications in numerous domains, such as in the financial market, in scientific applications, and in the biomedical field. Despite the extensive literature on storing, processing, and querying temporal data, and the importance of ranking (which is considered as a first-class citizen in database systems [50]), ranking temporal data has not been studied until recently [51]. However, only the *instant* top-k queries on temporal data were studied in [51], where objects with the k highest scores at a query time instance t are to be retrieved; it was denoted as the top-$k(t)$ query in [51]. The instant top-k definition clearly comes with obvious limitations (sensitivity to outliers, difficulty in choosing a meaningful single query time t). A much more flexible and general ranking operation is to rank temporal objects based on the aggregation of their scores in a query interval, which we dub the *aggregate* top-k query on temporal data, or top-$k(t_1, t_2, \sigma)$ for an interval $[t_1, t_2]$ and an aggregation function σ. For example, return the top-10 weather stations having the highest average temperature from 10/01/2010 to 10/07/2010; find the top-20 stocks having the largest total transaction volumes from 02/05/2011 to 02/07/2011.

Clearly, the instant top-k query is a special case of the aggregate top-k query (when $t_1 = t_2$). The work in [51] shows that even the instant top-k query is hard!

3.1.1 Problem formulation

In temporal data, each object has at least one score attribute A whose value changes over time, e.g., the temperature readings in a sensor database. An example of real temperature data from the MesoWest project appears in Figure 3.1. In general, we can represent the

*The work in this chapter appears in [49] ©2012 VLDB Endowment. Reprinted by Permission.

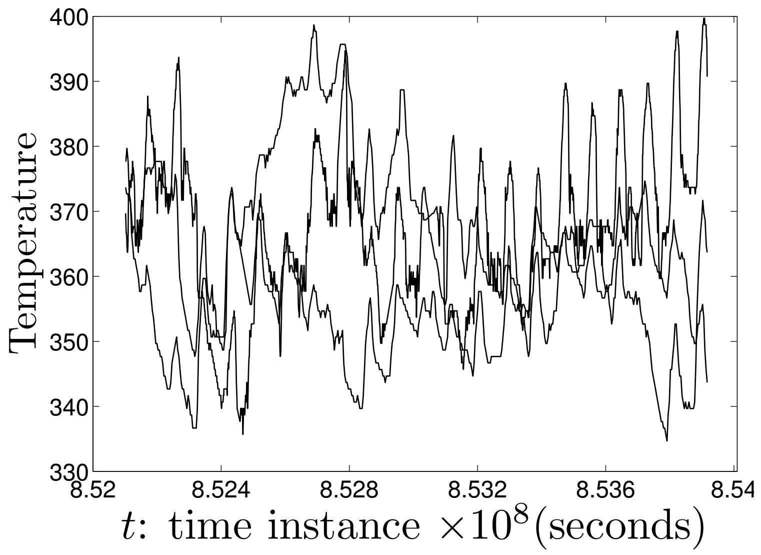


Figure 3.1. MesoWest data.

score attribute A of an object as an arbitrary function $f : \mathbb{R} \rightarrow \mathbb{R}$ (time to score), but for arbitrary temporal data, f could be expensive to describe and process. In practice, applications often approximate f using a piecewise linear function g [52–55]. The problem of approximating an arbitrary function f by a piecewise linear function g has been extensively studied (see [52–54,56] and references therein). Key observations are: 1) more segments in g lead to better approximation quality, but also are more expensive to represent; 2) adaptive methods, by allocating more segments to regions of high volatility and less to smoother regions, are better than nonadaptive methods with a fixed segmentation interval.

In this chapter, for the ease of discussion and illustration, we focus on temporal data represented by piecewise linear functions. Nevertheless, our results can be extended to other representations of time series data, as we will discuss in Section 3.4. Note that a lot of work in processing temporal data also assumes the use of piecewise linear functions as the main representation of the temporal data [52–55,57], including the prior work on the instant top-k queries in temporal data [51]. That said, how to approximate f with g is beyond the scope of this chapter, and we assume that the data have already been converted to a piecewise linear representation by *any* segmentation method. In particular, we require *neither* that they have the same number of segments *nor* that they have the aligned starting/ending time instances for segments from different functions. It is possible the data are collected from a variety of sources, where each source may apply different preprocessing modules on its respective data.

That said, formally, there are m objects in a temporal database; the ith object o_i is represented by a piecewise linear function g_i with n_i number of (linear line) segments. There are a total of $N = \sum_{i=1}^{m} n_i$ segments from all objects. The temporal range of any object is in $[0, T]$. An aggregate top-k query is denoted as top-$k(t_1, t_2, \sigma)$ for some aggregation function σ, which is to retrieve the k objects with the k highest aggregate scores in the range $[t_1, t_2]$, denoted as an ordered set $\mathcal{A}(k, t_1, t_2)$ (simply $\mathcal{A}$ when the context is clear). The aggregate score of o_i in $[t_1, t_2]$ is defined as $\sigma(g_i(t_1, t_2))$, or simply $\sigma_i(t_1, t_2)$, where $g_i(t_1, t_2)$ denotes the set of all possible values of function g_i evaluated at every time instance in $[t_1, t_2]$ (clearly an infinite set for continuous time domain). For example, when $\sigma = \text{sum}$, the aggregate score for o_i in $[t_1, t_2]$ is $\int_{t_1}^{t_2} g_i(t)dt$. An example sum top-2 query is shown in Figure 3.2. For ease of illustration, we assume nonnegative scores by default. This restriction is removed in Section 3.4. We also assume a max possible value $k_{\max}$ for k.

3.1.2 Our contributions

A straightforward observation is that a solution to the instant top-k query cannot be directly applied to solve the aggregate top-k query since: 1) the temporal dimension can be continuous; and 2) an object might not be in the top-k set for any top-$k(t)$ query for $t \in [t_1, t_2]$, but still belong to $\mathcal{A}(k, t_1, t_2)$ (for example, $\mathcal{A}(1, t_2, t_3)$ in Figure 3.2 is $\{o_1\}$, even though o_1 is never a top-1(t) object for any $t \in [t_2, t_3]$). Hence, the trivial solution (denoted as EXACT1) is for each query to compute $\sigma_i(t_1, t_2)$ of every object and insert them into a priority queue of size k, which takes $O(m(N + \log k))$ time per query and is clearly not scalable for large datasets (although our implementation slightly improves this query time, as described in Section 3.2). Our goal is then to design IO and computation efficient algorithms which can outperform the trivial solution and work well regardless

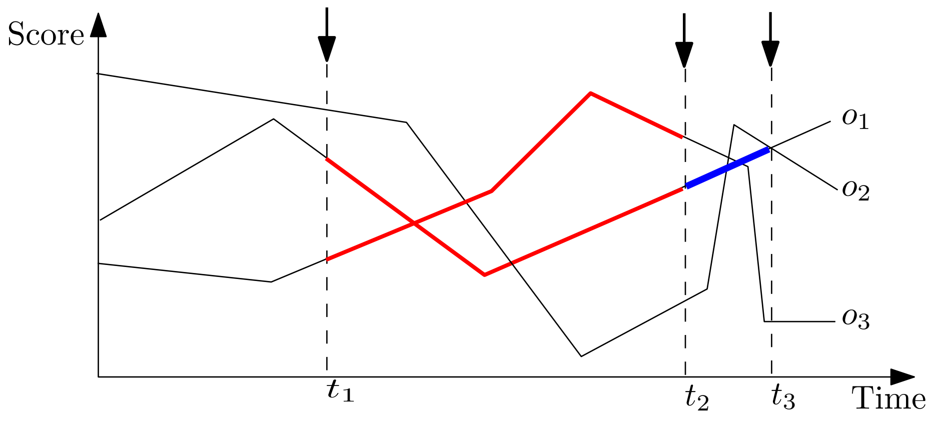


Figure 3.2. A top-2(t_1, t_2, sum) query example with answer $\{o_3, o_1\}$

of whether data fit in main memory or not. A design principle we have followed is to leverage on existing indexing structures whenever possible (so these algorithms can be easily adopted in practice). Our work focuses specifically on $\sigma = \text{sum}$, and we make the following contributions:

- We design a novel exact method in Section 3.2, based on using a single interval tree (Exact3).
- We present two approximate methods (and several variants) in Section 3.3. Each offers an approximation $\widetilde{\sigma}_i(t_1, t_2)$ on the aggregate score $\sigma_i(t_1, t_2)$ for objects in any query interval. We say $\tilde{X}$ is an (ε, α)-approximation of X if $X/\alpha - \varepsilon M \leq \tilde{X} \leq X + \varepsilon M$ for user-defined parameters $\alpha \geq 1$, $\varepsilon > 0$ and where $M = \sum_{i=1}^{m} \sigma_i(0, T)$. Now, for $i \in [1, m], [t_1, t_2] \subseteq [0, T]$, the Appx1 method guarantees that $\widetilde{\sigma}_i(t_1, t_2)$ is an $(\varepsilon, 1)$-approximation of $\sigma_i(t_1, t_2)$, and the Appx2 method guarantees $\widetilde{\sigma}_i(t_1, t_2)$ is an $(\varepsilon, 2\log(1/\varepsilon))$-approximation of $\sigma_i(t_1, t_2)$. We show an (ε, α)-approximation on $\sigma_i(t_1, t_2)$ implies an approximation $\widetilde{\mathcal{A}}(k, t_1, t_2)$ of $\mathcal{A}(k, t_1, t_2)$ such that the aggregate score of the jth ranked $(1 \leq j \leq k)$ object in $\widetilde{\mathcal{A}}(k, t_1, t_2)$ is always an (ε, α)-approximation of the aggregate score of the jth ranked object in $\mathcal{A}(k, t_1, t_2)$.
- We extend our results to general functions f for temporal data, other possible aggregates, negative scores, and deal with updates in Section 3.4.
- We show extensive experiments on massive real data sets in Section 3.5. The results clearly demonstrate the efficiency, effectiveness, and scalability of our methods compared to the baseline. Our approximate methods are especially appealing when approximation is admissible, given their better query costs than exact methods and high-quality approximations.

We survey the related work in Section 3.6, and conclude in Section 3.7. Table 3.1 summarizes our notations. A summary of the upper bounds on the preprocessing cost, the index size, the query cost, the update cost, and the approximation guarantee of all methods appears in Table 3.2; note for simplicity, $\log_B k_{\max}$ terms are absorbed in $O(\cdot)$ notation in Table 3.2.

Table 3.1. Frequently used notations.

Symbol	Description
$\mathcal{A}(k, t_1, t_2)$	ordered top-k objects for top-$k(t_1, t_2, \sigma)$.
$\widetilde{\mathcal{A}}(k, t_1, t_2)$	an approximation of $\mathcal{A}(k, t_1, t_2)$.
$\mathcal{A}(j), \widetilde{\mathcal{A}}(j)$	the jth ranked object in $\mathcal{A}$ or $\widetilde{\mathcal{A}}$.
B	block size.
$\mathcal{B}$	set of breakpoints ($\mathcal{B}_1$ and $\mathcal{B}_2$ are special cases).
$\mathcal{B}(t)$	smallest breakpoint in $\mathcal{B}$ larger than t.
g_i	the piecewise linear function of o_i.
$g_{i,j}$	the jth line segment in g_i, $j \in [1, n_i]$.
$g_i(t_1, t_2)$	the set of all possible values of g_i in $[t_1, t_2]$.
$k_{\max}$	the maximum k value for user queries.
$\ell(t)$	the value of a line segment ℓ at time instance t.
m	total number of objects.
M	$M = \sum_{i=1}^{m} \sigma_i(0, T)$.
n_i	number of line segments in g_i.
n, n_{avg}	$\max\{n_1, n_2, \ldots, n_m\}$, $\text{avg}\{n_1, n_2, \ldots, n_m\}$
N	number of line segments of all objects.
o_i	the ith object in the database.
q_i	number of segments in g_i overlapping $[t_1, t_2]$.
r	number of breakpoints in $\mathcal{B}$, bounded $O(1/\varepsilon)$.
$(t_{i,j}, v_{i,j})$	jth end-point of segments in g_i, $j \in [0, n_i]$.
$\sigma_i(t_1, t_2)$	aggregate score of o_i in an interval $[t_1, t_2]$.
$\widetilde{\sigma}_i(t_1, t_2)$	an approximation of $\sigma_i(t_1, t_2)$.
$[0, T]$	the temporal domain of all objects.

Table 3.2. IO costs, with block size B

method	index size	construction cost	query cost
Exact1	$O(\frac{N}{B})$	$O(\frac{N}{B}\log_B N)$	$O(\log_B N + \frac{\sum_{i=1}^{m} q_i}{B})$
Exact2	$O(\frac{N}{B})$	$O(\sum_{i=1}^{m} \frac{n_i}{B}\log_B n_i)$	$O(\sum_{i=1}^{m}\log_B n_i)$
Exact3	$O(\frac{N}{B})$	$O(\frac{N}{B}\log_B N)$	$O(\log_B N + \frac{m}{B})$
Appx1	$O(\frac{r^2}{B}k_{\max})$	$O(\frac{N}{B}(\log_B N + r))$	$O(\frac{k}{B} + \log_B r)$
Appx2	$O(\frac{r}{B}k_{\max})$	$O(\frac{N}{B}(\log_B N + \log r))$	$O(k\log r)$

method	update cost	approximation
Exact1	$O(\log_B N)$	$(0, 1)$
Exact2	$O(\log_B n)$	$(0, 1)$
Exact3	$O(\log_B N)$	$(0, 1)$
Appx1	$O(\frac{1}{B}(\log_B N + r))$	$(\varepsilon, 1)$
Appx2	$O(\frac{1}{B}(\log_B N + \log r))$	$(\varepsilon, 2\log r)$

3.2 Exact Methods

As explained in Section 3.1, a trivial exact solution Exact1 is to find the aggregate score of each object in the query interval and insert them into a priority queue of size k. We can improve this approach by indexing line segments from all objects with a B+-tree, where the key for a data entry e is the value of the time-instance for the left-end point of a line segment ℓ, and the value of e is just ℓ. Given a query interval $[t_1, t_2]$, this B+-tree allows us to find the first leaf node with segments that contain t_1 in $O(\log_B N)$ IOs. Assuming the leaf nodes are linked with forward pointers in the B+-tree, a forward scan of the leaf nodes (till t_2) then can retrieve all line segments whose temporal dimensions overlap with $[t_1, t_2]$ (either fully or partially). In this process, we simply maintain m running sums, one per object in the database. Suppose the ith running sum of object o_i is s_i and it is initialized with the value 0. Given a line segment ℓ defined by $(t_{i,j}, v_{i,j})$ and $(t_{i,j+1}, v_{i,j+1})$ from o_i (see an example in Figure 3.3), we define an interval $I = [t_1, t_2] \cap [t_{i,j}, t_{i,j+1}]$, let $t_L = \max\{t_1, t_{i,j}\}$ and $t_R = \min\{t_2, t_{i,j+1}\}$, and update $s_i = s_i + \sigma_i(I)$, where

$$\sigma_i(I) = \begin{cases} 0, \text{ if } t_2 < t_L \text{ or } t_1 > t_R; \\ \frac{1}{2}(t_R - t_L)(\ell(t_R) + \ell(t_L)), \quad \text{else.} \end{cases} \tag{3.1}$$

Note that $\ell(t)$ is the value of the line segment ℓ at time t. Note that if we follow the sequential scan process described above, we will only deal with line segments that do overlap with the temporal range $[t_1, t_2]$, in which the increment to s_i corresponds to the second case in (3.1). It is essentially an integral from $t_L = \max\{t_1, t_{i,j}\}$ to $t_R = \min\{t_2, t_{i,j+1}\}$ w.r.t. ℓ, i.e., $\int_{t_L}^{t_R} \ell(t)dt$. This range $[t_L, t_R]$ of ℓ also defines a trapezoid; hence, it is equal to the area of this trapezoid, which yields the formula in (3.1).

When we have scanned all line segments up to t_2 from the B+-tree, we stop and assign $\sigma_i(t_1, t_2) = s_i$ for $i = 1$ to m. Finally, we insert $(i, \sigma_i(t_1, t_2))$, for $i = 1$ to m, into a priority

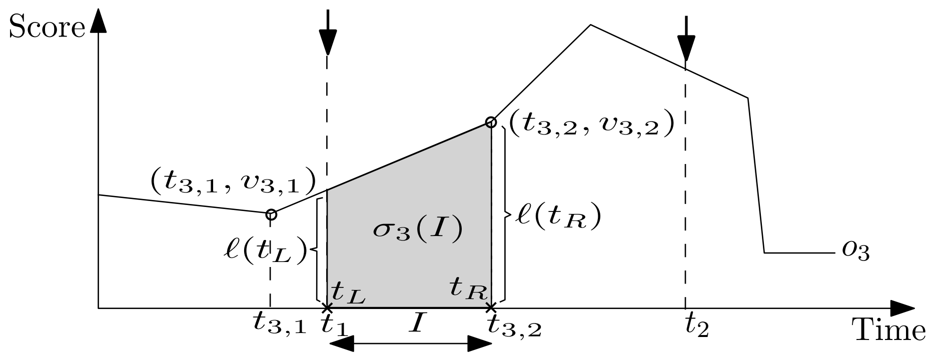


Figure 3.3. Compute $\sigma_i([t_1, t_2] \cap [t_{i,j}, t_{i,j+1}])$.

queue of size k sorted in the descending order of $\sigma_i(t_1, t_2)$. The answer $\mathcal{A}(k, t_1, t_2)$ is the (ordered) object ids in this queue when the last pair $(m, \sigma_m(t_1, t_2))$ has been processed.

This method EXACT1 has a cost of $O((N/B)\log_B N)$ IOs for building the B+-tree, an index size of $O(N/B)$ blocks, and a query cost of $O(\log_B N + \sum_{i=1}^{m} q_i/B + (m/B)\log_B k)$ IOs where q_i is the number of line segments from o_i overlapping with the temporal range $[t_1, t_2]$ of a query q=top-$k(t_1, t_2, \text{sum})$. In the worst case, $q_i = n_i$ for each i, then the query cost becomes $O(N/B)$!

3.2.1 A forest of B+-trees

EXACT1 becomes quite expensive when there are a lot of line segments in $[t_1, t_2]$, and its asymptotic query cost is actually $O(N/B)$ IOs, which is clearly nonscalable. The bottleneck of EXACT1 is the computation of the aggregate score of each object. One straightforward idea to improve the aggregate score computation is to leverage on precomputed prefix-sums [58]. We apply the notion of prefix-sums to continuous temporal data by precomputing the aggregate scores of some selected intervals in each object; this preprocessing helps reduce the cost of computing the aggregate score for an arbitrary interval in an object. Let $(t_{i,j}, v_{i,j})$ be the jth end-point of segments in g_i, where $j \in \{0, \dots, n_i\}$; clearly, the jth segment in g_i is then defined by $((t_{i,j-1}, v_{i,j-1}), (t_{i,j}, v_{i,j}))$ for $j \in \{1, \dots, n_i\}$, which we denote as $g_{i,j}$. Then, define intervals $I_{i,\ell} = [t_{i,0}, t_{i,\ell}]$ for $\ell = 1, \dots, n_i$, and compute the aggregate score $\sigma_i(I_{i,\ell})$ for each.

Once we have $(I_{i,\ell}, \sigma_i(I_{i,\ell}))$s, we build a B+-tree to index them. Specifically, we make a leaf-level data entry $e_{i,\ell}$ for $(I_{i,\ell}, \sigma_i(I_{i,\ell}))$, where the key in $e_{i,\ell}$ is $t_{i,\ell}$ (the right end-point of $I_{i,\ell}$), and the value of $e_{i,\ell}$ includes both $g_{i,\ell}$ and $\sigma_i(I_{i,\ell})$. Given $\{e_{i,1}, \dots, e_{i,n_i}\}$ for o_i, we bulk-load a B+-tree T_i using them as the leaf-level data entries (see Figure 3.4 for an example).

We do this for each object, resulting in m B+-trees. Given T_i, we can compute $g_i(t_1, t_2)$ for any interval $[t_1, t_2]$ efficiently. We first find the data entry $e_{i,L}$ such that its key value $t_{i,L}$ is the first succeeding key value of t_1; we then find the data entry $e_{i,R}$ such that its key value $t_{i,R}$ is the first succeeding key value of t_2. Next, we can calculate $\sigma_i(t_1, t_{i,L})$ using $g_{i,L}$ (stored in $e_{i,L}$), and $\sigma_i(t_2, t_{i,R})$ using $g_{i,R}$ (stored in $e_{i,R}$), simply based on (3.1). Finally,

$$\sigma_i(t_1, t_2) = \sigma_i(I_{i,R}) - \sigma_i(I_{i,L}) + \sigma_i(t_1, t_{i,L}) - \sigma_i(t_2, t_{i,R}), \tag{3.2}$$

where $\sigma_i(I_{i,R})$, $\sigma_i(I_{i,L})$ are available in $e_{i,R}$, $e_{i,L}$, respectively. Figure 3.4 also gives a query example using o_3.

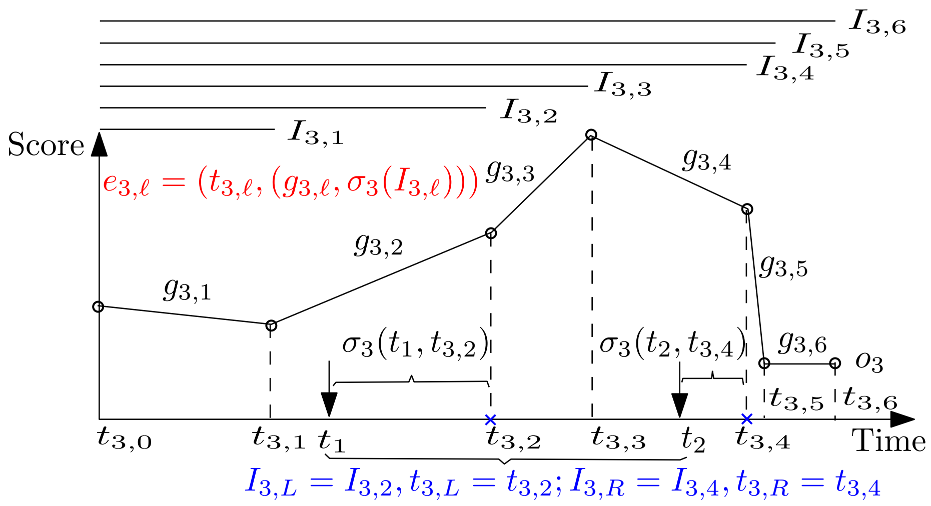


Figure 3.4. The method Exact2.

Once all $\sigma_i(t_1, t_2)$s are computed for $i = 1, \ldots, m$, the last step is the same as that in Exact1.

We denote this method as Exact2. Finding $e_{i,L}$ and $e_{i,R}$ from T_i takes only $\log_B n_i$ cost, and calculating (3.2) takes $O(1)$ time. Hence, its query cost is $O(\sum_{i=1}^{m} \log_B n_i + m/B \log_B k)$ IOs. The index size of this method is the size of all B+-trees, where T_is size is linear to n_i; so the total size is $O(N/B)$ blocks. Note that computing $\{\sigma_i(I_{i,1}), \ldots, \sigma_i(I_{i,n_i})\}$ can be easily done in $O(n_i/B)$ IOs, by sweeping through the line segments in g_i sequentially from left to right, and using (3.1) incrementally (i.e., computing $\sigma_i(I_{i,\ell+1})$ by initializing its value to $\sigma_i(I_{i,\ell})$). Hence, the construction cost is dominated by building each tree T_i with cost $O((n_i/B)\log_B n_i)$. The total construction cost is $O(\sum_{i=1}^{m}(n_i/B)\log_B n_i)$.

3.2.2 Using one interval tree

When m is large (as is the case for the real data sets we explore in Section 3.5), querying m B+-trees becomes very expensive, partly due to the overhead of opening and closing m disk files storing these B+-trees. Hence, an important improvement is to somehow index the data entries from all m B+-trees in a single disk-based data structure.

Consider any object o_i, and let intervals $I_{i,1}, \ldots, I_{i,n_i}$ be the same as that in Exact2, where $I_{i,\ell} = [t_{i,0}, t_{i,\ell}]$. Furthermore, we define intervals $I^-_{i,1}, \ldots, I^-_{i,n_i}$, such that $I^-_{i,\ell} = I_{i,\ell} - I_{i,\ell-1}$ (let $I_{i,0} = [t_{i,0}, t_{i,0}]$), i.e., $I^-_{i,\ell} = [t_{i,\ell-1}, t_{i,\ell}]$.

We define data entry $e_{i,\ell}$ such that its key is $I^-_{i,\ell}$, and its value is $(g_{i,\ell}, \sigma_i(I_{i,\ell}))$, for $\ell = 1, \ldots, n_i$. An object o_i yields n_i such entries; an example is shown in Figure 3.5. When we collect all entries from all objects, we end up with N entries in total. We denote these

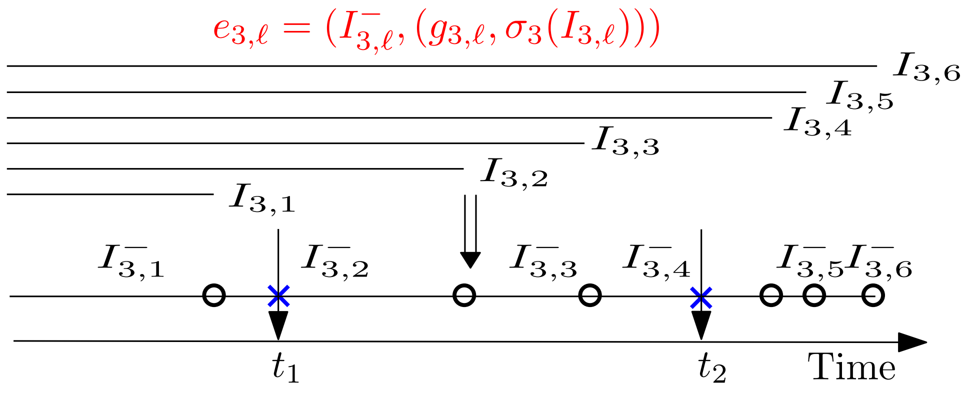


Figure 3.5. The method Exact3.

data entries as a set I^-; and it is important to note that the key value of each data entry in I^- is an interval. Hence, we can index I^- using a disk-based interval tree S [59–61].

Given this interval tree S, computing $\sigma_i(t_1, t_2)$ can now be reduced to two stabbing queries, using t_1 and t_2, respectively, which return the entries in S whose key values (intervals in I^-) contain t_1 or t_2, respectively. Note that each such stabbing query returns exactly m entries, one from each object o_i. This is because: 1) any two intervals $I^-_{i,x}$, $I^-_{i,y}$ for $x \neq y$ from o_i satisfies $I^-_{i,x} \cap I^-_{i,y} = \emptyset$; 2) and $I^-_{i,1} \cup I^-_{i,2} \cup \cdots \cup I^-_{i,n_i} = [0, T]$.

Now, suppose the stabbing query of t_1 returns an entry $e_{i,L}$ from o_i in S, and the stabbing query of t_2 returns an entry $e_{i,R}$ from o_i in S. It is easy to see that we can calculate $\sigma_i(t_1, t_2)$ just as (3.2) does in Exact2 (see Figure 3.5). Note that using only these two stabbing queries is sufficient to compute all $\sigma_i(t_1, t_2)$s for $i = 1, \ldots, m$.

Given N data entries, the external interval tree has a linear size, $O(N/B)$ blocks, and takes $O((N/B)\log_B N)$ IOs to build [60] (building entries $\{e_{i,1}, \ldots, e_{i,n_i}\}$ for o_i takes only $O(n_i/B)$ cost). The two stabbing queries take $O(\log_B N + m/B)$ IOs [60]; hence, the total query cost, by adding the cost of inserting $\sigma_i(t_1, t_2)$s into a priority queue of size k, is $O(\log_B N + (m/B)\log_B k)$. We denote this method Exact3.

3.2.3 Remarks

One technique we do not consider is indexing temporal data with R-trees to solve aggregate top-k queries. R-trees constructed over temporal data have been shown to perform orders of magnitude worse than other indexing techniques for answering instant top-k queries, even when branch-and-bound methods are used [51]. Given this fact, we do not attempt to extend the use of R-trees to solve the harder aggregate top-k query.

Temporal aggregation with range predicates has been studied in the classic work [62,63], however, with completely different objectives. Firstly, they dealt with multiversioned keys instead of time-series data, i.e., each key is alive with a constant value during a time period before it gets deleted. One can certainly model these keys as temporal objects with constant functions following our model (or even piecewise constant functions to model also updates to keys, instead of only insertions and deletions of keys). But more importantly, their definitions of the aggregation [62, 63] are fundamentally different from ours. The goal in [63] is to compute the sum of key values alive at a time instance, or alive at a time interval intersecting a query interval. The work in [62] extends [63] by allowing a range predicate on the key dimension as well, i.e., its goal is to compute the sum of key values that 1) are alive at a time instance, or alive at a time interval intersecting a query interval; 2) and are within a specified query range in the key dimension.

Clearly, these aggregations [62,63] are different from ours. They want to compute *a single aggregation* of all keys that “fall within” (are alive in) a two-dimensional query rectangle; while our goal is to compute the aggregate score values of many individual objects over a time interval (then rank objects based on these aggregations).

Zhang et al. [62] also extended their investigation to compute the sum of weighted key values, where each key value (that is alive in a two-dimensional query rectangle) is multiplied by a weight proportional to how long it is alive on the time dimension within the query interval. This weighted key value definition will be the same as our aggregation definition if an object's score is a constant in the query interval. They also claimed that their solutions can still work when the key value is not a constant, but a function with certain types of constraints. Nevertheless, even in these cases, their goal is to compute *a single sum over all weighted key values* for an arbitrary two-dimensional query rectangle, rather than each individual weighted key value over a time interval. Constructing m such structures, a separate one for *each* of the m objects in our problem, and only allowing an unbounded key domain can be seen as similar to our EXACT2 method, which on large data corpuses is the least efficient technique we consider. These fundamental differences make these works almost irrelevant in providing helpful insights for solving our temporal aggregation problems.

3.3 Approximate Methods

The exact approaches require explicit computation of $\sigma_i(t_1, t_2)$ for each of m objects, and we manage to reduce the IO cost of this from roughly N/B to m to m/B. Yet, on real datasets when m is quite large, this can still be infeasible for fast queries. Hence, we now study approximate methods that allow us to remove this requirement of computing all m

aggregates, while still allowing *any* query $[t_1, t_2]$ over the continuous time domain.

Our approximate methods focus on constructing a set of breakpoints $\mathcal{B} = \{b_1, b_2, \ldots, b_r\}$, $b_i \in [0, T]$ in the time domain, and snapping queries to align with these breakpoints. We prove the returned value $\tilde{\sigma}_i(t_1, t_2)$ for any curve $(\varepsilon, 1)$-approximates $\sigma_i(t_1, t_2)$. The size of the breakpoints and time for queries will be independent of the total number of segments N or objects m.

In this section, we devise two methods for constructing r breakpoints, BreakPoints1 and BreakPoints2. The first method BreakPoints1 guarantees $r = \Theta(1/\varepsilon)$ and is fairly straightforward to construct. The second method requires more advanced techniques to construct efficiently and guarantees $r = O(1/\varepsilon)$, but can be much smaller in practice.

Then given a set of breakpoints, we present two ways to answer approximate queries on them: Query1 and Query2. The first approach Query1 constructs $O(r^2)$ intervals, and uses a two-level B+-tree to retrieve the associated top k objects list from the one interval snapped to by the query. The second approach Query2 only builds $O(r)$ intervals and their associated $k_{\max}$ top objects, and on a query narrows the list of possible top k-objects to a reduced set of $O(k \log r)$ objects. Figure 3.6 shows an outline of these methods.

We define the following approximation metrics.

Definition 3.1. *G is an (ε, α)-approximation algorithm of the aggregate scores if for any $i \in [1, m], [t_1, t_2] \subseteq [0, T]$, G returns $\widetilde{\sigma}_i(t_1, t_2)$ such that $\sigma_i(t_1, t_2)/\alpha - \varepsilon M \leq \widetilde{\sigma}_i(t_1, t_2) \leq \sigma_i(t_1, t_2) + \varepsilon M$, for user-defined parameters $\alpha \geq 1$, $\varepsilon > 0$.*

Definition 3.2. *For $\mathcal{A}(k, t_1, t_2)$ (or $\widetilde{\mathcal{A}}(k, t_1, t_2)$), let $\mathcal{A}(j)$ (or $\widetilde{\mathcal{A}}(j)$) be the jth ranked object in $\mathcal{A}$ (or $\widetilde{\mathcal{A}}$). R is an (ε, α)-approximation algorithm of top-$k(t_1, t_2, \sigma)$ queries if for any $k \in [1, k_{\max}], [t_1, t_2] \subseteq [0, T]$, R returns $\widetilde{\mathcal{A}}(k, t_1, t_2)$ and $\widetilde{\sigma}_{\widetilde{\mathcal{A}}(j)}(t_1, t_2)$ for $j \in [1, k]$, s.t. $\widetilde{\sigma}_{\widetilde{\mathcal{A}}(j)}(t_1, t_2)$ is an (ε, α)-approximation of $\sigma_{\widetilde{\mathcal{A}}(j)}(t_1, t_2)$ and $\sigma_{\mathcal{A}(j)}(t_1, t_2)$.*

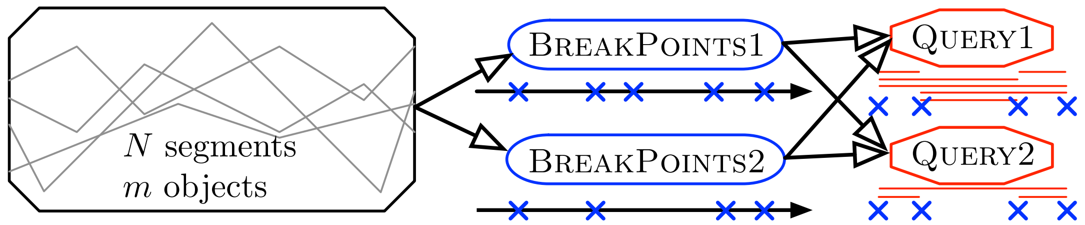


Figure 3.6. Outline of approximate methods.

Definition 3.2 states that $\widetilde{\mathcal{A}}$ will be a good approximation of $\mathcal{A}$ if (ε, α) are small, since at each rank, the two objects from $\widetilde{\mathcal{A}}$ and $\mathcal{A}$, respectively, will have really close aggregate scores. This implies that the exact ranking order in $\mathcal{A}$ will be preserved well by $\widetilde{\mathcal{A}}$ unless many objects have very close (smaller than the gap defined by (ε, α)) aggregate scores on some query interval; and this is unlikely in real datasets when users choose small values of (ε, α).

Lemma 3.1. *An algorithm G that satisfies Definition 3.1 implies an algorithm R that satisfies Definition 3.2.*

Proof of Lemma 3.1: Algorithm G creates $\widetilde{\mathcal{A}}(k, t_1, t_2)$ by finding the top k objects and approximate scores ranked by $\widetilde{\sigma}_i(t_1, t_2)$. By the definition of G, $\widetilde{\sigma}_{\widetilde{\mathcal{A}}(j)}(t_1, t_2)$ is an (ε, α)-approximation of $\sigma_{\widetilde{\mathcal{A}}(j)}(t_1, t_2)$. To see $\widetilde{\sigma}_{\widetilde{\mathcal{A}}(j)}(t_1, t_2)$ is an (ε, α)-approximation of $\sigma_{\mathcal{A}(j)}(t_1, t_2)$, note that all j objects $\mathcal{A}(j')$ for $j' \in [0, j]$ satisfy that $\widetilde{\sigma}_{\mathcal{A}(j')}(t_1, t_2) \geq \sigma_{\mathcal{A}(j')}(t_1, t_2)/\alpha - \varepsilon M \geq \sigma_{\mathcal{A}(j)}(t_1, t_2)/\alpha - \varepsilon M$, so $\widetilde{\sigma}_{\widetilde{\mathcal{A}}(j)}(t_1, t_2)$ is at least as large this lower bound. There must be $m - j - 1$ objects i with $\widetilde{\sigma}_i(t_1, t_2) \leq \sigma_{\mathcal{A}(j)}(t_1, t_2) + \varepsilon M$, implying $\widetilde{\sigma}_{\widetilde{\mathcal{A}}(j)}(t_1, t_2) \leq \sigma_{\mathcal{A}(j)}(t_1, t_2) + \varepsilon M$. ■

Lemma 3.1 shows that an algorithm G satisfying Definition 3.1 implies an algorithm R satisfying Definition 3.2. That said, for either BreakPoints1 or BreakPoints2, Query1 is an (ε, 1)-approximation for $\sigma_i(t_1, t_2)$ and $\mathcal{A}(k, t_1, t_2)$; Query2 is an (ε, $2 \log r$)-approximation for $\sigma_i(t_1, t_2)$ and $\mathcal{A}(k, t_1, t_2)$. Despite the reduction in guaranteed accuracy for Query2, in practice its accuracy is not much worse than Query1, and it is 1-2 orders of magnitude better in space and construction time; and Query1 improves upon Exact3, the best exact method.

3.3.1 Breakpoints

Our key insight is that $\sigma_i(t_1, t_2)$ does not depend on the number of segments between the boundary times t_1 and t_2; it only depends on the aggregate σ applied to that range. So to approximate the aggregate score of any object within a range, we can discretize them based on the accumulated σ value. Specifically, we ensure that between no two consecutive breakpoints in $b_j, b_{j+1} \in \mathcal{B}$ does the value $\sigma_i(b_j, b_{j+1})$ become too large for an object. Both sets of breakpoints $\mathcal{B}_1$ for BreakPoints1 and $\mathcal{B}_2$ for BreakPoints2 start with $b_0 = 0$ and end with $b_r = T$. Given b_0, they sweep forward in time, always constructing b_j before b_{j+1}, and define:

$$b_{j+1} \text{ so} \begin{cases} \sum_{i=1}^{m} \sigma_i(b_j, b_{j+1}) = \varepsilon M, & \text{in BreakPoints1,} \\ \max_{i=1}^{m} \sigma_i(b_j, b_{j+1}) = \varepsilon M, & \text{in BreakPoints2,} \end{cases}$$

where $M = \sum_{i=1}^m \sigma_i(0,T)$. Note that these breakpoints b_j are not restricted to, and in general will not, occur at the end points of segments of some o_i.

Since the total aggregate $\sum_{i=1}^m \sigma_i(0,T) = M$, for BreakPoints1, there will be exactly $r = \lceil 1/\varepsilon + 1 \rceil$ breakpoints, as each (except for the last b_r) accounts for εM towards the total integral. For ease of exposition, we will assume that $1/\varepsilon$ is integral and drop the $\lceil \cdot \rceil$ notation, hence $1/\varepsilon \cdot \varepsilon M = M$. Next, we notice that BreakPoints2 will have at most as many breakpoints as BreakPoints1 since $\max_{i=1}^m X_i \leq \sum_{i=1}^m X_i$ for any set of $X_i > 0$. However, the inequality is not strict and these quantities could be equal; this implies the two cases could have the same number of breakpoints. This is restricted to the special case where between *every* consecutive pair $b_j, b_{j+1} \in \mathcal{B}$ *exactly one* object o_i has $\sigma_i(b_j, b_{j+1}) = \varepsilon M$ and for *every* other object $o_{i'}$ for $i \neq i'$ has zero aggregate $\sigma_{i'}(b_j, b_{j+1}) = 0$. As we will demonstrate on real data in Section 3.5, in most reasonable cases, the size of BreakPoints2 is dramatically smaller than the size of BreakPoints1.

3.3.1.1 Construction of BreakPoints1

We first need to preprocess all of the objects according to individual tuples for each vertex between two line segments. Consider two line segments s_1 and s_2 that together span from time t_L to time t_R and transition at time t_M. If they are part of object o_i, then they have values $v_L = g_i(t_L)$, $v_M = g_i(t_M)$, and $v_R = g_i(t_R)$. Then, for the vertex at (t_M, v_M), we store the tuple $(t_L, t_M, t_R, v_L, v_M, v_R)$. Then, we sort all tuples across all objects according to t_M in ascending order and place them in a queue Q. The breakpoints $\mathcal{B}_1$ will be constructed by popping elements from Q.

We need to maintain some auxiliary information while processing each tuple. For each tuple, we can compute the slope of its two adjacent segments as $w_L = (v_M - v_L)/(t_M - t_L)$ and $w_R = (v_R - v_M)/(t_R - t_M)$. Between each pair of segment boundaries, the value of an object $g_i(t)$ varies linearly according to the slope $w_{i,\ell}$ in segment $g_{i,\ell}$. Thus, the sum $\sum_{i=1}^m g_i(t)$ varies linearly according to $W(t) = \sum_{i=1}^m w_{i,\ell_i}$ if each ith object is currently represented by segment g_{i,ℓ_i}. Also, at any time t, we can write the summed value as $V(t) = \sum_{i=1}^m g_i(t)$. Now, for any two time points t_1 and t_2 such that no segments starts or ends in the range (t_1, t_2), and given $V(t_1)$ and $W(t_1)$, we can calculate in constant time the sum $\sum_{i=1}^m \sigma_i(t_1, t_2) = \frac{1}{2}W(t_1)(t_2 - t_1)^2 + V(t_1)(t_2 - t_1)$; note this derivation comes from similar observation from deriving Equation 3.1 that the range $[t_1, t_2]$ defines trapezoids for the objects. Thus, we always maintain $V(t)$ and $W(t)$ for the current t.

Since $b_0 = 0$, to construct $\mathcal{B}_1$, we only need to show how to construct b_{j+1} given b_j. Starting at b_j, we reset to 0 a running sum up to a time $t \geq b_j$ written $I(t) = \sum_{i=1}^m \sigma_i(b_j, t)$.

Then, we pop a tuple $(t_L, t_M, t_R, v_L, v_M, v_R)$ from Q and process it as follows. We update the running sum to time t_M as $I(t_M) = I(t) + \frac{1}{2}W(t)(t_M - t)^2 + V(t)(t_M - t)$. If $I(t_M) < \varepsilon M$, then we update $V(t_M) = V(t) + W(t)(t_M - t)$, then $W(t_M) = W(t) - w_L + w_R$, and pop the next tuple off of Q.

If $I(t_M) \geq \varepsilon M$, that means that the breakpoint b_{j+1} occurred somewhere between t and t_M. We can solve for this time b_{j+1} in the equation $I(b_{j+1}) = \varepsilon M$ where

$$0 = (\frac{1}{2}W(t))(b_{j+1} - t)^2 + (V(t))(b_{j+1} - t) + (I(t) - \varepsilon M)$$

as

$$b_{j+1} = t - \frac{V(t)}{W(t)} + \frac{1}{W(t)}\sqrt{(V(t))^2 - 2W(t)(I(t) - \varepsilon M)}.$$

The slope $W(t)$ has not changed, but we have to update $V(b_{j+1}) = V(t) + W(t) \cdot (b_{j+1} - t)$. Now, we reinsert the tuple at the top of Q to begin the process of finding b_{j+2}. Since each of N tuples is processed in linear time, the construction time is dominated by the $O((N/B) \log_B N)$ IOs for sorting the tuples.

3.3.1.2 Baseline construction of BreakPoints2

While construction of BreakPoints1 reduces to a simple scan over all segments (represented as tuples), computing BreakPoints2 is not as easy because of the replacement of the sum operation with a max. The difficulties come in resetting the maintained data at each breakpoint.

Again, we first need to preprocess all of the objects according to individual tuples for each line segment. We store the ℓth segment of o_i as the tuple $s_{i,\ell} = (t_L, t_R, v_L, v_R, i)$ which stores the left and right endpoints of the segment in time as t_L and t_R, respectively, and also stores the values it has at those times as $v_L = g_i(t_L)$ and $v_R = g_i(t_R)$, respectively. Note for each segment $s_{i,\ell}$ we can compute its slope $w_{i,\ell} = (v_R - v_L)/(t_R - t_L)$. Then, we sort all tuples across all objects according to t_L in ascending order and place them in a queue Q. The breakpoints $\mathcal{B}_2$ will be constructed by popping elements from Q.

By starting with $b_0 = 0$, we only need to show how to compute b_{j+1} given b_j. We maintain a running integral $I_i(t) = \sigma_i(b_j, t)$ for each object. Thus, at the start of a new break point b_j, each integral is set to 0. Then, for each new segment $s_{i,\ell}$ that we pop from Q, we update $I_i(t)$ to $I_i(t_R) = I_i(t) + (v_R - v_L)(t_R - t_L)/2$. If $I_i(t_R) < \varepsilon M$, then we pop the next tuple from Q and continue.

However, if the updated $I_i(t_R) \geq \varepsilon M$, then it means we have an event before the next segment will be processed from o_i. As before with BreakPoints1, we calculate

$\hat{b}_{j+1,i} = t + \frac{g_i(t)}{w_{i,\ell}} + \frac{1}{w_{i,\ell}}\sqrt{(g_i(t))^2 - 2w_{i,\ell}(I_i(t) - \varepsilon M)}$. This is not necessarily the location of the next breakpoint b_{j+1}, but if the breakpoint is caused by o_i, then this will be it. We call such objects for which we have calculated $\hat{b}_{j+1,i}$ as *dangerous*. We let $\hat{b}_{j+1} = \min \hat{b}_{j+1,i}$ (where $\hat{b}_{j+1,i}$ is implicitly ∞ if it is not dangerous). To determine the true next breakpoint, we keep popping tuples from Q until for the current tuple $t_L > \hat{b}_{j+1}$. This indicates no more segment endpoints occur before some object o_i reaches $I_i(t) = \varepsilon M$. So we set $b_{j+1} = \hat{b}_{j+1}$, and reset maintained values in preparation for finding b_{j+2}.

Assuming $\Omega(m/B)$ internal memory space, this method runs in $O((N/B)\log_B N)$ IOs, as we can maintain m running sums in memory. We can remove this assumption in $O((N/B)\log_B N)$ IOs with some technical tricks which we omit. To summarize, after sorting in $O(\log_B N)$ passes on the data, we determine for each segment from each o_i how many segments occur again before another segment from o_i is seen. We then keep the auxiliary information for each object (e.g., running sums) in an IO-efficient priority queue [64] on the objects sorted by the order in which a segment from each object will next appear.

However, with limited internal space or in counting internal runtime, this method is still potentially slower than finding BreakPoints1 since it needs to reset each $I_i(b_{j+1}) = 0$ when we reach a new breakpoint. This becomes clear when studied from an internal memory runtime perspective, where this method may take $O(rm + N\log N)$ time.

3.3.1.3 Efficient construction of BreakPoints2

We can avoid the extra $O(rm)$ term in the run time by using clever bookkeeping that ensures we do not have to reset too much each time we find a breakpoint. The Appendix (Section 9.1) of our technical report [65] shows:

Lemma 3.2. BreakPoints2 *can be built in $O(N\log N)$ time (for $N > 1/\varepsilon$). Its size is $r = O(1/\varepsilon)$; and it takes $O((N/B)\log_B N)$ IOs to construct.*

3.3.1.4 Remarks

For specific datasets, there may be other specialized ways of choosing breakpoints. For real-world datasets, such as the MesoWest data as shown in Figure 3.1, our methods are both efficient and have excellent approximation quality (see Section 3.5).

3.3.2 Index breakpoints and queries

Given a set of breakpoints $\mathcal{B}$ (either $\mathcal{B}_1$ or $\mathcal{B}_2$), we show how to answer queries on the full dataset approximately. The approximation guarantees are based on the following property that holds for BreakPoints1 $\mathcal{B}_1$ and BreakPoints2 $\mathcal{B}_2$. For any query interval (t_1, t_2), let $(\mathcal{B}(t_1), \mathcal{B}(t_2))$ be the associated *approximate interval*, where $\mathcal{B}(t_1)$ (resp. $\mathcal{B}(t_2)$) is the smallest breakpoints in $\mathcal{B}$ such that $\mathcal{B}(t_1) \geq t_1$ (resp. $\mathcal{B}(t_2) \geq t_2$); see Figure 3.7.

Lemma 3.3. *For any query* $[t_1, t_2]$ *and associated approximate interval* $[\mathcal{B}(t_1), \mathcal{B}(t_2)]$*:* $\forall o_i$, $|\sigma_i(t_1, t_2) - \sigma_i(\mathcal{B}(t_1), \mathcal{B}(t_2))| \leq \varepsilon M$.

Proof of Lemma 3.3: Both $\mathcal{B}_1$ and $\mathcal{B}_2$ guarantee that between any two consecutive breakpoints $b_j, b_{j+1} \in \mathcal{B}$ that for any object $\sigma_i(b_j, b_{j+1}) \leq \varepsilon M$. This property is guaranteed directly for BreakPoints2, and is implied by BreakPoints1 because for any object o_i it holds that $\sigma_i(t_1, t_2) \leq \sum_{j=1}^{m} \sigma_j(t_1, t_2)$ for each $\sigma_j(t_1, t_2) \geq 0$, which is the case since we assume positive scores (this restriction is removed in Section 3.4).

Hence, by changing the query interval from $[t_1, t_2]$ to $[\mathcal{B}(t_1), t_2]$, the aggregate can only *decrease*, and can decrease by at most εM. Also, by changing the interval from $[\mathcal{B}(t_1), t_2]$ to $[\mathcal{B}(t_1), \mathcal{B}(t_2)]$, the aggregate can only *increase*, and can increase by at most εM. Thus, the inequality holds since each endpoint change can either increase *or* decrease the aggregate by at most εM. ■

We now present two query methods, and associate data structures, called Query1 and Query2.

3.3.2.1 Nested B+-tree queries

For Query1, we consider all $\binom{r}{2}$ intervals with a breakpoint from $\mathcal{B}$ at each endpoint. For each of these intervals $[b_j, b_{j'}]$, we construct the $k_{\max}$ objects with the largest aggregate $\sigma_i(b_j, b_{j'})$. Now we can show that this nested B+-tree yields an $(\varepsilon, 1)$-approximation for both the aggregate scores and $\mathcal{A}(k, t_1, t_2)$ for any $k \leq k_{\max}$.

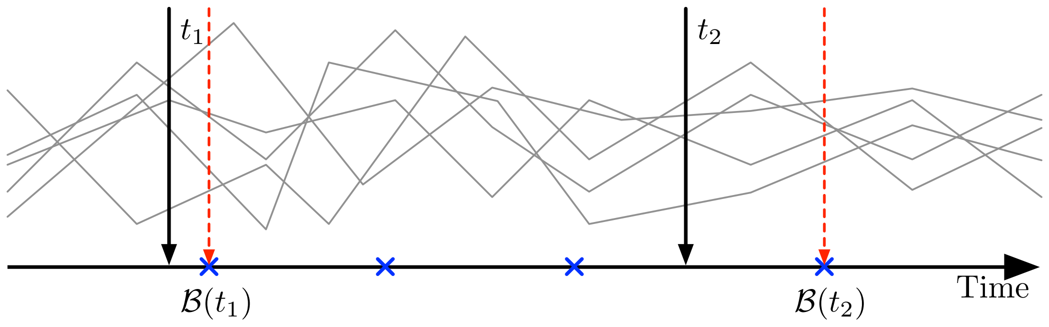


Figure 3.7. Associated approximate interval.

To construct the set of $k_{\max}$ objects associated with each interval $[b_j, b_{j'}]$, we use a single linear sweep over all segments using operations similar to Exact1. Starting at each breakpoint b_j, we initiate a running integral for each object to represent the intervals with b_j as their left endpoint. Then at each other breakpoints $b_{j'}$, we output the $k_{\max}$ objects with largest running integrals starting at each b_j up to $b_{j'}$ to represent $[b_j, b_{j'}]$. That is, we maintain $O(r)$ sets of m running integrals, one for each left breakpoint b_j we have seen so far (to avoid too much internal space in processing all N segments, we use a single IO-efficient priority queue as in constructing BreakPoints2, where each of m objects in the queue now also stores $O(r)$ running sums). We also maintain $O(r)$ priority queues of size $k_{\max}$ for each left endpoint b_j, over each set of m running integrals on different objects. This takes $O((N/B)(\log_B(mr) + r\log_B k_{\max}) + r(rk_{\max}/B + 1))$ IOs, where the last item counts for the output size (since we have $O(r^2)$ intervals and each interval stores $k_{\max}$ objects). We assume $rk_{\max} < N$ (to simplify and so index size $O(r^2 k_{\max})$ is feasible); hence, the last term is absorbed in $O(\cdot)$.

To index the set of these intervals, we use a nested set of B+-trees. We first build a B+-tree T_{top} on the breakpoints $\mathcal{B}$. Then for each leaf node associated with b_j, we point to another B+-tree T_j on $\mathcal{B}'_j$, where $\mathcal{B}'_j = \{b \in \mathcal{B} \mid b > b_j\}$. The top level B+-tree T_{top} indexes the left endpoint of an interval $[b_j, b_{j'}]$ and the lower level B+-tree T_j pointed to by b_j in T_{top} indexes the right end point $b_{j'}$ (for all $b_{j'} > b_j$). We build $O(r)$ B+-trees of size $O(r)$, hence, this step takes $O(r^2/B)$ IOs (by bulkloading). Again, we assume $r^2 < N$, and this cost will also be absorbed in the construction cost.

Now we can query any interval in $O(\log_B r)$ time, since each B+-tree requires $O(\log_B r)$ to query, and for a query top-$k(t_1, t_2, \sigma)$, we use T_{top} to find $\mathcal{B}(t_1)$, and the associated lower level B+-tree of $\mathcal{B}(t_1)$ to find $\mathcal{B}(t_2)$, which gives the top $k_{\max}$ objects in interval $[\mathcal{B}(t_1), \mathcal{B}(t_2)]$. We return the top k objects from them as $\widetilde{\mathcal{A}}$ (see Figure 3.8). The above and Lemma 3.3 imply the following results.

Lemma 3.4. *Given breakpoints $\mathcal{B}$ of size r ($r^2 < N$ and $rk_{\max} < N$), Query1 takes $O((N/B)\ (\log_B(mr) + r\log_B k_{\max}))$ IOs to build, has size $\Theta(r^2 k_{\max}/B)$, and returns $(\varepsilon, 1)$-approximate top-k queries, for any $k \le k_{\max}$, in $O(k/B + \log_B r)$ IOs.*

3.3.2.2 Dyadic interval queries

Query1 provides very efficient queries, but requires $\Omega(r^2 k_{\max}/B)$ blocks of space which for small values of ε can be too large (as $r = O(1/\varepsilon)$ in both types of breakpoints). For arbitrarily small ε, it could be that $r^2 > N$. It also takes $\Omega(rN \log k_{\max})$ time to build.

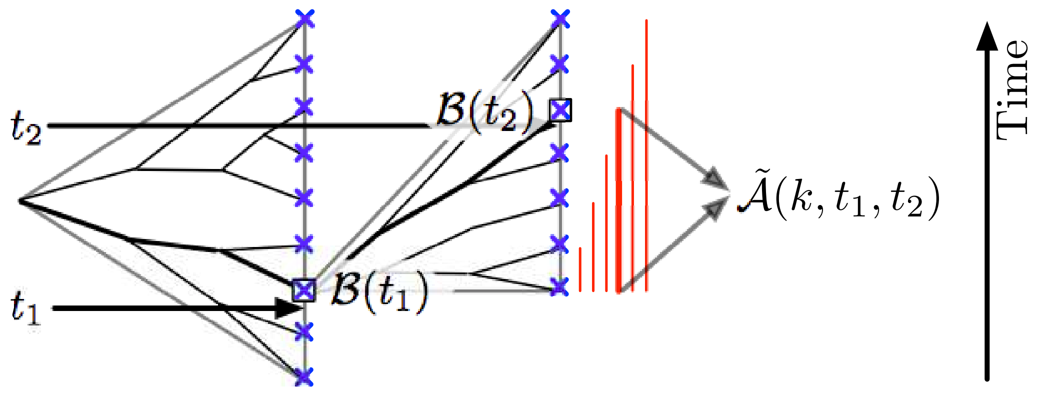


Figure 3.8. Illustration of Query1.

Thus, we present an alternative approximate query structure, called Query2, that uses only $O(rk_{\max}/B)$ space, still has efficient query times and high empirical accuracy, but has slightly worse accuracy guarantees. It is a $(\varepsilon, 2\log r)$-approximation for both $\sigma_i(t_1, t_2)$ and $\mathcal{A}(k, t_1, t_2)$.

We consider all dyadic intervals, that is all intervals $[b_j, b_{j'}]$ where $j = h2^\ell + 1$ and $j' = (h+1)2^\ell$ for some integer $0 \le \ell < \log r$ and $0 \le h \le r/2^\ell - 1$. Intuitively, these intervals represent the span of each node in a balanced binary tree. At each level ℓ, the intervals are of length 2^ℓ, and there are $\lceil r/2^\ell \rceil$ intervals. There are less than $2r + \log r$ such intervals in total since there are r at level 0, $\lceil r/2 \rceil$ at level 1, and so on, geometrically decreasing.

As with Query1 for each dyadic interval $[b_j, b_{j'}]$, we find the k_{max} objects with the largest $\sigma_i(b_j, b_{j'})$ in a single sweep over all N segments. There are $\log r$ active dyadic intervals at any time, one at each level, so we maintain $\log r$ running integrals per object. We do so again using two IO-efficient priority queues. One requires $O((1/B)\log_B(m \log r))$ IOs per segment, the elements correspond to objects sorted by which have segments to processes next, and each element stores the $\log r$ associated running integrals. The second is *a set* of $\log r$ IO-efficient priority queues of size $k_{\max}$, sorted by the value of the running integral; each requires $O((1/B)\log_B k_{\max})$ IOs per segment. The total construction is $O((N/B)(\log_B(m \log r) + \log r \log_B k_{\max}))$ IOs.

In dyadic intervals, any interval $[b_1, b_2]$ can be formed as the disjoint union of at most $2\log r$ dyadic intervals. We use this fact as follows: for each query interval $[t_1, t_2]$, we determine the at most $2\log r$ dyadic intervals that decompose the associated approximate query interval $[\mathcal{B}(t_1), \mathcal{B}(t_2)]$. For each such dyadic interval, we retrieve the top-k objects and scores from its associated top-$k_{\max}$ objects ($k \le k_{\max}$), and insert them into a candidate set $\mathcal{K}$, adding scores of objects inserted more than once. The set $\mathcal{K}$ is of size at most $k 2\log r$. We return the k objects with the top k summed aggregate scores from $\mathcal{K}$.

Lemma 3.5. QUERY2 *$(\varepsilon, 2\log r)$-approximations $\mathcal{A}(k, t_1, t_2)$.*

Proof of Lemma 3.5: Converting $[t_1, t_2]$ to $[\mathcal{B}(t_1), \mathcal{B}(t_2)]$ creates at most εM error between $\sigma_i(t_1, t_2)$ and $\sigma_i(\mathcal{B}(t_1), \mathcal{B}(t_2))$, as argued in Lemma 3.3. This describes the additive εM term in the error, and allows us to hereafter consider only the lower bound on scores over the approximate query interval $[\mathcal{B}(t_1), \mathcal{B}(t_2)]$.

The relative $2\log r$ factor is contributed to by the decomposition of $[\mathcal{B}(t_1), \mathcal{B}(t_2)]$ into at most $2\log r$ disjoint intervals. For each object $o_i \in \mathcal{A}(t_1, t_2)$, some such interval $[b_j, b_{j'}]$ must satisfy $\sigma_i(b_j, b_{j'}) \ge \sigma_i(\mathcal{B}(t_1), \mathcal{B}(t_2))/(2\log r)$. For this interval, if o_i is in the top-k, then we return a value at least $\sigma_i(b_j, b_{j'}) \ge \sigma_i(\mathcal{B}(t_1), \mathcal{B}(t_2))/(2\log r)$. If o_i is not in the top-k for $[b_j, b_{j'}]$, then each object $o_{i'}$ that is in that top-k set has

$$\sigma_{i'}(\mathcal{B}(t_1), \mathcal{B}(t_2)) \ge \sigma_{i'}(b_j, b_{j'}) \ge \sigma_i(b_j, b_{j'}) \ge \frac{\sigma_i(\mathcal{B}(t_1), \mathcal{B}(t_2))}{2\log r}.$$

Thus, there must be at least k objects $o_{i'} \in \tilde{\mathcal{A}}(\mathcal{B}(t_1), \mathcal{B}(t_2))$ with $\sigma_{i'}(\mathcal{B}(t_1), \mathcal{B}(t_2)) \ge \sigma_i(\mathcal{B}(t_1), \mathcal{B}(t_2))/(2\log r)$. ■

To efficiently construct the set $\mathcal{K}$ of at most $k 2\log r$ potential objects to consider being in $\tilde{\mathcal{A}}(k, t_1, t_2)$, we build a balanced binary tree over $\mathcal{B}$. Each node (either an internal node or leaf node) corresponds to a dyadic interval (see Figure 3.9). We construct the set of such intervals that form the disjoint union over $[\mathcal{B}(t_1), \mathcal{B}(t_2)]$ as follows. In Phase 1, starting at

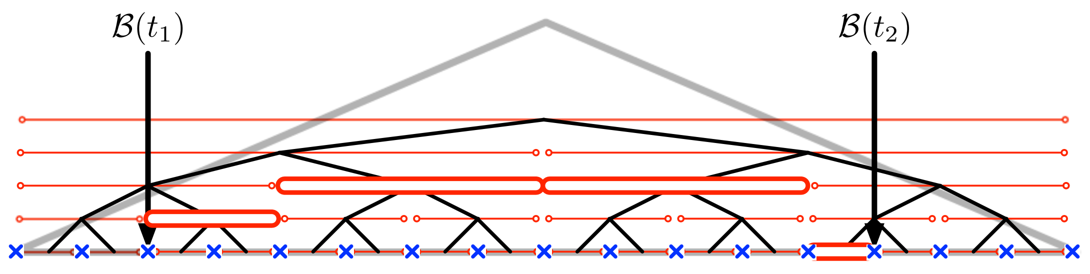


Figure 3.9. Illustration of QUERY2.

the root, if $[t_1, t_2]$ is completely contained within one child, we recurse to that child. Phase 2 begins when $[t_1, t_2]$ is split across both children of a node, so we recur on each child. On the next step, Phase 3 begins, we describe the process for the left child; the process is symmetric for the right child. If t_1 is within the right child, we recur to that child. If t_1 is within the left child, we return the dyadic interval associated with the right child and recur on the left child. Finally, if t_1 separates the left child from the right child, we return the dyadic interval associated with the right child and terminate. Since the height of the tree is at most $\log r$, and we return at most one dyadic interval at each level for the right and left case of Phase 3, then there are at most $2 \log r$ dyadic intervals returned. The above idea can be easily generalized to a B+-tree (simply with larger fanout) if r is large.

Lemma 3.6. *Given breakpoints $\mathcal{B}$ of size r,* QUERY2 *requires size $\Theta(rk_{\max}/B)$, takes $O((N/B)(\log_B(m \log r) + \log r \log_B k_{\max}))$ cost to build, and answers $(\varepsilon, 2\log r)$-approximate top-k queries, for any $k \leq k_{\max}$, in $O(k \log r \log_B k)$ IOs.*

Proof of Lemma 3.6: The error bound follows from Lemma 3.5, and the construction time is argued above. The query time is dominated by maintaining a size k priority queue over the set $\mathcal{K}$ with $O(k \log r)$ objects inserted, from k objects in $O(\log r)$ dyadic intervals.

■

3.3.3 Combined approximate methods

Finally, we formalize different approximate methods: APPX1-B, APPX2-B, APPX1, APPX2. As shown in Figure 3.6, the methods vary based on how we combine the construction of breakpoints and the query structure on top of them. APPX1 and APPX2 use BREAKPOINTS2 followed by either QUERY1 or QUERY2, respectively. As we will demonstrate in Section 3.5, BREAKPOINTS2 is superior to BREAKPOINTS1 in practice; so, we designate APPX1-B (BREAKPOINTS1 +QUERY1) the *basic* version of APPX1, and APPX2-B (BREAKPOINTS1 +QUERY2) the basic version of APPX2.

The analysis between the basic and improved versions are largely similar; hence, we only list the improved versions in Table 3.2. In particular, for the below results, since $r = \Theta(1/\varepsilon)$ in BREAKPOINTS1, we can replace r with $1/\varepsilon$ for the basic results.

APPX1 computes $r = O(1/\varepsilon)$ breakpoints $\mathcal{B}_2$ using BREAKPOINTS2 in $O((N/B) \log_B (N/B))$ IOs. Then QUERY1 requires $O(r^2 k_{\max}/B)$ space, $O((N/B)(\log_B(mr) + r \log_B k_{\max}))$ construction IOs, and can answer $(\varepsilon, 1)$-approximate queries in $O(k/B + \log_B r)$ IOs. Since $m, r < N$, this simplifies the total construction IOs to $O((N/B)\ (\log_B N + r \log_B k_{\max}),$

the index size to $O(r^2 k_{\max}/B)$, and the IOs for an $(\varepsilon, 1)$-approximate top-k query to $O(k/B + \log_B r)$.

In Appx2, Query2 has $O(rk_{\max}/B)$ space, builds in $O((N/B)$ $(\log_B(m \log r) + \log r \log_B$ $k_{\max}))$ IOs, and answers $(\varepsilon, 2\log r)$-approximate queries in $O(k \log r \log_B k)$ IOs. As $m, r < N$, the bounds simplify to $O((N/B)$ $(\log_B N + \log r \log_B k_{\max}))$ build cost, $O(k \log r \log_B k)$ query IOs, and $O(rk_{\max}/B)$ index size. We also consider a variant Appx2+, which discovers the exact aggregate value for each object in $\mathcal{K}$ using a B+-tree from Exact2. This increases the index size by $O(N/B)$ (basically just storing the full data), and increases the query IOs to $O(k \log r \log_B k)$, but significantly improves the empirical query accuracy.

3.4 Other Remarks

3.4.1 Updates

In most applications, temporal data receive updates only at the current time instance, which extend a temporal object for some specified time period. In this case, we can model an update to an object o_i as appending a new line segment g_{i,n_i+1} to the end of g_i, where that g_{i,n_i+1}'s left end-point is (t_{i,n_i}, v_{i,n_i}) (the right end-point of g_{i,n_i}); g_{i,n_i+1}'s right end-point is $(t_{i,n_i+1}, v_{i,n_i+1})$.

Handling updates in exact methods are straightforward. In Exact1, we insert a new entry (t_{i,n_i}, g_{i,n_i+1}) into the B+-tree; hence, the update cost is $O(\log_B N)$ IOs. In Exact2, we insert a new entry $(t_{i,n_i+1}, (g_{i,n_i+1}, \sigma_i(I_{i,n_i+1}))$ to the B+-tree T_i, where I_{i,n_i+1} $= [t_{i,0}, t_{i,n_i+1}]$. We can compute $\sigma_i(I_{i,n_i+1})$ based on $\sigma_i(I_{i,n_i})$ and g_{i,n_i+1} in $O(1)$ cost; and $\sigma_i(I_{i,n_i})$ is retrieved from the last entry in T_i in $O(\log_B n_i)$ IOs. So, the update cost is $O(\log_B n_i)$ IOs. In Exact3, a new entry $([t_{i,n_i}, t_{i,n_i+1}], (g_{i,n_i+1}, \sigma_i(I_{i,n_i+1})))$ is inserted into the interval tree S. For similar arguments, $\sigma_i(I_{i,n_i})$ is retrieved from S in $O(\log_B N)$ IOs; and then $\sigma_i(I_{i,n_i+1})$ is computed in $O(1)$. The insertion into S is $O(\log_B N)$ IOs [60]. Thus, the total update is $O(\log_B N)$ IOs.

Handling updates in approximate methods is more complicated. As such, we described amortized analysis for updates. This approach can be de-amortized using standard technical tricks. The construction of breakpoints depends on a threshold $\tau = \varepsilon M$; however, M increases with updates. We handle this by always constructing breakpoints (and the index structures on top of them) using a fixed value of τ, and when M doubles, we rebuild the structures. For this to work, we assume that it takes $\Omega(N)$ segments before M doubles; otherwise, a segment ℓ could have an aggregate of $M/2$, and one has to rebuild the entire query structure immediately after seeing ℓ. Thus, in an amortized sense, we can amortize the construction time $C(N)$ over $\Omega(N)$ segments, and charge $O(C(N)/N)$ to the update time of a segment.

We also need to maintain a query structure and set of breakpoints on top of the segments just added. Adding the breakpoints can be done by maintaining the same IO-efficient data structures as in their initial construction, using $O(\frac{1}{B}\log_B N)$ IOs per segment. To maintain the query structures, we again maintain the same auxiliary variables and running integrals as in the construction. Again, assuming that there are $\Omega(N/r)$ segments between any pair of breakpoints, we can amortize the building of the query structures to the construction cost divided by N. The amortized reconstruction or incremental construction of the query structures dominate the cost. For Appx1, we need $O(\frac{1}{B}(\log_B N + r\log_B k_{\max}))$ IOs to update Query1. For Appx2, we need $O(\frac{1}{B}(\log_B N + \log r \log_B k_{\max}))$ IOs to update Query2.

3.4.2 General time series with arbitrary functions

In some time series data, objects are described by arbitrary functions f, instead of piecewise linear functions g. However, as we explained in Section 3.1, a lot of efforts have been devoted to approximate an arbitrary function f using a piecewise linear function g in general time series (see [66] and references therein). All of our methods may be extended to work with any *piecewise polynomial functions* p: one change is that we need to deal with polynomial curve segments, instead of linear line segments. This affects how to compute $\sigma_i(I)$ of an interval I, which is a subinterval of the interval defined by the two end-points of a polynomial curve segment $p_{i,j}$ (the jth polynomial function in the ith object). However, this can be easily fixed. Instead of using (3.1) based on a trapezoid, we simply compute it using the integral over $p_{i,j}$, i.e., $\sigma_i(I) = \int_{t\in I} p_{i,j}(t)d(t)$. Given that $p_{i,j}(t)$ is a polynomial function, this can be easily computed. We have explicitly optimized the construction of BreakPoints1 and BreakPoints2 assuming a piecewise linear representation. Therefore, a secondary issue would be to develop efficient schemes for computing breakpoints where piecewise polynomial segments are used instead of piecewise linear segments. That said, when one needs more precision in representing an arbitrary time series, either one can use more line segments in a piecewise linear representation, or one can use a piecewise polynomial representation.

3.4.3 Negative values

We have assumed positive score values so far. However, this restriction can be easily removed. Clearly, it does not affect our exact methods at all. In the approximate methods, when computing the breakpoints (in either approach), we use the absolute values instead to define M and when searching for a breakpoint. We omit technical details, but we can show that doing so will still guarantee the same approximations.

3.4.4 Other aggregates

Our work focuses on the sum aggregation. This automatically implies the support to the avg aggregation, and many other aggregations that can be expressed as linear combinations of the sum (such as F_2, the 2nd frequency moment), e.g., for avg, the ranking of items within a time interval $[t1, t2]$ remain the same as we are only changing the aggregate score for each item o_i from $\int_{t_1}^{t_2} g_i(t)dt$ to $\frac{1}{t_2-t_1} \cdot \int_{t_1}^{t_2} g_i(t)dt$. However, ranking by some holistic aggregates is hard. An important one in this class is the quantile (median is a special case of the quantile). We leave the question of how to rank large temporal data using some of the holistic aggregates (e.g., quantile) as an open problem.

3.5 Experiments

We design all of our algorithms to efficiently consider disk IOs; in particular, we implemented all our methods using the TPIE-library in C++ [67]. This allows our methods to scale gracefully to massive data that do not fit in memory. All experiments were performed on a Linux machine with an Intel Core i7-2600 3.4GHz CPU, 8GB of memory, and a 1TB hard drive.

3.5.1 Datasets

We used two large real datasets. The first dataset is a temperature dataset, *Temp*, from the MesoWest project [5]. It contains temperature measurements from Jan 1997 to Oct 2011 from 26,383 distinct stations across the United States. There are almost N=2.6 billion total readings from all stations with an average of 98,425 readings per station. For our experiments, we preprocessed the *Temp* dataset to treat *each year of readings from a distinct station as a distinct object.* By aligning readings in this manner, we can ask which k stations had the highest aggregate temperatures in a (same) time interval amongst any of the recorded years. After preprocessing, *Temp* has m=145,628 objects with an average number of readings per object of n_{avg}=17,833. In each object, we connect all consecutive readings to obtain a piecewise-linear representation.

The second real dataset, *Meme*, was obtained from the Memetracker project. It tracks popular quotes and phrases which appear from various sources on the internet. The goal is to analyze how different quotes and phrases compete for coverage every day and how some quickly fade out of use while others persist for long periods of time. A record has 4 attributes, the URL of the website containing the memes, the time Memetracker observed the memes, a list of the observed memes, and links accessible from the website. We preprocess the *Meme* dataset, converting each record to have a distinct 4-byte integer *id* to represent the URL, an 8-byte double to represent the *time* of the record, and an 8-byte double to represent a record's *score*. A record's score is the number of memes appearing on the website, i.e., it is the cardinality of the list of memes. After preprocessing, *Meme* has almost m=1.5 million distinct objects (the distinct URLs) with N=100 million total records, an average of n_{avg}=67 records per object. For each object, we connect every two of its consecutive records in time (according to the date) to create a piecewise linear representation of its score.

3.5.2 Setup

We use *Temp* as the default dataset. To test the impact of different variables, we have sampled subsets of *Temp* to create datasets of different number of objects (m), different number of average line segments per object (n_{avg}, by limiting the maximum value T). By default, $m = 50,000$ and $n_{\text{avg}} = 1,000$ in *Temp*, so all exact methods can finish in a reasonable amount of time. Still, there are a total of $N = 50 \times 10^6$ line segments! The default values of other important variables in our experiments are: $k_{\max} = 200$, $k = 50$, $r = 500$ (number of breakpoints in both BreakPoints1 and BreakPoints2), and $(t_2 - t_1) = 20\%T$. The disk block size in TPIE is set to 4KB. For each query-related result, we generated 100 random queries and report *the average* for all query-related results, including query time, I/Os, precision/recall, and approximation ratio. Lastly, in all datasets, all line segments are sorted by the time value of their left end-point.

3.5.3 Number of breakpoints

We first investigate the effect of the number of breakpoints r on different approximate methods, by changing r from 100 to 1000. Figure 3.10 shows the preprocessing results and Figure 3.11 shows the query results. Figure 3.10(a) indicates that given the same number of breakpoints, the value of the error parameter ε using BreakPoints2 $\mathcal{B}_2$ is much smaller than that in BreakPoints1 $\mathcal{B}_1$ in practice; this confirms our theoretical analysis, since $r = 1/\varepsilon$ in $\mathcal{B}_1$, but $r = O(1/\varepsilon)$ in $\mathcal{B}_2$. This suggests that $\mathcal{B}_2$ offers *much higher* accuracy than $\mathcal{B}_1$ given the same budget r on real datasets. With 500 breakpoints, ε in $\mathcal{B}_2$ reduces to almost 10^{-8}, while it is still 0.02 in $\mathcal{B}_1$. Figure 3.10(b) shows the build time of $\mathcal{B}_1$ and $\mathcal{B}_2$. Clearly, building $\mathcal{B}_1$ is independent to r since its cost is dominated by the linear sweeping

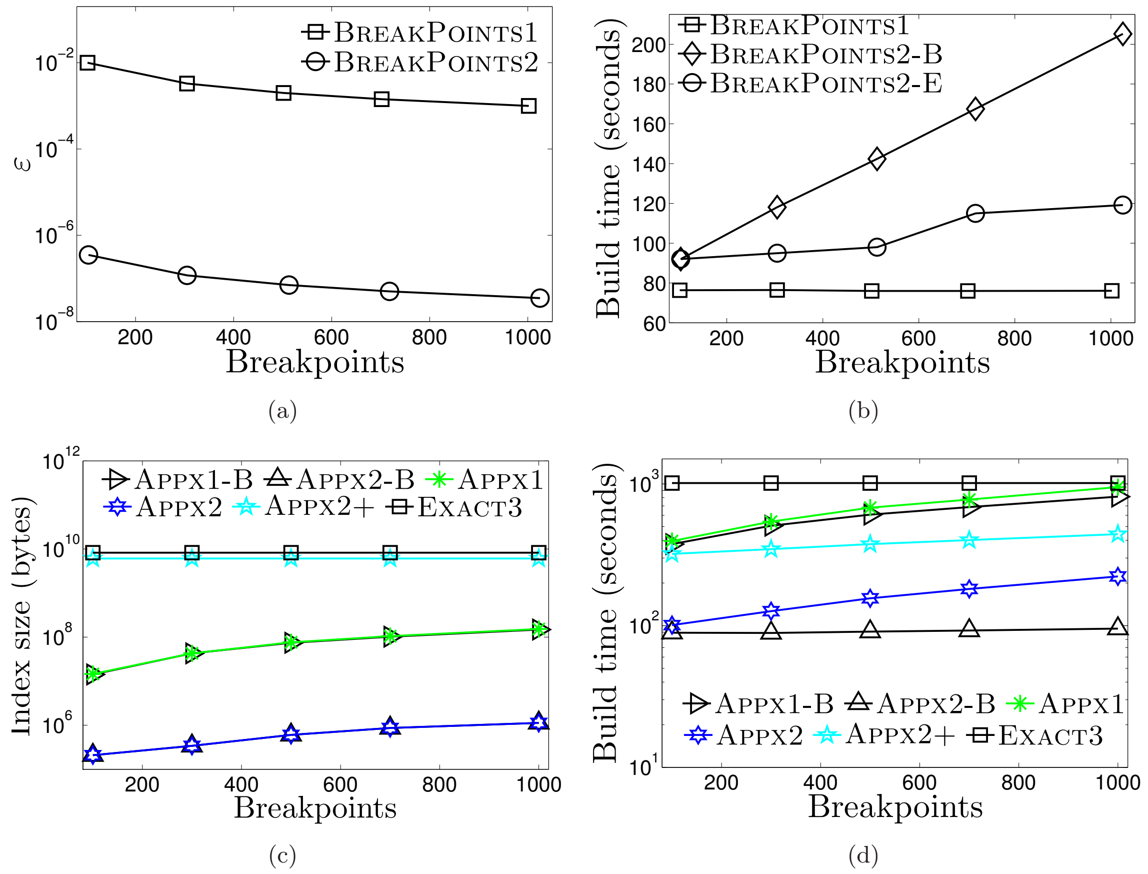


Figure 3.10. Vary r for approximate methods on Temp: versus (a) ε, (b) time, (c) index size, (d) build time.

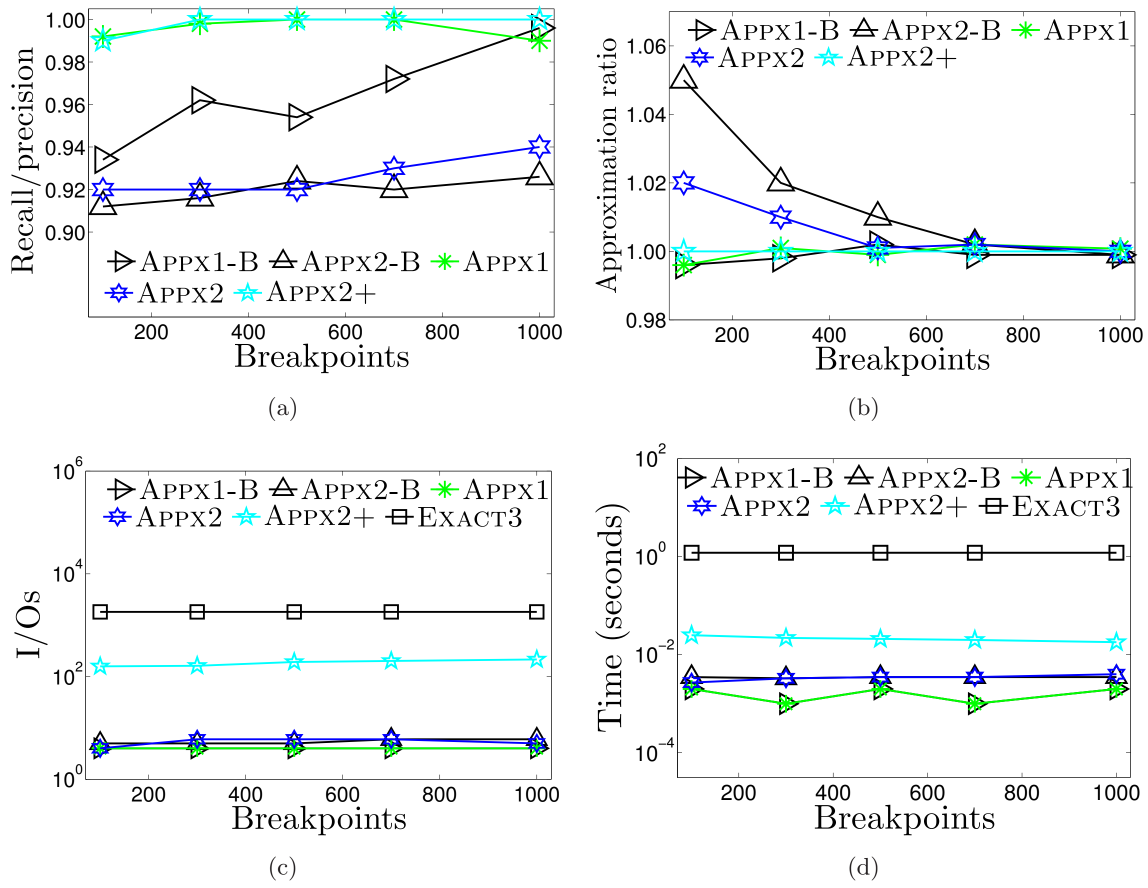


Figure 3.11. Vary r for approximate methods on Temp: versus (a) recall/precision, (b) ratio, (c) I/Os, (d) query time.

of all line segments. The baseline method for building $\mathcal{B}_2$, BreakPoints2-B clearly has a linear dependency on r (on m as well, which is not reflected by this experiment). However, our efficient method of building $\mathcal{B}_2$, BreakPoints2-E, has largely removed this dependency on r, as shown in Figure 3.10(b). It also removed the dependency on m, though not shown. In what follows, BreakPoints2-E was used by default. Both $\mathcal{B}_1$ and $\mathcal{B}_2$ can be built fairly fast, in only 80 and 100 seconds, respectively, when $r = 500$ (over 50×10^6 segments!).

Next, we investigate the index size and the construction cost of approximate methods, using Exact3 as a reference (as it has the best query performance among all exact methods). Figure 3.10(c) shows that all approximate methods have much smaller size than Exact3, except Appx2+ which also builds Exact2 since it calculates the exact aggregate score for candidates in $\mathcal{K}$ from Appx2. Clearly, Appx1-B and Appx1 have the same size; basic and improved versions only differ in which types of breakpoints they index using the two-level B+-trees. For the same reason, Appx2-B and Appx2 also have the same size; they index $\mathcal{B}_1$ or $\mathcal{B}_2$ using a binary tree over the dyadic intervals. Appx2-B and Appx2 only have size $O(rk_{\max})$, while Appx1-B and Appx1 have size $O(r^2k_{\max})$ and Exact3 and Appx2+ have linear size $O(N)$, which explains that the size of Appx2-B and Appx2 is more than 2 orders of magnitude smaller than the size of Appx1-B and Appx1, which are in turn 3-2 orders of magnitude smaller than Exact3 and Appx2+ when r changes from 100 to 1000. In fact, Appx2-B and Appx2 take only 1MB, and Appx1-B and Appx1 take only 100MB, when $r = 1000$; Exact3 and Appx2+ take more than 10GB. Construction time (for building both breakpoints and subsequent query structures) for approximate methods (including Appx2+) are much faster than Exact3, as shown in Figure 3.10(d). All structures build in only 100 to 1000 seconds. Not surprisingly, Appx2-B and Appx2 are the fastest, since they only need to find the top $k_{\max}$ objects for $O(r)$ intervals, while Appx1-B and Appx1 need to find the top $k_{\max}$ objects for $O(r^2)$ intervals. Even Appx2+ is significantly faster to build than Exact3 since Exact2 builds faster than Exact3. All approximate methods are generally faster to build than Exact3, by 1-2 orders of magnitude (except for Appx1 when r reaches 1000) since the top $k_{\max}$ objects can be found in a linear sweep over all line segments, as explained in Section 3.3.2.

In terms of the query performance, we first examine the approximation quality of all approximate methods, using both the standard precision/recall (between $\widetilde{\mathcal{A}}$ and $\mathcal{A}$), and the average of the approximation ratios defined as $\widetilde{\sigma}_i(t_1, t_2)/\sigma_i(t_1, t_2)$ for any o_i returned in $\widetilde{\mathcal{A}}$. Since $|\widetilde{\mathcal{A}}|$ and $|\mathcal{A}|$ are both k, the precision and the recall will have the same denominator value. Figure 3.11(a) shows that all approximate methods have precision/recall higher than

90% even in the worst case when $r = 100$; in fact, Appx1 and Appx2+ have precision/recall close to 1 in all cases. Figure 3.11(b) further shows that Appx1, Appx1-B, and Appx2+ have approximate ratios on the aggregate scores very close to 1, whereas Appx2 and Appx2-B have approximation ratios within 5% of 1. In both figures, Appx1 and Appx2 using $\mathcal{B}_2$ are indeed better than their basic versions Appx1-B and Appx2-B using $\mathcal{B}_1$, since given the same number of breakpoints, $\mathcal{B}_2$ results in much smaller ε values (see Figure 3.10(a)). Similar results hold for Appx2+, and are omitted to avoid clutter. Nevertheless, all methods perform much better in practice than their theoretical error parameter ε suggests (which indicates worst-case analysis). Not surprisingly, both types of approximation qualities from all approximate methods improve when r increases; but $r = 500$ already provides excellent qualities.

Finally, in terms of query cost, approximate methods are clear winners over the best exact method Exact3, with better IOs in Figure 3.11(c) and query time in Figure 3.11(d). In particular, Appx1-B and Appx1 (reps. Appx2-B and Appx2) have the same IOs given the same r values, since they have identical index structures except different values of entries to index. These four methods have the smallest number of IOs among all methods, in particular, 6-8 IOs in all cases. All require only two queries in a B+-tree of size r: a top-level and lower-level tree for Appx1 and Appx1-B, and a left- and right-endpoint query for Appx2 and Appx2-B. Appx2+ is slower with about 100 to 150 IOs in all cases, due to the fact that after identifying the candidate set $\mathcal{K}$, it needs to verify the exact score of each candidate. However, since it only needs to deal with $2k \log r$ candidates in the worst case, and in practice, $|\mathcal{K}| \ll 2k \log r$, its IOs are still very small. In contrast, the best exact method Exact3 takes more than 1000 IOs.

Smaller IO costs lead to much better query performance; all approximate methods outperform the best exact method Exact3 by at least 2 orders of magnitude in Figure 3.11(d). In particular, they generally take less than 0.01 seconds to answer a top-50(t_1, t_2, sum) query, in 20% time span over the entire temporal domain, over 50×10^6 line segments from $50,000$ objects, while the best exact method Exact3 takes around 1 second for the same query. The fastest approximate method only takes close to 0.001 second!

From these results, clearly, Appx1 and Appx2 using $\mathcal{B}_2$ are better than their corresponding basic versions Appx1-B and Appx2-B using $\mathcal{B}_1$, given the same number of breakpoints; and $r = 500$ already gives excellent approximation quality (the same holds for Appx2+, which we omit to avoid clutter). As such, we only use Appx1, Appx2, and Appx2 + for the remaining experiments with $r = 500$. Among the three, Appx2+ is larger and slower to build than Appx1, followed by Appx2; the fastest to query are Appx1 and Appx2, then Appx2+; however, Appx1 and Appx2+ have better approximation quality than Appx2 (as shown in later experiments and as suggested by their theoretical guarantees for Appx1).

3.5.4 Scalability

Next, we investigate the scalability of different methods, using all three exact methods and the three selected approximate methods, when we vary the number of objects m, and the average number of line segments per object n_{avg}, in the *Temp* dataset. Figures 3.12, 3.13, and 3.14 show the results. In general, the trends are very consistent and agree with our theoretical analysis. All exact methods consume linear space $O(N)$ and take $O(N \log N)$ time to build. Exact3 is clearly the overall best exact method in terms of query costs, outperforming the other two by 2-3 orders of magnitude in terms of IOs and query time

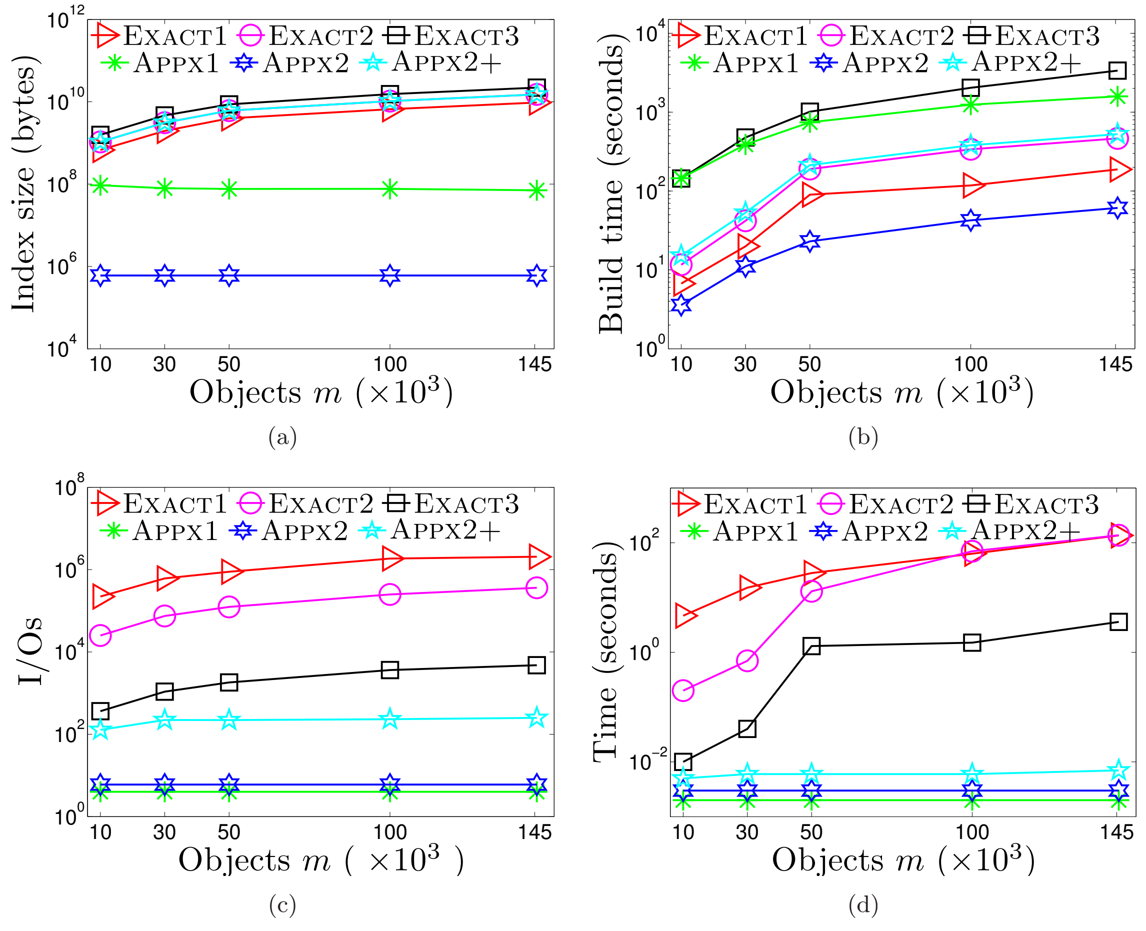
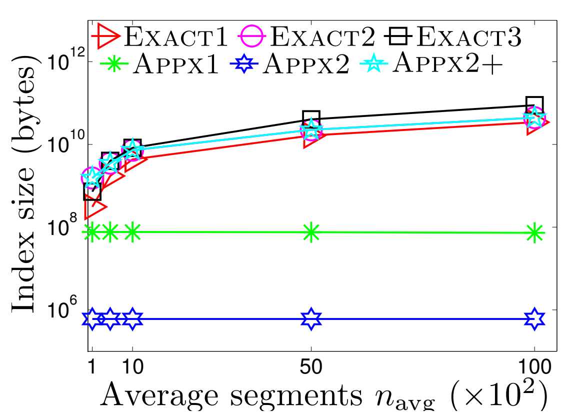
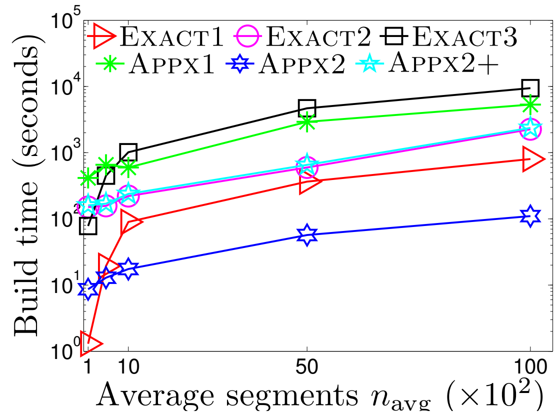


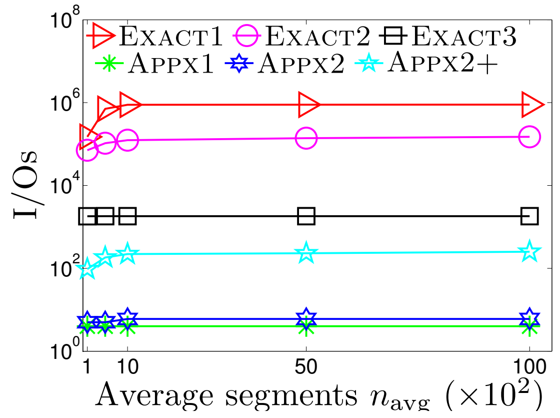
Figure 3.12. Vary number of objects m on Temp: versus (a) index size, (b) build time, (c) query I/Os, (d) query time.



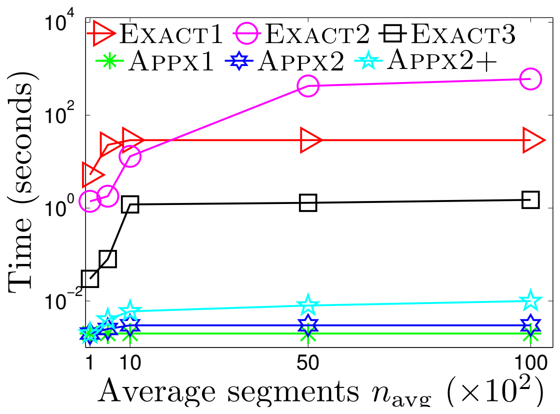
(a)



(b)

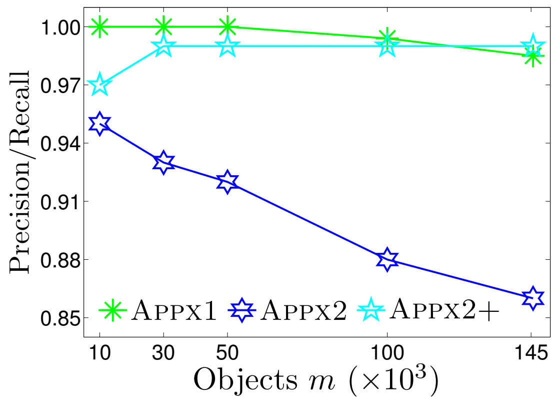


(c)

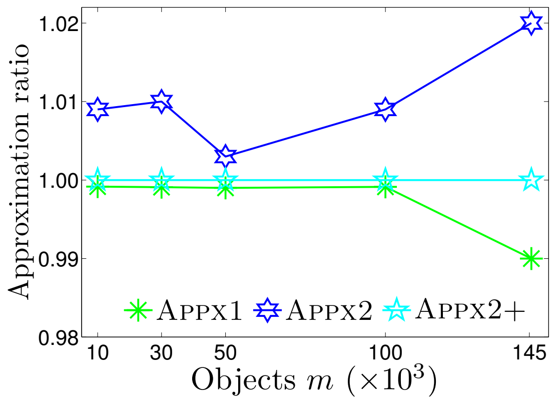


(d)

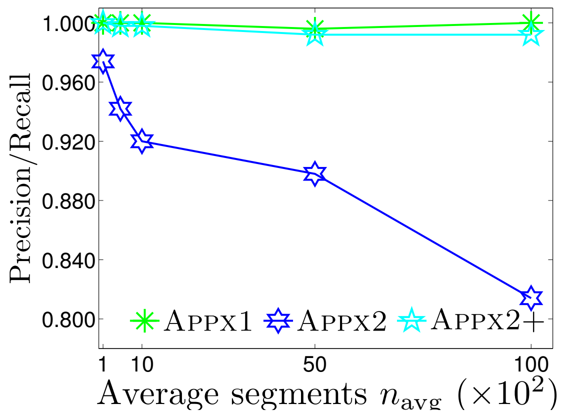
Figure 3.13. Vary average number of segments n_{avg} on Temp: versus (a) index size, (b) build time, (c) query I/Os, (d) query time.



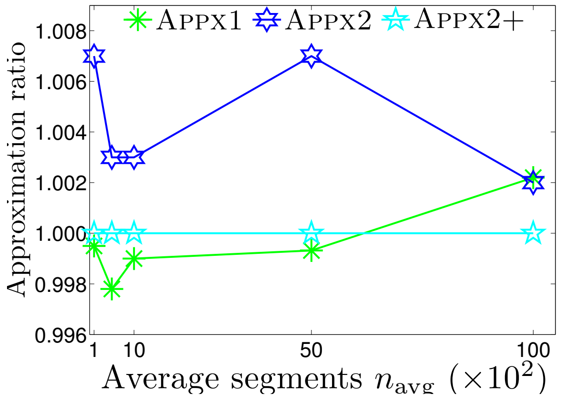
(a)



(b)



(c)



(d)

Figure 3.14. Approximation quality for Temp: m versus (a) precision/recall and (b) ratio; n_{avg} versus (c) precision/recall and (d) ratio.

(even though it costs slightly more to build). In general, Exact3 takes hundreds to a few thousand IOs, and about 1 to a few seconds to answer an aggregate top-$k(t_1, t_2, \text{sum})$ query in the *Temp* dataset (with a few hundred million segments from 145,628 objects). Its query performance is not clearly affected by n_{avg}, but has a linear dependency on m.

The approximate methods consistently beat the best exact algorithm in query performance by more than 2 orders of magnitude in terms of running time. Even on the largest dataset with a few hundred million segments from 145,628 different objects, they still take less than 0.01 seconds per query! Among the three, Appx1 and Appx2 clearly take fewer IOs, since their query cost is actually independent of both m and n_{avg}. Appx2+'s query IO does depend on $\log n_{\text{avg}}$, but is independent of m; hence, it is still very small. Appx1 (and even more so Appx2+) occupy much more space, and take much longer to build. Nevertheless, both Appx1 and Appx2 have much smaller index size than Exact3, by 4 (Appx1) and 6 (Appx2) orders of magnitude, respectively. More importantly, their index size is independent of both m and n. In terms of the construction cost, Appx2-B is the most efficient to build (1-2 orders of magnitude faster than all other methods except Appx2).

Figure 3.14 shows that both Appx1 and Appx2+ retain their high approximation quality when m or n_{avg} vary; despite some fluctuation, precision/recall and approximation ratios in both Appx1 and Appx2+ stay very close to 1. Appx2 remains at an acceptable level of accuracy, especially considering the index size is 1MB from 50GB of data! Although the precision/recall drops as n_{avg} and m increases, the very accurate approximation ratio indicates this is because there are many *very* similar objects.

3.5.5 Query time interval

Based on our cost analysis, clearly, the length of the query time interval does not affect the query performance of most of our methods, except for Exact1 that has a linear dependency on $(t_2 - t_1)$ (since it has to scan more line segments). In Figure 3.15(a) and 3.15(b), we notice Exact1 has a linear increase in both I/Os and running time (note the log-scale of the plots) and even for small ($2\%T$) query intervals, it is still much slower than Exact3 and approximate methods.

In Figures 3.15(c) and 3.15(d), we analyze the quality of all approximation techniques as the query interval increases. Appx1 and Appx2+ clearly have the best precision/recall and approximation ratio with a precision/recall above 99% and ratio very close to 1 in all cases. Appx2 shows a slight decline in precision/recall from roughly 98% to above 90% as the size of $(t_2 - t_1)$ increases from 2% to 50% of the maximum temporal value T. This decrease in precision/recall is reasonable since as we increase $(t_2 - t_1)$, the number of dyadic

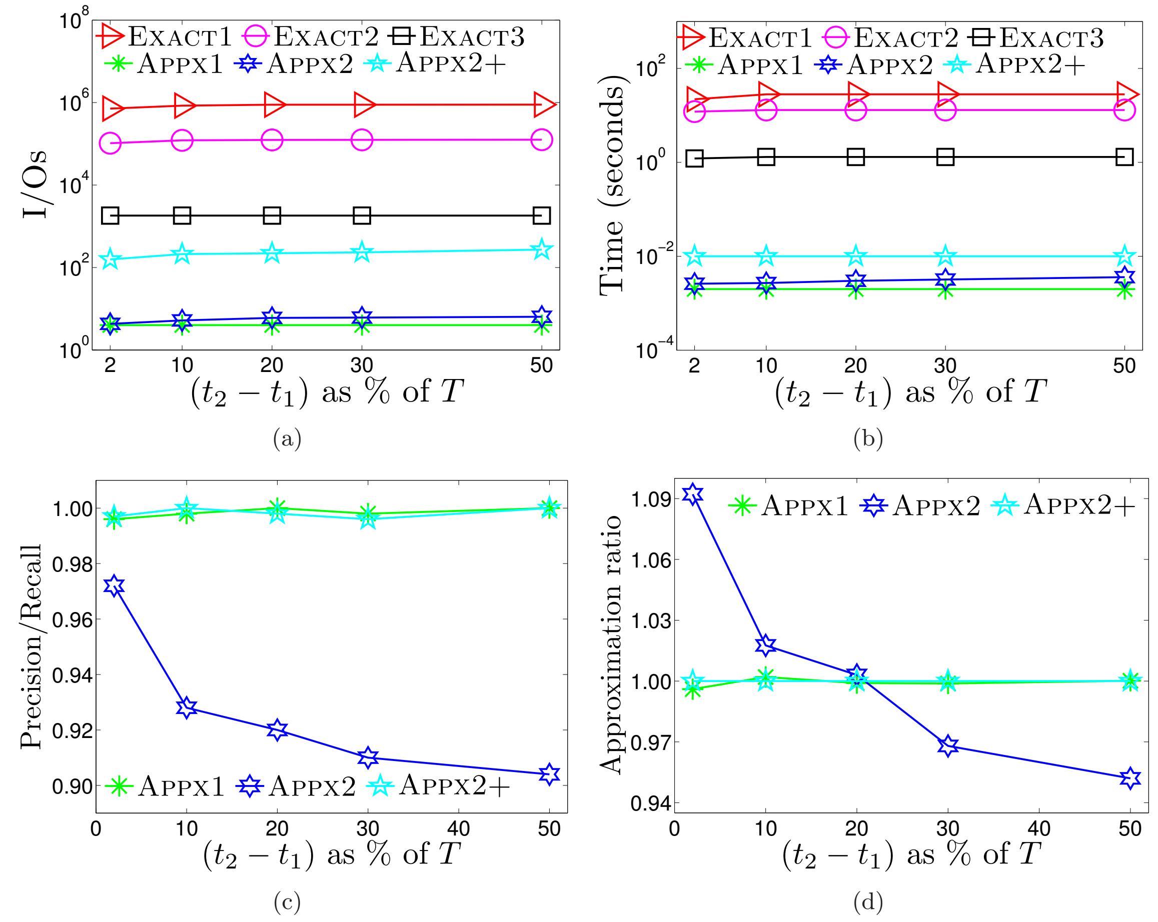


Figure 3.15. Vary size of $(t_2 - t_1)$ as % of T on Temp: effect on (a) query I/Os, (b) query time, (c) precision/recall, (d) ratio.

intervals that compose the approximate query interval $[\mathcal{B}(t_1), \mathcal{B}(t_2)]$ typically increases. As the number of dyadic intervals increases, there is an increased probability that not every candidate in $\mathcal{K}$ will be in the top-k_{max} over each of the dyadic intervals and so APPX2 will be missing some of a candidate's aggregate scores. This can cause an item to be falsely ejected from the top k. The effect of missing aggregate scores is clearly seen in Figure 3.15(d), which shows that APPX2's approximation ratio drops slightly as the time range increases.

3.5.6 k and $k_{\max}$

We studied the effect of k and $k_{\max}$; the results are shown in Figures 3.16 and 3.17. Figures 3.16(a) and 3.16(b) show that the query performance of most methods is not affected by the value of k when it changes from 10 to $k_{\max} = 200$ (a relatively small to moderate change w.r.t. the database size) except for APPX2 and APPX2+. This results since larger k values lead to more candidates in $\mathcal{K}$, which results in higher query cost. Nevertheless,

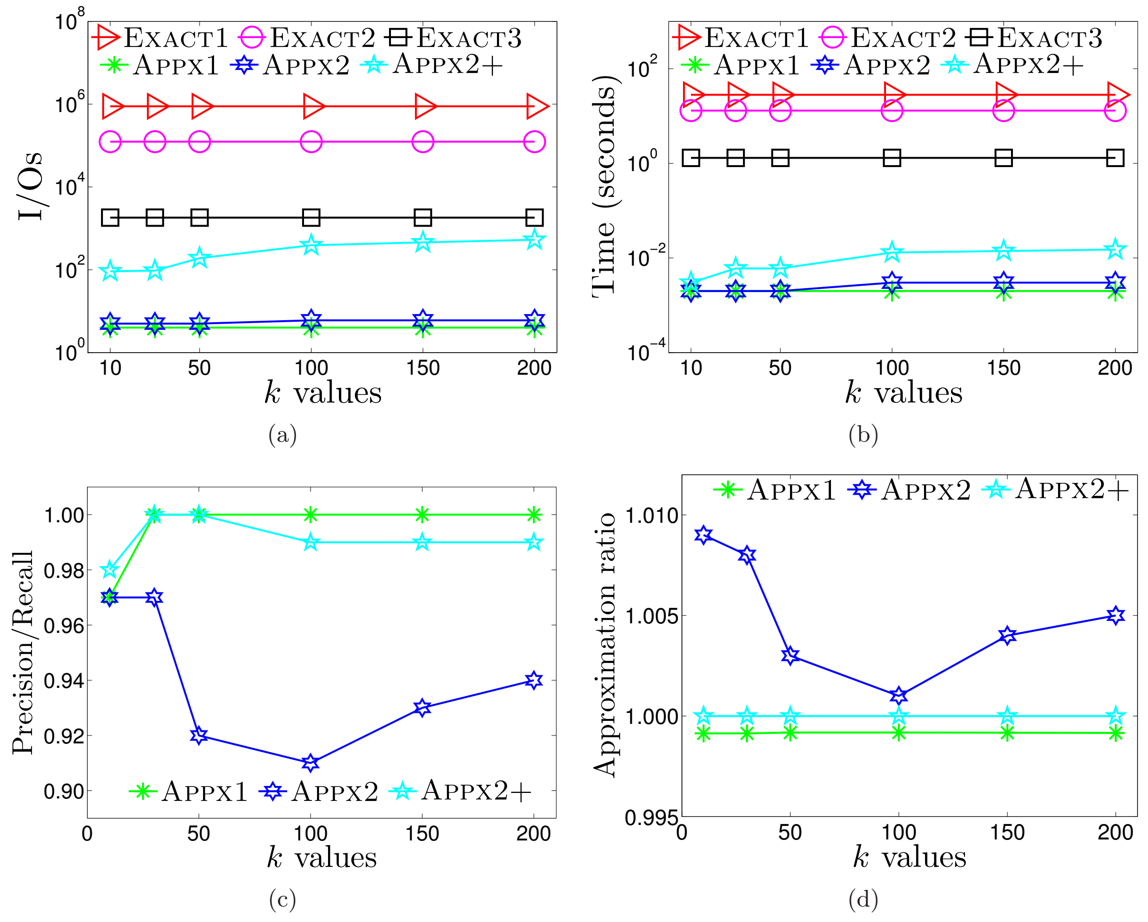


Figure 3.16. Vary k values on Temp: effect on (a) query I/Os, (b) query time, (c) precision/recall, (d) ratio.

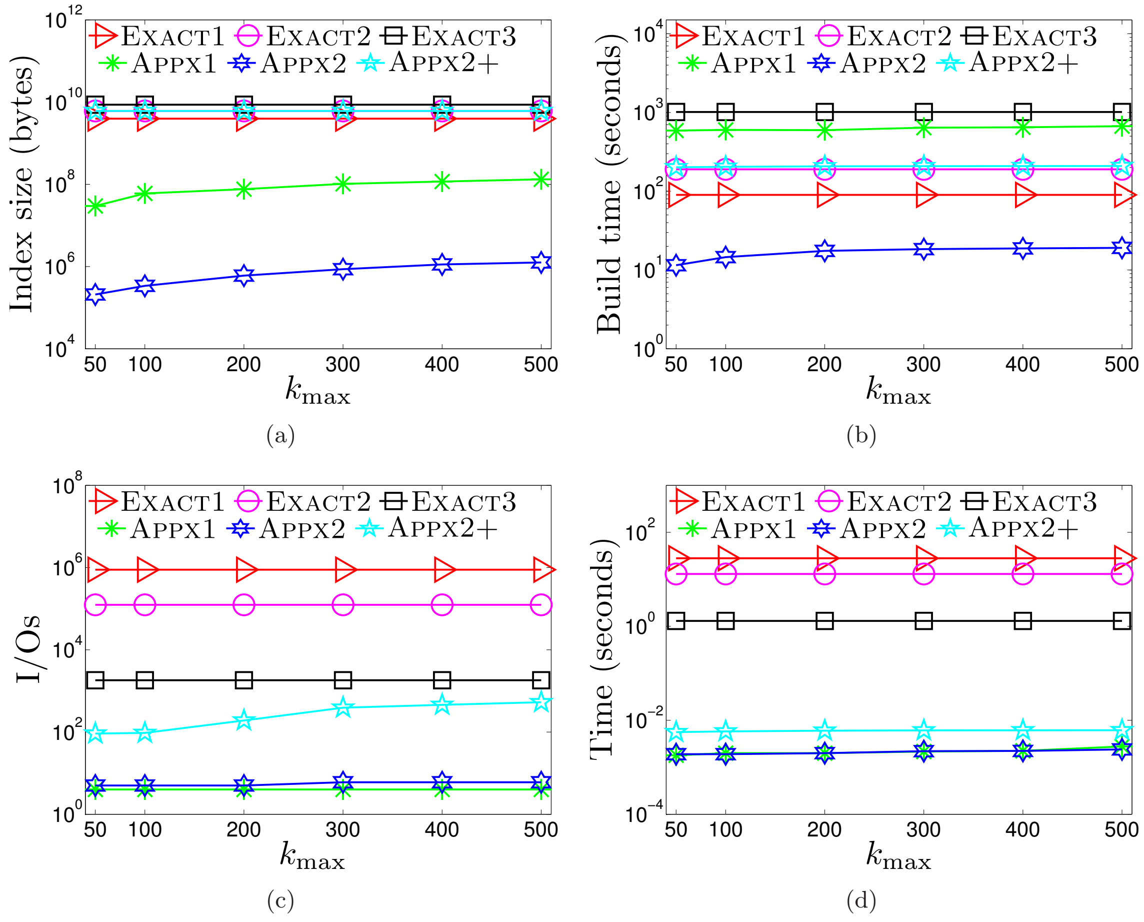


Figure 3.17. Vary $k_{\max}$ on Temp: effect on (a) index size, (b) construction time, (c) query I/Os, (d) query time.

they still have better IOs than the best exact method Exact3, and much better query cost (still 2 orders of magnitude improvement in the worst case, which can be attributed to the caching effect by the OS). Figure 3.16(c) and 3.16(d) show some fluctuation, but no trending changes in accuracy due to variation in k.

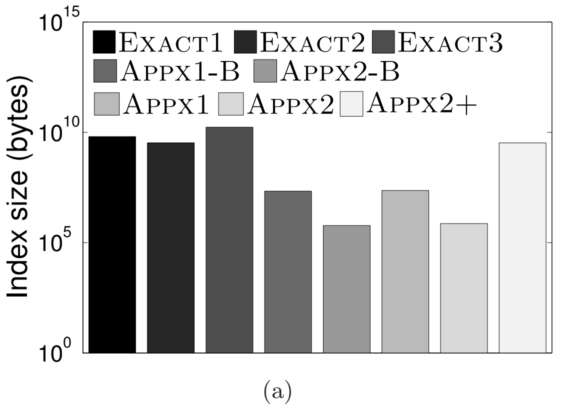
We vary $k_{\max}$ from 50 to 500 in Figure 3.17. $k_{\max}$ obviously has no effect on exact methods. It linearly affects the construction cost and the size of index for Appx1 and Appx2, but they are still much better than exact methods even when $k_{\max} = 500$. In terms of query cost, given the same k values, $k_{\max}$ does not clearly affect any approximate methods when it only changes moderately w.r.t. the database size.

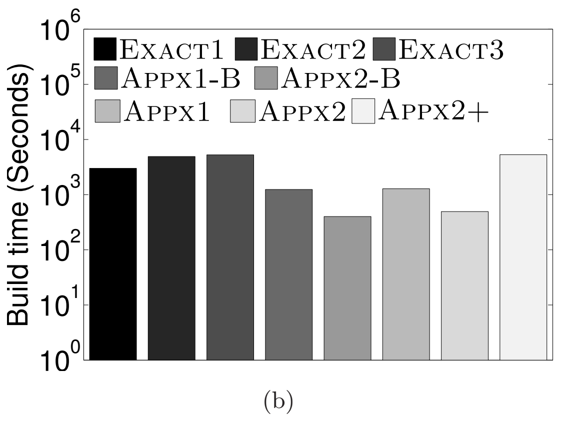
3.5.7 Updates

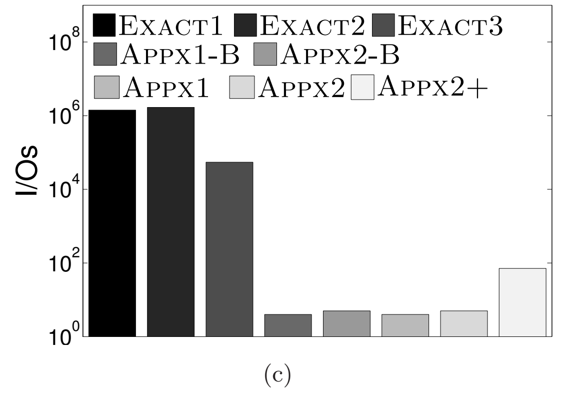
As suggested by the cost analysis, the update time for each index structure is roughly proportional to the build time divided by the number of segments. Relative to these build times over N, however, Exact1 is slower because it cannot bulk load, and Exact2 and Appx2+ are faster because they only update a single B+-tree. For space, we omit these results.

3.5.8 Meme dataset

We have also tested all our methods on the *full Meme* dataset (still using $r = 500$ breakpoints for all approximate methods), and the results are shown in Figure 3.18. In terms of the index size, three exact methods (and Appx2+) are comparable, as seen in Figure 3.18(a), while other approximate methods take much less space, by 3-5 orders of magnitude! In terms of the construction cost, it is interesting to note that Exact1 is the fastest to build in this case, due to the bulk-loading algorithm in the B+-tree (since all segments are sorted), while all other methods have some dependency on m. However, approximate methods (excluding Appx2+) generally are much faster to build than other exact methods, as seen in Figure 3.18(b). They also outperform all exact methods by 3-5 orders of magnitude in IOs in Figure 3.18(c) and 3-4 orders of magnitude in running time in Figure 3.18(d). The best exact method for queries is still Exact3, which is faster than the other two exact methods by 1-2 orders of magnitude. Finally, all approximate methods maintain their high (or acceptable for Appx2) approximation quality on this very bursty dataset, as seen in Figure 3.19. Note Appx2 achieves this 90% precision/recall and close to 1 approximation ratio while compressing to about 1MB. Also, Appx1 and Appx2 using $\mathcal{B}_2$ show better results than their basic versions Appx1-B and Appx2-B using $\mathcal{B}_1$, given the same number of breakpoints, which agrees with the trend from the *Temp* dataset.







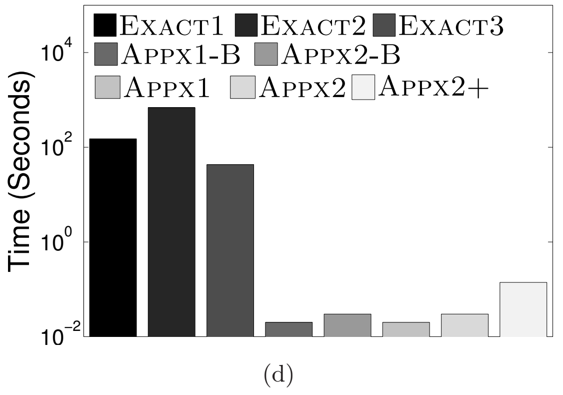
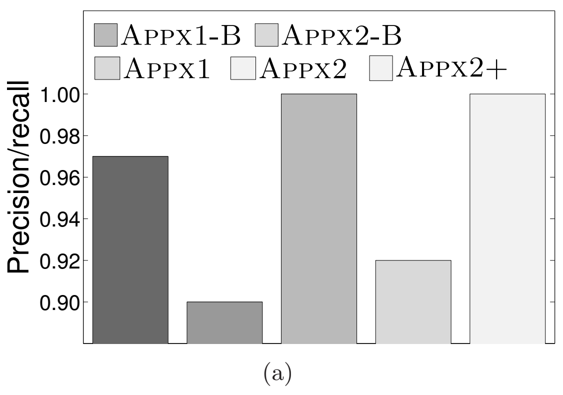


Figure 3.18. Meme dataset evaluation: observed (a) index size, (b) build time, (c) I/Os, (d) query time.



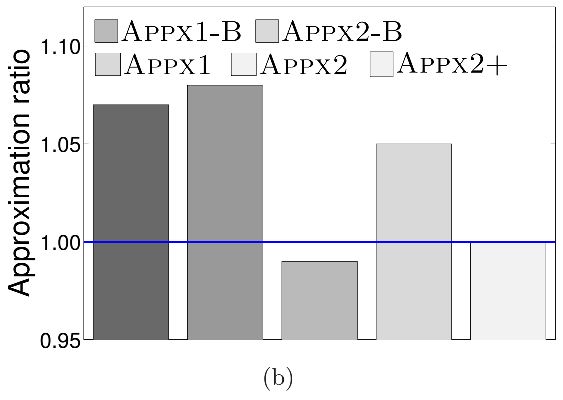


Figure 3.19. Quality of approximations on Meme: observed (a) precision/recall and (b) approximation ratio.

3.6 Related Work

To the best of our knowledge, ranking temporal data based on their aggregation scores in a query interval has not been studied before. Ranking temporal data based on the instant top-k definition has been recently studied in [51]; however, as we have pointed out in Section 3.1, one cannot apply their results in our setting. In another work on ranking temporal data [57], they retrieve k objects that are always amongst the top-k list at every time instance over a query time interval. Clearly, this definition is very restrictive and may not even have k objects satisfying this condition in a query interval. This could be relaxed to require an object to be in the top-k list at *most time instances* of an interval, instead of at *all time instances*, like the intuition used in finding durable top-k documents [68], but this has yet to be studied in time series/temporal data. Even then, ranking by aggregation scores still offers quite different semantics, is new, and, is useful in numerous applications.

Our study is related to work on temporal aggregation [62, 63]. As mentioned in Section 3.2, [62, 63] focus on multiversioned keys (instead of time series data), and their objective is to *compute a single aggregation of all keys alive in a query time interval and/or a query key range*, which is different from our definition of aggregation, which is to compute an aggregation over a query time interval, *one per object* (then rank objects based on their aggregation values).

Approximate versions of [62, 63] were presented in Tao *et al.* [69, 70], which also leveraged on a discretization approach (the general principle behind the construction of our breakpoints). As their goal is to approximate aggregates over all keys alive in any query rectangles over the time and the key dimensions (a single aggregate per query rectangle), instead of time-aggregates over each element individually, their approach is not appropriate for our setting.

Our methods require the segmentation of time series data, which has been extensively studied, and the general principles appear in Section 3.1. A more detailed discussion of this topic is beyond the scope of this chapter and we refer interested readers to [52–54, 56, 66].

3.7 Conclusion

We have studied one of the emerging challenges with massive data in this chapter, namely the *complex structure and rich semantics* of temporal data, and have shown how to use the *ranking operator* to summarize the data to the temporal objects with the top-k aggregate values over a query interval, which has numerous applications. Our best exact method EXACT3 is much more efficient than baseline methods, and our approximate methods, which themselves are also data summaries, offer further improvements. Interesting open problems include ranking with holistic aggregations and extending to the distributed setting.

CHAPTER 4

RANKING SEMANTICS FOR PROBABILISITC DATA

4.1 Introduction

In this chapter, we look at the emerging problem of *uncertain data* and its semantics. We observe numerous *ranking operators* that have been proposed which lack many intuitive properties that hold over certain data, such as containment, exact-k, value invariance, etc. Given this shortcoming, we propose novel ranking operators to summarize uncertain data which satisfy all of these intuitive properties.

Ranking queries are a powerful concept in focusing attention on the most important answers to a query. To deal with massive quantities of data, such as multimedia search, streaming data, web data, and distributed systems, tuples from the underlying database are ranked by a score, usually computed based on a user-defined scoring function. Only the top-k tuples with the highest scores are returned for further inspection. Following the seminal work by Fagin *et al.* [72], such queries have received considerable attention in traditional relational databases, including [73–75] and many others. See the excellent survey by Ilyas *et al.* [50] for a more complete overview of the many important studies in this area.

Within these motivating application domains—distributed, streaming, web, and multimedia applications—data arrive in massive quantities, underlining the need for ordering by score. However, an additional challenge is that data are typically inherently fuzzy or uncertain. For instance, multimedia and unstructured web data frequently require data integration or schema mapping [76–78]. Data items in the output of such operations are usually associated with a confidence, reflecting how well they are matched with other records from different data sources. In applications that handle measurement data, e.g., sensor

*The work in this chapter appears in (J. Jestes, G. Cormode, F.Li and K. Yi, "Semantics of ranking queries for probabilistic data," IEEE TKDE, vol. 23, pp. 1903-1917, 2011) ©2011 IEEE [71] readings and distances to a query point, the data are inherently noisy, and are better represented by a probability distribution rather than a single deterministic value [79,80]. In recognition of this aspect of the data, there have been significant research efforts devoted to producing *probabilistic database management systems*, which can represent and manage data with explicit probabilistic models of uncertainty. Some notable examples of such systems include MystiQ [81], Trio [82], Orion [83], and MayBMS [84].

With a probabilistic database, it is possible to compactly represent a huge number of possible (deterministic) realizations of the (probabilistic) data—an exponential blow-up from the size of the relation representing the data. A key problem in such databases is how to extend the familiar semantics of the top-k query to this setting, and how to answer such queries efficiently. To this end, there have been several recent works outlining possible definitions, and associated algorithms. Ré *et al.* [85] base their ranking on the confidence associated with each query result. Soliman *et al.* [86] extend the semantics of ranking queries from certain data and study the problem of ranking tuples when there is both a score and probability for each tuple. Subsequently, there have been several other approaches to ranking based on combining score and likelihood [87–90] (discussed in detail in Section 4.4.4).

For certain data with a single score value, there is a clear total ordering based on their scores from which the top-k is derived, which leads to clean and intuitive semantics. This is particularly natural, by analogy with the many occurrences of top-k lists in daily life: movies ranked by box-office receipts, athletes ranked by race times, researchers ranked by number of publications (or other metrics), and so on. With uncertain data, there are two distinct orders to work with: ordering by score, and ordering by probability. There are many possible ways of combining these two, leading to quite different results, as evidenced by the multiple definitions which have been proposed in the literature, such as U-Top$k$ [86], U-kRanks [86], Global-Top$k$ [87], and PT-k [90]. In choosing a definition, we must ask, what conditions do we want the resulting query answer to satisfy. We address this following a principled approach, returning to ranking query properties on certain data. We provide the following properties which are desirable on the output of a ranking query as a minimum:

- *Exact-k:* The top-k list should contain exactly k items;
- *Containment:* The top-$(k+1)$ list should contain all items in the top-k;
- *Unique-ranking:* Within the top-k, each reported item should be assigned exactly one position: the same item should not be listed multiple times within the top-k.

- *Stability:* Making an item in the top-k list more likely or more important should not remove it from the list.

- *Value-invariance:* The scores only determine the relative behavior of the tuples: changing the score values without altering the relative ordering should not change the top-k;

We define these properties more formally in Section 4.4.1. These properties are satisfied for certain data, and capture much of our intuition on how a "ranking" query should behave. A general axiom of work on extending data management from certain data to the uncertain domain has been that basic properties of query semantics should be preserved to the best extent possible [81, 91]. However, as we demonstrate, none of the prior works on ranking queries for probabilistic data has systematically examined these properties and studied whether a ranking definition satisfies them. It should be noted these five ranking properties are by no means a complete characterization for ranking uncertain data. Nevertheless, it is an interesting and important problem to search for meaningful definitions that satisfy at least these properties.

Lastly, we note prior work stated results primarily in the *tuple-level* model [81,82]; here, we show results for both *tuple-level* and *attribute-level* models [79, 92].

4.1.1 Our contributions

To remedy the shortcomings we identify, we propose an intuitive new approach for ranking based on the rank distribution of a tuple's ranks across all possible worlds. Using this well-founded rank distribution as the basis of the ranking, we study ranking definitions based on typical statistical values over a distribution. Specifically,

- We formalize some important semantics of ranking queries on certain data and migrate them to probabilistic data (Section 4.4.1), and systematically examine the characteristics of existing approaches for this problem with respect to these properties (Section 4.4.4).

- We propose a new approach based on the distribution of each tuple's ranks across all possible worlds. By leveraging statistical properties on such a rank distribution, such as the expectation, the median, and the quantile, we derive the expected rank, the median rank, and quantile rank. We are able to show that the new definitions provably satisfy these requirements. These new definitions work seamlessly with both the *attribute-level* and *tuple-level* uncertainty models (Section 4.4.5).

- We provide efficient algorithms for expected ranks in both models. For an uncertain relation of N constant-sized tuples, the processing cost of expected ranks is $O(N \log N)$ for both models. In settings where there is a high cost for accessing tuples, we show pruning techniques based on probabilistic tail bounds that can terminate the search early and guarantee that the top-k has been found (Section 4.5 and 4.6).

- We study additional properties guaranteed by median and quantile ranks and present dynamic programs for computing them. The formulations are different in the *attribute-level* and the *tuple-level* models; however, they are similar for the median and different quantile values. For an uncertain relation of N tuples, the processing cost of our algorithm is $O(N^3)$ in the *attribute-level* model, and $O(NM^2)$ in the *tuple-level* model where M is the number of rules in the database (Section 4.7).

- We discuss other issues related to this work in (Section 4.8), e.g., continuous functions, further properties of a ranking, and the interesting relationship between our study and [93] which proposes a general framework for imposing different ranking definitions in probabilistic data.

- We present a comprehensive experimental study that confirms the effectiveness of our approach for various ranking definitions (Section 4.9).

4.2 Background

Much effort has been devoted to modeling and processing uncertain data, so we survey only the most related work. TRIO [82,91,94], MayBMS [84], Orion [83,95], and MystiQ [81] are promising systems currently being developed. General query processing techniques have been extensively studied under the possible worlds semantics [79, 81, 96, 97], and important query types with specific semantics are explored in more depth, skyline queries [98] and heavy hitters [99]. Indexing and nearest neighbor queries under the attribute-level model have also been explored [79, 92, 100–103].

Section 4.4.4 discusses the most closely related works on answering top-k queries on uncertain databases [86, 87, 89, 90]. Techniques have included the Monte Carlo approach of sampling possible worlds [85], AI-style branch-and-bound search of the probability state space [86], dynamic programming approaches [87, 89, 104], and applying tail (Chernoff) bounds to determine when to prune [90]. There is ongoing work to understand semantics of top-k queries in a variety of contexts. For example, the work of Lian and Chen [105] deals with ranking objects based on spatial uncertainty, and ranking based on linear functions.

Ge *et al.* [106] presented a detailed study on finding the typical vectors that effectively sample the score distribution from the top-k query results in uncertain databases.

Our study on the tuple-level model limits us to considering correlations in the form of mutual exclusions. More advanced rules and processing may be needed for complex correlations. Recent works based on graphical probabilistic models and Bayesian networks have shown promising results in both offline [107] and streaming data [108]. In these situations, initial approaches are based on Monte-Carlo simulations [85, 97].

4.3 Uncertain Data Models

Many models for describing uncertain data have been presented in the literature. The work by Das Sarma *et al.* [94] describes the main features and contrasts their properties and descriptive ability. Each model describes a probability distribution over *possible worlds*, where each possible world corresponds to a single deterministic data instance. The most expressive approach is to explicitly list each possible world and its associated probability; such a method is referred to as *complete*, as it can capture all possible correlations. However, complete models are very costly to describe and manipulate since there can be exponentially many combinations of tuples each generating a distinct possible world [94].

Typically, we are able to make certain *independence assumptions*, that unless correlations are explicitly described, events are assumed to be independent. Consequently, likelihoods can be computed using standard probability calculations (i.e., multiplication of probabilities of independent events). The strongest independence assumptions lead to the *basic model*, where each tuple has a probability of occurrence, and all tuples are assumed fully independent of each other. This is typically too strong an assumption, and so intermediate models allow the description of simple correlations between tuples. This extends the expressiveness of the models, while keeping computations of probability tractable. We consider two models that have been used frequently within the database community. In our discussion, without loss of generality, a probabilistic database contains simply one relation.

4.3.1 Attribute-level uncertainty model

In this model, the probabilistic database is a table of N tuples. Each tuple has one attribute whose value is uncertain (together with other certain attributes). This uncertain attribute has a (finite) discrete pdf describing its value distribution. When instantiating this uncertain relation to a certain instance, each tuple draws a value for its uncertain attribute based on the associated discrete pdf and the choice is independent among tuples. This model has many practical applications such as sensor readings [80, 108], spatial objects with fuzzy locations [79, 92, 100, 102, 103], etc. More important, it is very easy to represent this model using the traditional relational database model, as observed by Antova *et al.* [109]. For the purpose of ranking queries, the important case is when the uncertain attribute represents the score for the tuple, and we would like to rank the tuples based on this score attribute. Let X_i be the random variable denoting the score of tuple t_i. We assume that X_i has a discrete pdf with bounded size s_i. This is a realistic assumption for many practical applications, including movie ratings [81], and string matching [77]. In this model, we are essentially ranking the set of independent random variables $X_1, \dots, X_N$. A relation following this model is illustrated in Tables 4.1 and 4.2. For tuple t_i, the score takes the value $v_{i,j}$ with probability $p_{i,j}$ for $1 \le j \le s_i$.

4.3.2 Tuple-level uncertainty model

In the second model, the attributes of each tuple are fixed, but the entire tuple may or may not appear. In the basic model, each tuple t appears with probability $p(t)$ independently. In more complex models, there are dependencies among the tuples, which can be specified by a set of *generation rules*. These can be in the form of *x-relations* [82, 91], complex events [81], or other forms.

All previous work concerned with ranking queries in uncertain data has focused on the tuple-level uncertainty model with *exclusion rules* [86, 87, 89, 90] where each tuple appears

Table 4.1. Attribute-level uncertainty model. ©2011 IEEE

tuples	score
t_1	$\{(v_{1,1}, p_{1,1}), (v_{1,2}, p_{1,2}), \dots, (v_{1,s_1}, p_{1,s_1})\}$
t_2	$\{(v_{2,1}, p_{2,1}), \dots, v_{2,s_2}, p_{2,s_2})\}$
$\vdots$	$\vdots$
t_N	$\{(v_{N,1}, p_{N,1}), \dots, (v_{N,s_N}, p_{N,s_N})\}$

Table 4.2. An example of possible worlds for attribute-level uncertainty model. ©2011 IEEE

tuples	score
t_1	$\{(100, 0.4), (70, 0.6)\}$
t_2	$\{(92, 0.6), (80, 0.4)\}$
t_3	$\{(85, 1)\}$

world W	$\Pr[W]$
$\{t_1 = 100, t_2 = 92, t_3 = 85\}$	$0.4 \times 0.6 \times 1 = 0.24$
$\{t_1 = 100, t_3 = 85, t_2 = 80\}$	$0.4 \times 0.4 \times 1 = 0.16$
$\{t_2 = 92, t_3 = 85, t_1 = 70\}$	$0.6 \times 0.6 \times 1 = 0.36$
$\{t_3 = 85, t_2 = 80, t_1 = 70\}$	$0.6 \times 0.4 \times 1 = 0.24$

in a single rule τ. Each rule τ lists a set of tuples that are mutually exclusive so that at most one of these can appear in any possible world. Arbitrary generation rules have been discussed in [86, 88], but they have been shown to require exponential processing complexity [89, 90]. Hence, as with many other works in the literature [86, 89, 90, 99], we primarily consider exclusion rules in this model, where each exclusion rule has a constant number of choices. In addition, each tuple appears in at most one rule. The total probability for all tuples in one rule must be less or equal to one, so that it can be properly interpreted as a probability distribution. To simplify our discussion, we allow rules containing only one tuple and require that all tuples appear in (exactly) one of the rules. This is essentially equivalent to the popular x-relations model [82]. This tuple-level model is a good fit for applications where it is important to capture correlations between tuples; this model has been used to fit a large number of real-life examples [81,86,90,91,99]. Examples of a relation in this model are shown in Tables 4.3 and 4.4. This relation has N tuples and M rules. The second rule says that t_2 and t_4 cannot appear together in any certain instance of this relation. It also constrains that $p(t_2) + p(t_4) \leq 1$.

4.3.3 The possible world semantics

We denote the uncertain relation as $\mathcal{D}$. In the attribute-level uncertainty model, an uncertain relation is instantiated into a *possible world* by taking one independent value for each tuple's uncertain attribute according to its distribution. Denote a possible world as W and the value for t_i's uncertain attribute in W as w_{t_i}. In the attribute-level uncertainty

Table 4.3. Tuple-level uncertainty model. ©2011 IEEE

tuples	score	$p(t)$
t_1	v_1	$p(t_1)$
t_2	v_2	$p(t_2)$
$\vdots$	$\vdots$	
t_N	v_N	$p(t_N)$

rules	
τ_1	$\{t_1\}$
τ_2	$\{t_2, t_4\}$
$\vdots$	$\vdots$
τ_M	$\{t_5, t_8, t_N\}$

Table 4.4. An example of possible worlds for tuple-level uncertainty model. ©2011 IEEE

tuples	score	$p(t)$
t_1	100	0.4
t_2	92	0.5
t_3	80	1
t_4	70	0.5

rules	
τ_1	$\{t_1\}$
τ_2	$\{t_2, t_4\}$
τ_3	$\{t_3\}$

world W	$\Pr[W]$
$\{t_1, t_2, t_3\}$	$p(t_1)p(t_2)p(t_3) = 0.2$
$\{t_1, t_3, t_4\}$	$p(t_1)p(t_3)p(t_4) = 0.2$
$\{t_2, t_3\}$	$(1 - p(t_1))p(t_2)p(t_3) = 0.3$
$\{t_3, t_4\}$	$(1 - p(t_1))p(t_3)p(t_4) = 0.3$

model, the probability that W occurs is $\Pr[W] = \prod_{j=1}^{N} p_{j,x}$, where x satisfies $v_{j,x} = w_{t_j}$. It is worth mentioning that in the attribute-level case, we always have $\forall W \in \mathcal{W}, |W| = N$, where $\mathcal{W}$ is the space of all the possible worlds. The example in Table 4.2 illustrates the possible worlds for an uncertain relation in this model.

For the tuple-level uncertainty model, a possible world W from $\mathcal{W}$ is now a *subset* of tuples from the uncertain relation $\mathcal{D}$. The probability of W occurring is $\Pr[W] = \prod_{j=1}^{M} p_W(\tau_j)$, where for any rule τ that applies to $\mathcal{D}$, $p_W(\tau)$ is defined as

$$p_W(\tau) = \begin{cases} p(t), & \text{if } \tau \cap W = \{t\}; \\ 1 - \sum_{t_i \in \tau} p(t_i), & \text{if } \tau \cap W = \emptyset; \\ 0, & \text{otherwise.} \end{cases}$$

A notable difference for the tuple-level uncertain model is that given a random possible world W, not all tuples from $\mathcal{D}$ will appear. Hence, the size of the possible world can range from 0 to N. The example in Table 4.4 illustrates the possible worlds for an uncertain relation in this model.

We iterate that every uncertain data model can be seen as a succinct description of a distribution over possible worlds $\mathcal{W}$. Each possible world is a certain table on which we can evaluate any traditional query. The focus of uncertain query processing is (1) how to "combine" the query results from all the possible worlds into a meaningful result for the query, and (2) how to process such a combination efficiently without explicitly materializing the exponentially many possible worlds.

4.3.4 Difference of the two models under ranking queries

We emphasize that there is a significant difference for the two models in the context of *ranking tuples*. More specifically, the goal of ranking queries in uncertain databases is to derive a meaningful ordering for all tuples in the database $\mathcal{D}$. Note that this is not equivalent to deriving an ordering for all values that tuples in $\mathcal{D}$ may take. In the attribute-level model, all tuples in $\mathcal{D}$ will participate in the ranking process in every possible world. In contrast, in the tuple-level model, only a subset of tuples in $\mathcal{D}$ will participate in the ranking process for a given possible world. In particular, although there are mappings between relations in the attribute-level and tuple-level models, these have different sets of tuples to rank (often, with different cardinalities). As such, this means that there is no simple reduction between the two cases, and different algorithmic solutions are needed for each. It remains a tantalizing prospect to make further use of structural similarities between the two models to design a unified approach for ranking in both. We hope this can be addressed in future work.

4.4 Ranking Query Semantics

4.4.1 Properties of ranking queries

We now define a set of properties for ranking tuples. These are chosen to describe key properties of ranking certain data, and hence give properties which a user would naturally expect of a ranking over uncertain data. These properties should be seen largely as *desirable* but by no means *sufficient* for a ranking. Our main purpose in introducing them is to make them explicit, demonstrate prior definitions do not adhere to them all, and provoke discussion about which properties should hold in general for proposed ranking methods.

The first property is very natural, and is also used in [87].

Definition 4.1 (Exact-k). *Let R_k be the set of tuples (associated with their ranks) in the top-k query result. If $|\mathcal{D}| \geq k$, then $|R_k| = k$.*

The second property captures the intuition that if an item is in the top-k, it should be in the top-k' for any $k' > k$. Equivalently, the choice of k is simply a slider that chooses how many results are to be returned to the user, and changing k should only change the number of results returned, not the underlying set of results.

Definition 4.2 (Containment). *For any k, $R_k \subset R_{k+1}$.*

Replacing "$\subset$" with "$\subseteq$" gives the *weak containment* property.

The next property stipulates that the rank assigned to each tuple in the top-k list should be unique.

Definition 4.3 (Unique ranking). *Let $r_k(i)$ be the identity of the tuple from the input assigned rank i in the output of the ranking procedure. The* unique ranking *property requires that $\forall i \neq j, r_k(i) \neq r_k(j)$.*

Zhang and Chomicki [87] proposed the *stability* condition in the tuple-level uncertainty model. We adopt this property and generalize it to the attribute-level model:

Definition 4.4 (Stability). *In the tuple-level model, given a tuple $t_i = (v_i, p(t_i))$ from $\mathcal{D}$, if we replace t_i with $t_i^{\uparrow} = (v_i^{\uparrow}, p(t_i^{\uparrow}))$ where $v_i^{\uparrow} \geq v_i, p(t_i^{\uparrow}) \geq p(t_i)$, then*

$$t_i \in R_k(\mathcal{D}) \Rightarrow t_i^{\uparrow} \in R_k(\mathcal{D}'),$$

where $\mathcal{D}'$ is obtained by replacing t_i with $t_i^{\uparrow}$ in $\mathcal{D}$.

For the attribute-level model, the statement for stability remains the same but with $t_i^{\uparrow}$ defined as follows. Given a tuple t_i whose score is a random variable X_i, we obtain $t_i^{\uparrow}$ by *replacing* X_i *with a random variable* $X_i^{\uparrow}$ *that is* stochastically greater than or equal to *[110]* X_i*, denoted as* $X_i^{\uparrow} \succeq X_i$*, meaning* $\Pr(X_i^{\uparrow} \geq x) \geq \Pr(X_i \geq x)$ *for all* $x \in (-\infty, \infty)$.

Stability captures the intuition that if a tuple is already in the top-k, making it "probabilistically larger" should not eject it. Stability also implies that making a tuple not in the top-k probabilistically smaller should not bring it into the top-k.

The final property captures the semantics that the score function is assumed to only give a relative ordering, and is not an absolute measure of the value of a tuple.

Definition 4.5 (Value invariance)**.** *Let* $\mathcal{D}$ *denote the relation which includes score values* $v_1 \leq v_2 \leq \ldots$. *Let* s'_i *be any set of score values satisfying* $v'_1 \leq v'_2 \leq \ldots$*, and define* $\mathcal{D}'$ *to be* $\mathcal{D}$ *with all scores* v_i *replaced with* v'_i*. The* value invariance *property requires that* $R_k(\mathcal{D}) = R_k(\mathcal{D}')$ *for any* k.

4.4.2 Discussion of value invariance

The value-invariance property is defined as it (trivially) holds in the deterministic setting. It is more debatable whether it should always be enforced over uncertain data. The argument against value-invariance notably arises when the score may have an intuitive linear interpretation (e.g., when measuring financial profits, twice the profit is considered twice as good). In such scenarios, value invariance can ignore the "common sense" meaning of the scores and lead to counter-intuitive results. For these cases, it is clearly preferable to choose a method which does *not* obey this property, and instead define an appropriate requirement which captures the given semantics of the score value.

Nevertheless, we argue this property is important to consider for a number of reasons. There are many cases when the score has no such linear interpretation. For example, consider the general case where there is no explicit score value revealed to the algorithm; instead, for any pair of (deterministic) tuples, there is an "oracle" which reports which ranks above the other. This encodes a total order. Then, any method which can operate in this setting must necessarily obey value invariance, whereas methods which rely on being given a score value will be unable to operate. Other examples arise when scores arise from outputs from the sum of classification algorithms, and so have no linear property. Instead, we only have that a larger total score is preferable. Here (as in the deterministic ranking case), the ranking should be invariant under different score values which give the same total ordering. For example, consider the relation with tuple-level uncertainty illustrated in Table 4.4. Here, the scores are $70 \leq 80 \leq 92 \leq 100$. The value invariance property demands that we could replace these scores with, say, $1 \leq 2 \leq 3 \leq 1000$, and the result of the ranking would still be the same.

Whether or not value invariance is considered desirable in a given ranking situation, it is important to know if a proposed ranking method will guarantee the property or not. It is perhaps surprising to note that *all* existing ranking definitions [86, 87, 90, 105] for probabilistic data have this property.

4.4.3 Properties and probabilities

Observe that these conditions make little explicit reference to probability models, and can apply to almost any ranking setting. They trivially hold for the top-k semantics over certain data. It should nevertheless be noted that these properties are not meant to be a complete characterization of ranking queries in probabilistic data. Indeed, in some cases, a specific application may only require a subset of these conditions. However, in order to choose a ranking definition to work with for different domain requirements, it is imperative to examine the semantics of ranking queries for probabilistic data, especially in the context of real-world applications, and to understand which ranking definitions provide which properties. The intricate interplay between the score and the probability attributes indicates that no single definition will be a universal best choice for all applications. Nevertheless, we believe that many natural situations will require all these five simple properties to hold.

4.4.4 Top-k queries on probabilistic data

We now consider how to extend ranking queries to uncertain data. The two uncertainty models require different approaches: In the attribute-level model, a tuple has a random score but it always exists in any random possible world, i.e., every tuple participates in the ranking process in all possible worlds, and we rank these N tuples based on their score distribution. In contrast, in the tuple-level model, a tuple has a fixed score but it may not always appear, i.e., it may not participate in the ranking process in some possible worlds. We still aim to produce a ranking on all N tuples, taking this into account.

Considering the tuple-level model, the difficulty of extending ranking queries to probabilistic data is that there are now two distinct orderings present in the data: that given by the score, and that given by the probabilities. These two types of information need to be combined in some meaningful way to produce the top-k (this can be orthogonal to the model used to describe the uncertainty in the data). We now detail a variety of approaches that have been taken, and discuss their shortcomings with respect to the conditions we have defined. The key properties are summarized in Table 4.5.

Table 4.5. Summary of ranking methods for uncertain data. ©2011 IEEE

Ranking method	Exact-k	Containment	Unique-Rank	Value-Invariant	Stability
U-top$k$ [86]	×	×	✓	✓	✓
U-kRanks [86, 105]	✓	✓	×	✓	×
PT-k [90]	×	weak	✓	✓	✓
Global-top$k$ [87]	✓	×	✓	✓	✓
Expected score	✓	✓	✓	×	✓
Expected rank	✓	✓	✓	✓	✓

4.4.4.1 Ignore one dimension

A baseline approach is to ignore one dimension (score or likelihood). If we ignore likelihood, it becomes an instance of ranking certain data. The work of Ré *et al.* [85] studies the case where there is no score, and instead ranks the results of a query solely by their probability (across all possible worlds). However, when there is both score and probability information available, ignoring one dimension is insufficient for most purposes. Such simple methods may trivially satisfy the above five basic properties, but they fail to meaningfully combine information in the input. They are easily shown to lead to undesirable features, such as ranking very low probability tuples above much more probable ones.

4.4.4.2 Combine two rankings

There has been much work on taking multiple rankings and combining them (e.g., taking the top 50 query web search results from multiple search engines, and combining them to get an overall ranking) based on minimizing disagreements [111, 112]. Likewise, skyline-based approaches extract points which do not dominate each other, and are not themselves dominated, under multiple ordered dimensions [113]. However, such approaches fail to account for the inherent semantics of the probability distribution: it is insufficient to treat it simply as an ordinal attribute, as this loses the meaning of the relative likelihoods, and does not guarantee our required properties.

4.4.4.3 Most likely top-k

Since a probabilistic relation can define exponentially many possible worlds, one approach to the top-k problem finds the top-k set that has the highest support over all possible worlds. In other words, (conceptually) extract the top-k from each possible world, and compute the support (probability) of each distinct top-k set found. The *U-Topk* approach [86] reports the most likely top-k as the answer to the ranking query (that is, the top-k set with the highest total probability across all worlds). This method has the advantage that it more directly incorporates the likelihood information, and satisfies unique ranking, value invariance, and stability. However, it may not always return k tuples when $\mathcal{D}$ is small, as also pointed out in [87]. More importantly, it violates the containment property. In fact, there are simple examples where the top-k can be completely disjoint from the top-$(k + 1)$. Consider the attribute-level model example in Table 4.2. The top-1 result under the U-Top$k$ definition is t_1, since its probability of having the highest score in a random possible world is $0.24 + 0.16 = 0.4$, larger than that of t_2 or t_3. However, the top-2 result is (t_2, t_3), whose probability of being the top-2 is 0.36, larger than that of (t_1, t_2) or (t_1, t_3). Thus, the top-2 list is completely disjoint from the top-1. Similarly, one can verify that for the tuple-level model example in Table 4.4, the top-1 result is t_1 but the top-2 is (t_2, t_3) or (t_3, t_4). No matter what tie-breaking rule is used, the top-2 is completely disjoint from the top-1.

4.4.4.4 Most likely tuple at each rank

The previous approach fails because it deals with top-k sets as immutable objects. Instead, we could consider the property of a certain tuple being ranked kth in a possible world. In particular, let $X_{i,j}$ be the event that tuple j is ranked i within a possible world. Computing $\Pr[X_{i,j}]$ for all i, j pairs, this approach reports the ith result as $\arg\max_j \Pr[X_{i,j}]$, i.e., the tuple that is most likely to be ranked ith over all possible worlds. This is the *U-kRanks* approach [86]; essentially the same definition is proposed as *PRank* in [105] and analyzed in the context of distributions over spatial data. This definition overcomes the shortcomings of U-Top$k$ and satisfies exact-k and containment. However, it fails on unique ranking, as one tuple may dominate multiple ranks at the same time. A related issue is that some tuples may be quite likely, but never get reported. So in Table 4.2, the top-3 under this definition is t_1, t_3, t_1: t_1 appears twice and t_2 never; for Table 4.4, there is a tie for the third position, and there is no fourth placed tuple, even though $N = 4$. These issues have also been pointed out in [87, 90]. In addition, it fails on stability, as shown in [87], since when the score of a tuple becomes larger, it may leave its original rank but cannot take over any higher ranks as the dominating winner.

4.4.4.5 Rank by top-k probability

Attempting to patch the previous definition, we can replace the event "tuple i is at rank k" with the event "tuple i is at rank k or better," and reason about the probability of this event. That is, define the top-k probability of a tuple as the probability that it is in the top-k over all possible worlds. The *probabilistic threshold top-k* query (*PT-k* for short) returns the set of all tuples whose top-k probability exceeds a user-specified probability p [90]. However, for a user-specified p, the "top-k" list may not contain k tuples, violating exact-k. If we fix p and increase k, the top-k lists do expand, but they only satisfy the weak containment property. For instance, consider the tuple-level example in Table 4.2. If we set $p = 0.4$, then the top-1 list is (t_1). However, both the top-2 and top-3 lists contain the same set of tuples: t_1, t_2, t_3. A further drawback of using PT-k for ranking is that the user has to specify the threshold p, which greatly affects the result.

Similarly, the *Global-Topk* method ranks the tuples by their top-k probability, and then takes the top-k of these [87] based on this probability. This makes sure that exactly k tuples are returned, but it again fails on containment. In Table 4.2, under the Global-Top$k$ definition, the top-1 is t_1, but the top-2 is (t_2, t_3). In Table 4.4, the top-1 is t_1, but the top-2 is (t_3, t_2).

Further, note that as k increases towards N, then the importance of the score diminishes, so these two methods reduce to simply ranking the reported top-k items by probability alone.

4.4.4.6 Expected score

The above approaches all differ from traditional ranking queries, in that they do not define a single ordering of the tuples from which the top-k is taken—in other words, they do not resemble "top-k" in the literal interpretation of the term. A simple approach in this direction is to just compute the expected score of each tuple, and rank by this score, then take the top-k. This method may be desirable when the score has a strong linear interpretation (e.g., it represents a financial profit), but it does not apply in the "oracle model" where only the relative ordering of each pair of tuples is given. It is easy to check that the expected score approach directly implies exact-k, containment, unique ranking, and stability. However, this is very dependent on the values of the scores: consider a tuple which has very low probability but a score that is orders of magnitude higher than others—then it gets propelled to the top of the ranking, since it has the highest expected score, even though it is unlikely. However, if we reduce this score to being just greater than the next highest score, the tuple will drop down the ranking. It therefore violates value invariance. Furthermore, in the tuple-level model, simply using the expected score ignores all the correlation rules completely.

4.4.5 The rank distribution and expected ranks

Motivated by deficiencies of existing definitions, we propose a new ranking framework that depends on the ranks of a tuple across all possible worlds and we refer to these ranks (for a given tuple t), together with the corresponding possible worlds' probabilities, as t's rank distribution. Our intuition is that top-k over certain data is defined by first providing a total ordering of the tuples, and then selecting the k "best" tuples under the ordering. Any such definition immediately provides the containment and unique-ranking properties. After rejecting expected score due to its sensitivity to the score values (i.e., it does not provide value invariance), a natural candidate is to consider the orderings based on the *ranks* of the tuple over the possible worlds. More formally,

Definition 4.6 (Ranks of a tuple in all possible worlds)**.** *The rank of tuple t_i in a possible world W is defined to be the number of tuples whose score is higher than t_i (the top tuple has rank 0), i.e.,* $\mathrm{rank}_W(t_i) = |\{t_j \in W | v_j > v_i\}|$*. In the tuple-level model, for a world W where t_i does not appear, we define* $\mathrm{rank}_W(t_i) = |W|$*, i.e., it follows after all appearing tuples.*

The ranks of a tuple t_i in all possible worlds and the probabilities of all worlds constitute a proper probability distribution function (pdf): $\mathrm{rank}(t_i)$, i.e., $\mathrm{rank}(t_i) = \{(\mathrm{rank}_W(t_i), \Pr[W])\}$ for $\forall W \in \mathcal{W}$, since $\sum_{W \in \mathcal{W}} \Pr[W] = 1$. Note to form a well-defined pdf, we need to combine (sum up the corresponding probabilities) the ranks from different possible worlds that have the same value (i.e., $\mathrm{rank}_W(t_i)$) from the above set. Formally, let $R(t_i)$ be a random variable for the rank of tuple t_i in a random selected possible world,

Definition 4.7 (Rank Distribution)**.** *The rank distribution of a tuple t_i,* $\mathrm{rank}(t_i)$*, is a proper probability distribution function (pdf) for the random variable $R(t_i)$ defined as:*

$$\mathrm{rank}(t_i) : \Pr[R(t_i) = V] = \sum_{W \in \mathcal{W} | \mathrm{rank}_W(t_i) = V} \Pr[W], \quad \forall V \in [0, N]$$

The rank distribution for a tuple captures important information on how a tuple behaves in terms of ranking across all possible worlds. Applying statistical operators to each distribution to generate a single statistic is a natural way to summarize the rank distribution, and can be used as the basis for ranking. We first study the *expectation*, which leads to a new ranking method, which we call the *expected rank.*

Definition 4.8 (Expected Rank)**.** *The expected rank of tuple t_i is the expectation of* $\mathrm{rank}(t_i)$*. The smaller* $\mathrm{rank}(t_i)$*'s expectation, the smaller t_is final rank, denoted as $r(t_i)$.*

In the attribute-level model, the expected rank $r(t_i)$ can be computed as the expectation on $\text{rank}_W(t_i)$*'s, then the top-k tuples with the lowest $r(t_i)$ can be returned. More precisely,*

$$r(t_i) = \mathbf{E}[R(t_i)] = \sum_{W \in \mathcal{W}, t_i \in W} \Pr[W] \cdot \text{rank}_W(t_i) \tag{4.1}$$

In the tuple-level model, in a world W where t_i does not appear, $\text{rank}_W(t_i) = |W|$*, i.e., we imagine it follows after all the tuples which do appear (as per Definition 4.6 above). So,*

$$\begin{aligned} r(t_i) &= \sum_{t_i \in W} \Pr[W] \,\text{rank}_W(t_i) + \sum_{t_i \notin W} \Pr[W] \cdot |W| \\ &= \sum_{W \in \mathcal{W}} \Pr[W] \,\text{rank}_W(t_i), \end{aligned} \tag{4.2}$$

where the definition of $\text{rank}_W(t_i)$ *is extended so that* $\text{rank}_W(t_i) = |W|$ *if $t_i \notin W$.*

For the example in Table 4.2, the expected rank for t_2 is $r(t_2) = 0.24 \times 1 + 0.16 \times 2 + 0.36 \times 0 + 0.24 \times 1 = 0.8$. Similarly $r(t_1) = 1.2$, $r(t_3) = 1$, and so the final ranking is (t_2, t_3, t_1). For the example in Table 4.4, $r(t_2) = 0.2 \times 1 + 0.2 \times 3 + 0.3 \times 0 + 0.3 \times 2 = 1.4$. Note t_2 does not appear in the second and the fourth worlds, so its ranks are taken to be 3 and 2, respectively. Similarly $r(t_1) = 1.2$, $r(t_3) = 0.9$, $r(t_4) = 1.9$. So the final ranking is (t_3, t_1, t_2, t_4).

We prove expected ranks satisfies all five fundamental properties. For simplicity, we assume the expected ranks are unique, and so the ranking forms a total ordering. In practice, ties can be broken arbitrarily, e.g., based on having the lexicographically smaller identifier. The same tie-breaking issues affect the ranking of certain data as well.

Theorem 4.1. *Expected rank satisfies exact-k, containment, unique ranking, value invariance, and stability.*

Proof of Theorem 4.1: The first three properties follow immediately from the fact that the expected rank is used to give an ordering. Value invariance follows by observing that changing the score values will not change the rankings in possible worlds, and therefore does not change the expected ranks.

For stability, we show that when we change a tuple t_i to $t_i^{\uparrow}$ (as in Definition 4.4), its expected rank will not increase, while the expected rank of any other tuple will not decrease. Let r' be the expected rank in the uncertain relation $\mathcal{D}'$ after changing t_i to $t_i^{\uparrow}$. We need to show that $r(t_i) \geq r'(t_i^{\uparrow})$ and $r(t_{i'}) \leq r'(t_{i'})$ for any $i' \neq i$.

Consider the attribute-level model first. By Definition 4.8 and linearity of expectation, we have

$$\begin{aligned} r(t_i) &= \sum_{j\neq i} \Pr[X_i < X_j] = \sum_{j\neq i}\sum_{\ell} p_{j,\ell} \Pr[X_i < v_{j,\ell}] \\ &\geq \sum_{j\neq i}\sum_{\ell} p_{j,\ell} \Pr[X_i^{\uparrow} < v_{j,\ell}] \quad (\text{because } X_i \preceq X_i^{\uparrow}) \\ &= \sum_{j\neq i} \Pr[X_i^{\uparrow} < X_j] = r'(t_i^{\uparrow}). \end{aligned}$$

For any $i' \neq i$,

$$\begin{aligned} r(t_{i'}) &= \Pr[X_{i'} < X_i] + \sum_{j\neq i', j\neq i} \Pr[X_{i'} < X_j] \\ &= \sum_{\ell} p_{i',\ell} \Pr[v_{i',\ell} < X_i] + \sum_{j\neq i', j\neq i} \Pr[X_{i'} < X_j] \\ &\leq \sum_{\ell} p_{i',\ell} \Pr[v_{i',\ell} < X_i^{\uparrow}] + \sum_{j\neq i', j\neq i} \Pr[X_{i'} < X_j] \\ &= \Pr[X_{i'} < X_i^{\uparrow}] + \sum_{j\neq i', j\neq i} \Pr[X_{i'} < X_j] = r'(t_{i'}) \end{aligned}$$

Next, consider the tuple-level model. If $t_i^{\uparrow}$ has a larger score than t_i but the same probability, then $r(t_i) \geq r'(t_i^{\uparrow})$ follows easily from (4.2) since $\text{rank}_W(t_i)$ can only get smaller while the second term of (4.2) remains unchanged. For similar reasons, $r(t_{i'}) \leq r'(t_{i'})$ for any $i' \neq i$.

If $t_i^{\uparrow}$ has the same score as t_i but a larger probability, $\text{rank}_W(t_i)$ stays the same for any possible world W, but $\Pr[W]$ may change. We divide all the possible worlds into three categories: (a) those containing t_i, (b) those containing one of the tuples in the exclusion rule of t_i (other than t_i), and (c) all other possible worlds. Note that $\Pr[W]$ does not change for any W in category (b), so we only focus on categories (a) and (c). Since $r(t_i)$ is nothing but a weighted average of the ranks in all the possible worlds, where the weight of W is $\Pr[W]$, it is sufficient to consider the changes in the contribution of the possible worlds in categories (a) and (c). Observe that there is a one-to-one mapping between the possible worlds in category (c) and (a): $W \leftrightarrow W \cup \{t_i\}$. For each such pair, its contribution to $r(t_i)$ is

$$\Pr[W]\cdot |W| + \Pr[W\cup\{t_i\}]\cdot \text{rank}_{W\cup\{t_i\}}(t_i). \tag{4.3}$$

Suppose the tuples in the exclusion rule of t_i are $t_{i,1},\ldots,t_{i,s}$. Note that W and $W\cup\{t_i\}$ differ only in the inclusion of t_i, so we can write $\Pr[W] = \pi\left(1 - \sum_{\ell} p(t_{i,\ell}) - p(t_i)\right)$ and $\Pr[W\cup\{t_i\}] = \pi p(t_i)$ for some π. When $p(t_i)$ increases to $p(t_i^{\uparrow})$, the increase in (4.3) is

$$\begin{aligned} &\pi(p(t_i) - p(t_i^{\uparrow}))|W| + \pi(p(t_i^{\uparrow}) - p(t_i))\,\text{rank}_{W\cup\{t_i\}}(t_i) \\ = \quad &\pi(p(t_i) - p(t_i^{\uparrow}))(|W| - \text{rank}_{W\cup\{t_i\}}(t_i)) \leq 0. \end{aligned}$$

The same holds for each pair of possible worlds in categories (a) and (c). Therefore, we have $r(t_i) \geq r'(t_i^{\uparrow})$.

For any $i' \neq i$, the contribution of each pair is

$$\Pr[W] \cdot \mathrm{rank}_W(t_{i'}) + \Pr[W \cup \{t_i\}] \cdot \mathrm{rank}_{W \cup \{t_i\}}(t_{i'}). \tag{4.4}$$

When $p(t_i)$ increases to $p(t_i^{\uparrow})$, the increase in (4.4) is

$$\pi(p(t_i) - p(t_i^{\uparrow}))(\mathrm{rank}_W(t_{i'}) - \mathrm{rank}_{W \cup \{t_i\}}(t_{i'})) \geq 0.$$

The same holds for each pair of possible worlds in categories (a) and (c). Therefore, we have $r'(t_{i'}) \geq r(t_{i'})$. ■

4.5 Attribute-Level Uncertainty Model

This section presents efficient algorithms for calculating the expected rank of an uncertain relation $\mathcal{D}$ with N tuples in the attribute-level uncertainty model. We first show an exact algorithm that can calculate the expected ranks of all tuples in $\mathcal{D}$ with $O(N \log N)$ processing cost. We then propose an algorithm that can terminate the search as soon as the top-k tuples with the k smallest expected ranks are guaranteed to be found without accessing all tuples.

4.5.1 Exact computation

By Definition 4.8 and the linearity of expectation, we have

$$r(t_i) = \sum_{j \neq i} \Pr[X_j > X_i]. \tag{4.5}$$

The brute-force search (BFS) approach requires $O(N)$ time to compute $r(t_i)$ for one tuple and $O(N^2)$ time to compute ranks of all tuples. The quadratic dependence on N is prohibitive for large N. Below we present an improved algorithm requiring $O(N \log N)$ time. We observe that (4.5) can be written as:

$$\begin{aligned} r(t_i) &= \sum_{j \neq i} \sum_{\ell=1}^{s_i} p_{i,\ell} \Pr[X_j > v_{i,\ell}] = \sum_{\ell=1}^{s_i} p_{i,\ell} \sum_{j \neq i} \Pr[X_j > v_{i,\ell}] \\ &= \sum_{\ell=1}^{s_i} p_{i,\ell} \big(\sum_j \Pr[X_j > v_{i,\ell}] - \Pr[X_i > v_{i,\ell}] \big) \\ &= \sum_{\ell=1}^{s_i} p_{i,\ell} \big(q(v_{i,\ell}) - \Pr[X_i > v_{i,\ell}] \big), \end{aligned} \tag{4.6}$$

where we define $q(v) = \sum_j \Pr[X_j > v]$. Let U be the universe of all possible values of all X_i, $i = 1, \ldots, N$. Because we assume each pdf has size bounded by s, we have $|U| \leq |sN|$. When s is a constant, we have $|U| = O(N)$.

Now observe that we can precompute $q(v)$ for all $v \in U$ with a linear pass over the input after sorting U which has a cost of $O(N \log N)$. Following (4.6), exact computation of the expected rank for a single tuple can now be done in constant time given $q(v)$ for all $v \in U$. While computing these expected ranks, we maintain a priority queue of size k that stores the k tuples with smallest expected ranks dynamically. When all tuples have been processed, the contents of the priority queue are returned as the final answer. Computing $q(v)$ takes time $O(N \log N)$; getting expected ranks of all tuples while maintaining the priority queue takes $O(N \log k)$ time. Hence, the overall cost of this approach is $O(N \log N)$. We denote this algorithm as *A-ERrank*.

4.5.2 Pruning by expected scores

A-ERank is very efficient even for large N values. However, in certain scenarios, accessing a tuple is considerably expensive (if it requires significant IO access). It then becomes desirable to reduce the number of tuples accessed in order to find the answer. It is possible to find a set of (possibly more than k tuples) which is guaranteed to include the true top-k expected ranks, by pruning based on tail bounds of the score distribution. If tuples are sorted in decreasing order of their expected scores, i.e., $\mathbf{E}[X_i]$'s, we can terminate the search early. In the following discussion, we assume that if $i < j$, then $\mathbf{E}[X_i] \geq \mathbf{E}[X_j]$ for all $1 \leq i, j \leq N$. Equivalently, we can think of this as an interface which generates each tuple in turn, in decreasing order of $\mathbf{E}[X_i]$.

The pruning algorithm scans these tuples, and maintains an upper bound on $r(t_i)$, denoted $r^+(t_i)$, for each t_i seen so far, and a lower bound on $r(t_u)$ for any unseen tuple t_u, denoted r^-. The algorithm halts when there are at least k $r^+(t_i)$s that are smaller than r^-. Suppose n tuples $t_1, \ldots, t_n$ have been scanned. For $\forall i \in [1, n]$, we have:

$$\begin{aligned}
r(t_i) &= \sum_{j \leq n, j \neq i} \Pr[X_j > X_i] + \sum_{n < j \leq N} \Pr[X_j > X_i] \\
&= \sum_{j \leq n, j \neq i} \Pr[X_j > X_i] + \sum_{n < j \leq N} \sum_{\ell=1}^{s_i} p_{i,\ell} \Pr[X_j > v_{i,\ell}] \\
&\leq \sum_{j \leq n, j \neq i} \Pr[X_j > X_i] + \sum_{n < j \leq N} \sum_{\ell=1}^{s_i} p_{i,\ell} \frac{\mathbf{E}[X_j]}{v_{i,\ell}} \\
&\qquad\qquad\qquad\qquad\qquad \text{(Markov Ineq.)} \\
&\leq \sum_{j \leq n, j \neq i} \Pr[X_j > X_i] + (N - n) \sum_{\ell=1}^{s_i} p_{i,\ell} \frac{\mathbf{E}[X_n]}{v_{i,\ell}}. \qquad (4.7)
\end{aligned}$$

The first term in (4.7) can be computed using only seen tuples $t_1, \ldots, t_n$. The second term could be computed using X_i and X_n. Hence, from scanned tuples, we can maintain an upper bound on $r(t_i)$ for each tuple in $\{t_1, \ldots, t_n\}$, i.e., we can set $r^+(t_i)$ to be (4.7) for $i = 1, \ldots, n$. $r^+(t_i)$'s second term is updated for every new t_n (as well as the first term for t_n).

Now we provide the lower bound r^-. Consider any unseen tuple $t_u, u > n$, we have:

$$\begin{aligned} r(t_u) &\geq \sum_{j \leq n} \Pr[X_j > X_u] = n - \sum_{j \leq n} \Pr[X_u \geq X_j] \\ &= n - \sum_{j \leq n} \sum_{\ell=1}^{s_j} p_{j,\ell} \Pr[X_u > v_{j,\ell}] \\ &\geq n - \sum_{j \leq n} \sum_{\ell=1}^{s_j} p_{j,\ell} \frac{\mathbf{E}[X_n]}{v_{j,\ell}}. \qquad \text{(Markov Ineq.)} \end{aligned} \tag{4.8}$$

This holds for any unseen tuple. Hence, we set r^- to be (4.8). Note that (4.8) only depends on the seen tuples. It is updated with every new tuple t_n.

These bounds lead immediately to an algorithm that maintains $r^+(t_i)$s for all tuples $t_1, \ldots, t_n$ and r^-. For each new tuple t_n, the $r^+(t_i)$s and r^- are updated. From these, we find the kth largest $r^+(t_i)$ value, and compare this to r^-. If it is less, then we know for sure that the k tuples with the smallest expected ranks *globally* are among the first n tuples, and can stop retrieving tuples. Otherwise, we move on to the next tuple. We refer to this algorithm as *A-ERank-Prune*.

A remaining challenge is how to find the k tuples with the smallest expected ranks using the first n tuples alone. This is difficult as it is not possible to obtain a precise order on their final ranks without inspecting all N tuples in $\mathcal{D}$. Instead, we use the curtailed database $\mathcal{D}' = \{t_1, \ldots, t_n\}$, and compute exact expected rank $r'(t_i)$ for every tuple (for $i \in [1, n]$) t_i in $\mathcal{D}'$. The rank $r'(t_i)$ turns out to be an excellent surrogate for $r(t_i)$ for $i \in [1, n]$ in $\mathcal{D}$ (when the pruning algorithm terminates after processing n tuples). Hence, we return the top-k of these as the result of the query. We show an evaluation of the quality of this approach in our experimental study.

A straightforward implementation of *A-ERrank-Prune* requires $O(n^2)$ time. After seeing t_n, the bounds in both (4.7) and (4.8) can be updated in constant time, by retaining $\sum_{\ell=1}^{s_j} \frac{p_{i,\ell}}{v_{i,\ell}}$ for each seen tuple. The challenge is updating the first term in (4.7) for all $i \leq n$. A basic approach requires linear time, for adding $\Pr[X_n > X_i]$ to the already computed $\sum_{j \leq n-1, j \neq i} \Pr[X_j > X_i]$ for all i's as well as computing $\sum_{i \leq n-1} \Pr[X_i > X_n]$). This leads to a complexity of $O(n^2)$ for algorithm A-ERrank-Prune. Using a similar idea in designing algorithm A-ERank, it is possible to utilize value universe U' of all seen tuples and maintain prefix sums of the $q(v)$ values, which would drive down the cost of this step to $O(n \log n)$.

4.6 Tuple-Level Uncertainty Model

We now consider ranking an uncertain database $\mathcal{D}$ in the *tuple-level uncertainty model.* For $\mathcal{D}$ with N tuples and M rules, the aim is to retrieve the k tuples with the smallest expected ranks. Recall each rule τ_j is a set of tuples, where $\sum_{t_i \in \tau_j} p(t_i) \leq 1$. Without loss of generality, we assume the tuples $t_1, \ldots, t_n$ are already sorted by the ranking attribute and t_1 is the tuple with the highest score. We use $t_i \diamond t_j$ to denote t_i and t_j are in the same exclusion rule and $t_i \neq t_j$; we use $t_i \bar{\diamond} t_j$ to denote t_i and t_j are not in the same exclusion rule. We first give an exact $O(N \log N)$ algorithm which accesses every tuple. Secondly, we show an $O(n \log n)$ pruning algorithm, which only reads the first n tuples, assuming the *expected* number of tuples in $\mathcal{D}$ is known to the algorithm.

4.6.1 Exact computation

From Definition 4.8, in particular (4.2), given tuples that are sorted by their score attribute, we have:

$$r(t_i) = p(t_i) \cdot \sum_{t_j \bar{\diamond} t_i, j<i} p(t_j) + (1 - p(t_i)) \cdot \left(\frac{\sum_{t_j \diamond t_i} p(t_j)}{1 - p(t_i)} + \sum_{t_j \bar{\diamond} t_i} p(t_j) \right)$$

The first term computes t_i's expected rank for random worlds when it appears, and the second term computes the expected size of a random world W when t_i does not appear in W. The term $\frac{\sum_{t_j \diamond t_i} p(t_j)}{1-p(t_i)}$ is the expected number of appearing tuples in the same rule as t_i, conditioned on t_i not appearing, while $\sum_{t_j \bar{\diamond} t_i} p(t_j)$ accounts for the rest of the tuples. Rewriting,

$$r(t_i) = p(t_i) \cdot \sum_{t_j \bar{\diamond} t_i, j<i} p(t_j) + \sum_{t_j \diamond t_i} p(t_j) + (1 - p(t_i)) \cdot \sum_{t_j \bar{\diamond} t_i} p(t_j). \tag{4.9}$$

Let $q_i = \sum_{j<i} p(t_j)$. We first compute q_i in $O(N)$ time. At the same time, we find the expected number of tuples, $\mathbf{E}[|W|] = \sum_{j=1}^{N} p(t_j)$. Now (4.9) can be rewritten as:

$$r(t_i) = p(t_i) \cdot (q_i - \sum_{t_j \diamond t_i, j<i} p(t_j)) + \sum_{t_j \diamond t_i} p(t_j) + (1 - p(t_i))(\mathbf{E}[|W|] - p(t_i) - \sum_{t_j \diamond t_i} p(t_j)). \tag{4.10}$$

By keeping the auxiliary information $\sum_{t_j \diamond t_i, j<i} p(t_j)$ (i.e., the sum of probabilities of tuples that have score values higher than t_i in the same rule as t_i) and $\sum_{t_j \diamond t_i} p(t_j)$ (i.e., the sum of probabilities of tuples that are in the same rule as t_i) for each tuple t_i in $\mathcal{D}$, $r(t_i)$ can be computed in $O(1)$ time. By maintaining a priority queue of size k that keeps the k tuples with the smallest $r(t_i)$'s, we can select the top-k tuples in $O(N \log k)$ time. Note that both

$\sum_{t_j \diamond t_i, j<i} p(t_j)$ and $\sum_{t_j \diamond t_i} p(t_j)$ are cheap to calculate initially given all the rules in a single scan of the relation (taking time $O(N)$, since each tuple appears in exactly one rule). When $\mathcal{D}$ is not presorted by t_i's score attribute, the running time of this algorithm is dominated by the sorting step, $O(N \log N)$.

4.6.2 Pruning

Provided that the expected number of tuples $\mathbf{E}[|W|]$ is known, we can answer top-k queries more efficiently using pruning techniques without accessing all tuples. Note that $\mathbf{E}[|W|]$ can be efficiently maintained in $O(1)$ time when $\mathcal{D}$ is updated with deletion or insertion of tuples. As $\mathbf{E}[|W|]$ is simply the sum of all the probabilities (note that it does not depend on the rules), it is reasonable to assume that it is always available. Similar to the attribute-level uncertainty case, we assume that $\mathcal{D}$ provides an interface to retrieve tuples in order of their score attribute from the highest to the lowest.

The pruning algorithm scans the tuples in order. After seeing t_n, it can compute $r(t_n)$ exactly using $\mathbf{E}[|W|]$ and q_n in $O(1)$ time based on (4.10). It also maintains $r^{(k)}$, the kth smallest $r(t_i)$ among all the tuples that have been retrieved. This can be done with a priority queue in $O(\log k)$ time per tuple. A lower bound on $r(t_\ell)$ for any $\ell > n$ is computed as follows:

$$\begin{aligned}
&r(t_\ell) \\
&\quad + \sum_{t_j \diamond t_\ell} p(t_j) + (1 - p(t_\ell)) \cdot \sum_{t_j \bar{\diamond} t_\ell} p(t_j) \quad \text{(from (4.9))} \\
&= p(t_\ell) \cdot \sum_{t_j \bar{\diamond} t_\ell, j<\ell} p(t_j) + \mathbf{E}[|W|] - p(t_\ell) - p(t_\ell) \cdot \sum_{t_j \bar{\diamond} t_\ell} p(t_j) \\
&= \mathbf{E}[|W|] - p(t_\ell) - p(t_\ell) \cdot \left(\sum_{t_j \bar{\diamond} t_\ell} p(t_j) - \sum_{t_j \bar{\diamond} t_\ell, j<\ell} p(t_j) \right) \\
&= \mathbf{E}[|W|] - p(t_\ell) - p(t_\ell) \cdot \sum_{t_j \bar{\diamond} t_\ell, j>\ell} p(t_j). \qquad (4.11)
\end{aligned}$$

In the second step, we used the fact that

$$\sum_{t_j \diamond t_\ell} p(t_j) + \sum_{t_j \bar{\diamond} t_\ell} p(t_j) = \mathbf{E}[|W|] - p(t_\ell).$$

Now, since $q_\ell = \sum_{j<\ell} p(t_j)$, we observe that

$$\mathbf{E}[|W|] - q_\ell = \sum_{j>\ell} p(t_j) + p(t_\ell) \geq \sum_{t_j \bar{\diamond} t_\ell, j>\ell} p(t_j).$$

Continuing with (4.11), we have:

$$\begin{aligned}
r(t_\ell) &\geq \mathbf{E}[|W|] - p(t_\ell) - p(t_\ell) \cdot (\mathbf{E}[|W|] - q_\ell) \\
&\geq q_\ell - 1 \geq q_n - 1. \qquad (4.12)
\end{aligned}$$

The last step uses the monotonicity of q_i—by definition, $q_n \le q_\ell$ if $n \le \ell$. Since tuples are scanned in order, obviously $\ell > n$.

Thus, when $r^{(k)} \le q_n - 1$, we know for sure there are at least k tuples amongst the first n with expected ranks smaller than all unseen tuples. At this point, we can safely terminate the search. In addition, recall that for all the scanned tuples, their expected ranks are calculated *exactly* by (4.10). Hence, this algorithm—which we dub *T-ERank-Prune*—can simply return the current top-k tuples. From the above analysis, its time cost is $O(n \log k)$ where n is potentially much smaller than N.

4.7 Median and Quantile Ranks

Our expected rank definition uses the expectation as the basis of ranking, i.e., the absolute ranks of each tuple from all possible worlds are represented by their mean. It is well known that the mean (or equivalently, the expectation) is statistically sensitive to the distribution of the underlying values (in our case, the absolute ranks of the tuple from all possible worlds), especially when there are outliers in the distribution. It is natural to consider alternate statistical operators as the basis of the ranking, such as the median of the rank distribution instead of the mean. This can be further generalized to any quantile for the distribution of absolute ranks for a tuple across all possible worlds. We can then derive the final ranking based on such quantiles. Furthermore, the rank distribution $\text{rank}(t_i)$ for a tuple t_i reflects important characteristics of t_i's rank in any random possible world. Studying these critical statistics (median and general quantiles) for this rank distribution is of independent interest. This section formalizes these ideas and presents efficient algorithms to compute the median and quantile ranks for uncertain databases in both the attribute-level and tuple-level uncertainty models. Similarly to the approach taken for the expected rank, we first present the definitions for these ranks and discuss the efficient, polynomial-time algorithms that compute such ranks; then, we improve the efficiency of our algorithms by designing necessary pruning techniques.

4.7.1 Definitions and properties

We formally define the median and quantile rank as,

Definition 4.9. *For tuple $t_i \in \mathcal{D}$, its median rank $r_m(t_i)$ is the median value from t_i's rank distribution* $\text{rank}(t_i)$*, i.e., it is the value in the cumulative distributive function (cdf) of* $\text{rank}(t_i)$*, denoted as* $\text{cdf}(\text{rank}(t_i))$*, that has a cumulative probability of* 0.5*; for any user-defined ϕ-quantile where $\phi \in (0,1)$, the ϕ-quantile rank of t_i is the value in the* $\text{cdf}(\text{rank}(t_i))$

that has a cumulative probability of ϕ, denoted as $r_\phi(t_i)$.

For the attribute-level uncertainty model example in Table 4.2, t_1's rank distribution rank(t_1) is $\{(0, 0.4), (1, 0), (2, 0.6)\}$. Therefore, t_1's median rank $r_m(t_1)$ is 2. Similarly, $r_m(t_2) = 1$ and $r_m(t_3) = 1$. Hence, the final ranking is (t_2, t_3, t_1), which is identical to the final ranking obtained from the expected ranks. For the tuple-level uncertainty model example in Table 4.4, t_4's rank distribution rank(t_4) is $\{(0, 0), (1, 0.3), (2, 0.5), (3, 0.2)\}$. t_4's median rank $r_m(t_4)$ is 2. Similarly, $r_m(t_1) = 2$, $r_m(t_2) = 1$, and $r_m(t_3) = 1$. The final ranking is (t_2, t_3, t_1, t_4) whereas the final ranking from the expected ranks was (t_3, t_1, t_2, t_4).

Clearly, the median rank is a special case of ϕ-quantile rank where $\phi = 0.5$. Ranking by the median or the ϕ-quantile ranks of all tuples is straightforward. We first derive a total ordering of all tuples in the *ascending* order of their $r_m(t_i)$s (or $r_\phi(t_i)$s), and the k tuples with the smallest values of $r_m(t_i)$s (or $r_\phi(t_i)$s) are returned as the top-k. We can show that ranking by the median rank (or the general quantile rank) offers all properties satisfied by the expected rank. Formally,

Theorem 4.2. *Ranking by median and quantile ranks satisfies all properties in Table 4.5. The proof of is quite similar to the proof of Theorem 4.1, so we omit it.*

The next important problem is whether we can calculate $r_m(t_i)$ or $r_\phi(t_i)$ efficiently for a tuple t_i. Note, besides for ranking purposes, these values themselves are important statistics to characterize the rank distribution of t_i in all possible worlds, rank(t_i). In the sequel, in a random possible world, if t_i and t_j are tied ranking by their scores, t_i ranks before t_j if $i < j$. In general, other tie-breaking mechanisms could be easily substituted. Also, our algorithms work the same way for median and quantile ranks for any quantile values. Hence, for brevity, we focus on discussing median ranks and extend it to quantiles ranks at the end of each case. Recall that for any tuple t, $R(t)$ is a random variable that denotes t's rank in a random possible world. $R(t)$'s distribution is represented by t's rank distribution rank(t).

4.7.2 Attribute-level uncertainty model

In the attribute-level uncertainty model, given a tuple t_i, its uncertain score attribute is represented by the random variable $X_i = \{(v_{i,1}, p_{i,1}), \ldots, (v_{i,s_i}, p_{i,s_i})\}$ for some constant s_i. When X_i takes the value $v_{i,1}$ (denote this as a special tuple t_i^1), tuple t_i's rank in all possible worlds could be described by a probability distribution rank(t_i^1) = rank($t_i | t_i = v_{i,1}$) where '|' means "given that." We concentrate on calculating rank(t_i^1) first. Note that rank(t_i^1) $\equiv \Pr[R(t_i^1) = \ell]$ for $\ell = 0, \ldots, N-1$ (that is, the distribution gives the probability of the random variable $R(t_i^l)$ taking on each of the N possible rank values). For any other tuple $t_j \in \mathcal{D} \wedge j \neq i$, we calculate $\Pr[t_j > t_i^1] = \sum_{\ell=1}^{s_j} p_{j,\ell}|v_{j,\ell} > v_{i,1}$, and $\Pr[t_j < t_i^1]$, $\Pr[t_j = t_i^1]$ similarly. Then when $j < i$,

$$\Pr[R(t_j) < R(t_i^1)] = \Pr[t_j > t_i^1] + \Pr[t_j = t_i^1] \text{ and}$$
$$\Pr[R(t_j) > R(t_i^1)] = \Pr[t_j < t_i^1]$$

and when $j > i$,

$$\Pr[R(t_j) < R(t_i^1)] = \Pr[t_j > t_i^1] \text{ and}$$
$$\Pr[R(t_j) > R(t_i^1)] = \Pr[t_j < t_i^1] + \Pr[t_j = t_i^1].$$

In short, for any t_j, we can calculate both $\Pr[R(t_j) < R(t_i^1)]$, denoted as $p_j^{\uparrow}$, and $\Pr[R(t_j) > R(t_i^1)]$, denoted as $p_j^{\downarrow}$, efficiently. Now, for each tuple t_j there are two possible outcomes, either it ranks higher than t_i^1 with probability $p_j^{\uparrow}$ or it ranks lower than t_i^1 with probability $p_j^{\downarrow}$. There are $(N-1)$ such independent events (one for each t_j for $j \in \{1, \ldots, N\} - \{i\}$). This could be viewed as a *generalized binomial distribution*. In the jth trial, the head probability is $p_j^{\uparrow}$ and the tail probability is $p_j^{\downarrow}$. The possible ranks of t_i^1 (essentially it is $(t_i|t_i = v_{i,1})$) are simply the possible number of heads from this distribution. Then, $\text{rank}(t_i^1)$ is the probability distribution on the number of heads in this generalized binomial distribution, i.e., $\text{rank}(t_i^1) = \Pr[\text{number of heads} = \ell]$ for $\ell = 1, \ldots, N-1$. For the binomial distribution, there is a compact closed-form formula to calculate $\Pr[\text{number of heads} = \ell]$ for any ℓ. This no longer holds for the generalized binomial distribution. However, one can efficiently compute $\Pr[\text{number of heads} = \ell]$ for all ℓ in this case as follows. Let $E_{\gamma,j}$ be the probability that in the first j trials there are γ number of heads. Then,

$$E_{\gamma,j} = E_{\gamma-1,j-1} \times p_j^{\uparrow} + E_{\gamma,j-1} \times p_j^{\downarrow} \tag{4.13}$$

From equation 4.13, the rank distribution $\text{rank}(t_i^1)$ is defined by $\Pr[\text{number of heads} = \ell]$ for $\ell \in \{0, 1, \ldots, N-1\}$, which is given by $E_{\ell,N-1}$'s for $\ell \in \{0, 1, \ldots, N-1\}$. This observation and Equation 4.13 immediately give us a dynamic programming formulation to calculate the rank distribution $\text{rank}(t_i^1)$.

We can carry out the similar procedure to obtain the distributions $\text{rank}(t_i^1), \ldots, \text{rank}(t_i^{s_i})$, one for each choice of t_i. Finally, $\text{rank}(t_i) \equiv \Pr[R(t_i) = \ell]$ for $\ell \in \{0, 1, \ldots, N-1\}$, where $\Pr[R(t_i) = \ell] = \sum_{\kappa=1}^{s_i} p_{i,\kappa} \times \Pr[R(t_i^{\kappa}) = \ell]$ and $\Pr[R(t_i^{\kappa}) = \ell]$ is given by $\text{rank}(t_i^{\kappa})$, i.e, the rank distribution of t_i, $\text{rank}(t_i)$, is simply the weighted sum of the distributions

$\text{rank}(t_i^1), \ldots, \text{rank}(t_i^{s_i})$. With $\text{rank}(t_i)$, one can easily get t_i's median rank $r_m(t_i)$ or any ϕ-quantile rank $r_\phi(t_i)$. Given a query parameter k, the final step is to simply retrieve the k tuples with the smallest median (or quantile) rank values. We denote this algorithm as *A-MQRank*.

Example 4.1. *For the attribute-level uncertainty model example in Table 4.2, t_2^1's rank distribution* $\text{rank}(t_2^1) = \text{rank}(t_2 | t_2 = 92)$ *is* $\{(0, 0.6), (1, 0.4), (2, 0)\}$ *and t_2^2's rank distribution* $\text{rank}(t_2^2) = \text{rank}(t_2 | t_2 = 80)$ *is* $\{(0, 0), (1, 0.6), (2, 0.4)\}$. *Therefore, t_2's rank distribution* $\text{rank}(t_2)$ *is* $\{(0, 0.36), (1, 0.48), (2, 0.16)\}$.

We observe that the same principle could be applied for the case of continuous distributions. Essentially, for a tuple t_i, we need to compute $p_j^{\uparrow} = \Pr[X_j > X_i]$ and $p_j^{\downarrow} = \Pr[X_j < X_i]$ for any other tuple $t_j \in \mathcal{D}$, where X_j and X_i are two random variables with continuous distributions. Once $p_j^{\uparrow}$ and $p_j^{\downarrow}$ are available, the rest is the same as discussed above.

4.7.3 The complexity of *A-MQRank*

For an uncertain database $\mathcal{D}$ with N tuples and assuming that each tuple takes on at most s possible choices, the cost of one dynamic program for one choice of a tuple (i.e., applying (4.13)) is $O(N^2)$. We have to do this for each choice of every tuple. Hence, the cost of the algorithm *A-MQRank* is $O(sN^3)$. When s is a constant, the complexity of *A-MQRank* is $O(N^3)$.

4.7.4 Pruning techniques

In practice, the number of choices of each tuple could be large. The contribution by the factor of s in the overall computation cost for the algorithm $O(sN^3)$ may be nonnegligible. To remedy this problem, an important observation is that if we rank the score values of all choices for a tuple in decreasing order, then the median (or any quantile) rank value for a tuple after seeing more choices will only increase. This intuition gives us a way to lower-bound the median (or any quantile) rank value in any intermediate steps after seeing any number of choices for a tuple. Specifically, after calculating the rank distributions $\text{rank}(t_i^{\ell})$'s for the first x number of choices for a tuple t_i, we denote the lower-bound on $r_m(t_i)$ as $r_m^x(t_i)$ (or $r_\phi^x(t_i)$ for a quantile rank $r_\phi(t_i)$ with a quantile value ϕ). We would like to maintain $r_m^x(t_i)$ such that (a) for $\forall \ell_1, \ell_2 \in [1, s_i]$, if $\ell_1 < \ell_2$, then $r_m^{\ell_1}(t_i) \le r_m^{\ell_2}(t_i)$; and (b) $r_m^{s_i}(t_i) = r_m(t_i)$. The same holds for the quantile ranks.

Assume we can achieve the above, an immediate pruning technique is to maintain a priority queue of size k for the first ℓ ($\ell \in [1, N)$) tuples whose exact median ranks (or quantile ranks) have already been calculated. The priority queue is ordered by increasing order of the tuple's exact median rank (or quantile rank) and we denote the kth tuple's rank as r_k. When processing the $(\ell+1)$th tuple, after calculating $\text{rank}(t^i_{\ell+1})$'s for $i = 1, \ldots, x$th choices of $t_{\ell+1}$, if $r^x_m(t_i) \geq r_k$ (or $r^x_\phi(t_i) \geq r_k$), we can stop processing the remaining choices of $t_{\ell+1}$ and safely claim $t_{\ell+1}$ has no chance to be in the final top-k answer. If one has to exhaust all choices for $t_{\ell+1}$, then the exact rank $r_m(t_{\ell+1})$ (or $r_\phi(t_{\ell+1})$) is obtained and the priority queue is updated if necessary: if $t_{\ell+1}$'s rank is smaller than r_k, it will be inserted into the queue with its rank and the last (the kth) element of the queue will be deleted.

The remaining challenge is how to calculate the lower-bound $r^x_m(t_i)$ for a tuple t_i after processing its first x choices. A key observation is each tuple's choices are sorted by descending order of their score values, i.e., for $\forall t_i$ and $\forall \ell_1, \ell_2 \in [1, s_i]$, if $\ell_1 < \ell_2$, then $v_{i,\ell_1} > v_{i,\ell_2}$. After processing the first x ($x < s_i$) choices of t_i, we have obtained x rank distributions, $\text{rank}(t^1_i), \ldots, \text{rank}(t^x_i)$. At this point, we can construct a (notional) tuple $t_{i,x}$ with $(x+1)$ choices as follows: $\{(v_{i,1}, p_{i,1}), \ldots, (v_{i,x}, p_{i,x}),\ (v_{i,x}, 1 - \sum_{\ell=1}^{x} p_{i,\ell})\}$. The first x choices of $t_{i,x}$ are identical to the first x choices of t_i; and the last choice of $t_{i,x}$ has the same score value, $v_{i,x}$ with probability as $1 - \sum_{\ell=1}^{x} p_{i,\ell}$ — we aggregate the probability of all remaining choices (of t_i) into one choice and make its score value the same as the last processed choice (the xth one) from t_i. We can write the rank distribution for constructed tuple $t_{i,x}$ as follows, which follows immediately from its construction:

Lemma 4.1. *We have* $\forall \ell \in [1, x]$, $\text{rank}(t^\ell_{i,x}) = \text{rank}(t^\ell_i)$*;* $\text{rank}(t^{x+1}_{i,x}) = \text{rank}(t^x_{i,x})$*; and* $\text{rank}(t_{i,x}) = \sum_{\ell=1}^{x} p_{i,\ell} \times \text{rank}(t^\ell_i) + (1 - \sum_{\ell=1}^{x} p_{i,\ell}) \times \text{rank}(t^x_i)$.

The next result is that the median (quantile) value from distribution $\text{rank}(t_{i,x})$ will not be larger than the median (or corresponding quantile) value in distribution $\text{rank}(t_i)$, i.e.,

Lemma 4.2. $\forall x \in [1, s_i)$ *and* $\forall \phi \in [0, 1]$, $r_\phi(t_{i,x}) \leq r_\phi(t_i)$. *A special case is that* $r_m(t_{i,x}) \leq r_m(t_i)$.

This follows immediately from Lemma 4.1 and the definition of $r_\phi(t)$ and $r_m(t)$ for any tuple t. Lemma 4.2 indicates that by constructing $t_{i,x}$ after processing the first x choices of the tuple t_i, we can efficiently obtain a lower bound on $r_\phi(t_i)$ or $r_m(t_i)$. This can be done after processing each choice of t_i and it perfectly satisfies our pruning framework. Note Lemma 4.1 shows that after processing the first x choices of the tuple t_i, $\text{rank}(t_{i,x})$ can be obtained immediately (and hence its $r_\phi(t_{i,x})$ or $r_m(t_{i,x})$ is immediate). We denote this pruning technique as the *A-MQRank-Prune* algorithm.

4.7.5 Tuple-level uncertainty model

In the tuple-level model, we have additional challenges due to the presence of exclusion rules. We leverage the dynamic programming formulation that is similar in spirit to some of the existing work in the tuple-level model [89, 114].

Our approach begins by sorting all tuples by the *descending* order of their score values. Without loss of generality, we assume that for any $t_i, t_j \in \mathcal{D}, i \neq j$, if $i < j$, then t_i's score value is greater or equal than t_j's score value. Ties are broken arbitrarily or can be otherwise specified.

Let $\mathcal{D}_i$ be the database when $\mathcal{D}$ is *restricted* (both the tuples and the rules) on the first i tuples $\{t_1, \ldots, t_i\}$ for $i = 1, \ldots, N$, i.e., for each $\tau \in \mathcal{D}$, $\tau' = \tau \bigcap \{t_1, \ldots, t_i\}$ is included in $\mathcal{D}_i$. We first discuss the simplified tuple-level model where each rule contains just one tuple, i.e., every tuple is independent from all other tuples in the database. In this case, our idea is based on the following simple intuition. The probability that a tuple t_i appears at rank j depends only on the event that exactly j tuples from the first $i-1$ tuples appear, no matter *which* tuples appear. Now, let $R_{i,j}$ be the probability that a randomly generated world from $\mathcal{D}_i$ has exactly j tuples, i.e., $R_{i,j} = \sum_{|W|=j} \Pr[W|\mathcal{D}_i]$, and $R_{0,0} = 1$, $R_{i,j} = 0$ if $j > i$. Also, let $R^{-i}_{N-1,j}$ be the probability that a randomly generated world from $\mathcal{D} - \{t_i\}$ has exactly j tuples. Then, based on Definition 4.6, it is clear that the probability that t_i's rank equals to j in a randomly generated world from $\mathcal{D}$ is:

$$\Pr[R(t_i) = j] = p(t_i) \cdot R_{i-1,j} + (1 - p(t_i)) \cdot R^{-i}_{N-1,j}, \tag{4.14}$$

recalling $R(t_i)$ is a random variable denoting the rank for t_i in a random possible world. The final rank distribution $\text{rank}(t_i)$ is simply represented by the pairs, $(j, \Pr[R(t_i) = j])$, for $j = 0, \ldots, N-1$. For any tuple t_i, it is straightforward to calculate $R^{-i}_{N-1,j}$ for all js in a similar fashion as we compute $R_{i,j}$s. Our job is then to compute $R_{i,j}$s, which in this case is:

$$R_{i,j} = p(t_i) R_{i-1,j-1} + (1 - p(t_i)) R_{i-1,j}. \tag{4.15}$$

This gives us a dynamic programming formulation to calculate the $R_{i,j}$s. We can then effectively compute the rank distribution $\text{rank}(t_i)$ for the tuple t_i. Consequently, both the median rank and the ϕ-quantile rank for the tuple t_i can be easily obtained. We do this for each tuple in the database and return the k tuples with the smallest median ranks or the ϕ-quantile ranks as the answer to the top-k query.

The general case when there are multiple tuples in one rule is more complex. Nevertheless, in this case, the probability that t_i's rank is equal to j in a randomly generated world from $\mathcal{D}$ still follows the principle outlined in (4.14). However, $R_{i,j}$ can no longer be calculated as in (4.15), because if t_i has some preceding tuples from the same rule, the event that t_i appears is no longer independent of the event exactly $j-1$ tuples in $\mathcal{D}_{i-1}$ appear. Similarly, for the second term in (4.14), when t_i does not appear in a random world with j tuples, we may not simply multiply $(1-p(t_i))$ by the probability of this event: it could be one tuple from the same rule containing t_i has already appeared, which asserts t_i cannot appear at all.

To overcome this difficulty, we first convert $\mathcal{D}_i$ to a database $\bar{\mathcal{D}}_i$ where all rules contain only one tuple and apply the above algorithm on $\bar{\mathcal{D}}_i$ to compute $R_{i,j}$s. We construct $\bar{\mathcal{D}}_i$ as follows: For each rule $\tau \in \mathcal{D}_i$ and tuples in τ, we create one tuple $\bar{t}$ and one rule $\bar{\tau} = \{\bar{t}\} \in \bar{\mathcal{D}}_i$, where $p(\bar{t}) = \sum_{t\in\tau} p(t)$, with all of $\bar{t}$'s other attributes set to null. Essentially, $\bar{t}$ represents all tuples in a rule $\tau \in \mathcal{D}_i$. Now, the $R_{i,j}$ computed from $\bar{\mathcal{D}}_i$ is the same as the probability that exactly j tuples in $\mathcal{D}_i$ appear, because for $R_{i,j}$, we only care about the number of tuples appearing; merging does not affect anything since the probability that $\bar{t}$ appears is the same as the probability that one of the tuples in τ appears.

Now, we can compute all the $R_{i,j}$s, but another difficulty is the probability t_i's rank is equal to j if it appears is no longer simply $p(t_i)\cdot R_{i-1,j}$. This happens if t_i has some preceding tuples from the same rule in $\mathcal{D}_{i-1}$. Then, the existence of t_i has to exclude all these tuples, while $R_{i-1,j}$ includes the probability of the possible worlds that contain one of them. To handle this case, one can define $\mathcal{D}^-_{i-1} = \mathcal{D}_{i-1} - \{t | t \in \mathcal{D}_{i-1} \text{ and } t \in \tau \text{ and } t_i \in \tau\}$, i.e., $\mathcal{D}^-_{i-1}$ is the version of $\mathcal{D}_{i-1}$ that excludes all tuples from the same rule τ which contains t_i . Just as $R_{i-1,j}$ is defined with respect to $\mathcal{D}_{i-1}$, let $R^-_{i-1,j}$ be the probability exactly j tuples from $\mathcal{D}^-_{i-1}$ have appeared in a random possible world. We construct $\bar{\mathcal{D}}^-_{i-1}$ as follows: For each rule $\tau \in \mathcal{D}^-_{i-1}$ and tuples in τ, we create one tuple $\bar{t}$ and one rule $\bar{\tau} = \{\bar{t}\} \in \bar{\mathcal{D}}^-_{i-1}$, where $p(\bar{t}) = \sum_{t\in\tau} p(t)$, with all of $\bar{t}$'s other attributes set to null. Now, computing $R^-_{i,j}$ from $\bar{\mathcal{D}}^-_{i-1}$ is done in the same fashion as (4.15).

For the second term in (4.14), to cater for the case when a random world generated from database $\mathcal{D} - \{t_i\}$ has exactly j tuples and t_i does not appear, there are two cases. In case one, none of the j tuples in this random world is from the same rule that contains t_i; we denote this event's probability as $R^{-i-}_{N-1,j}$. In the second case, one of the j tuples in this random world comes from the same rule which contains t_i, and this event's probability is $R^{-i+}_{N-1,j}$. We can compute both $R^{-i-}_{N-1,j}$ and $R^{-i+}_{N-1,j}$ similarly as we compute the $R_{i,j}$s.

The probability t_i's rank in a random world equals j is:

$$\Pr[R(t_i) = j] = p(t_i)\cdot R^-_{i-1,j} + (1-p(t_i))\cdot R^{-i-}_{N-1,j} + R^{-i+}_{N-1,j}, \tag{4.16}$$

since $R^{-}_{i-1,j}$ already excludes all tuples from the same rule containing t_i. We calculate this probability for all js where $j = 0, \ldots, N-1$; then $(j, \Pr[R(t_i) = j])$s for $j = 0, \ldots, N-1$ is the rank distribution rank(t_i) for tuple t_i. Both the median rank and the ϕ-quantile rank can be easily obtained thereafter. The top-k answer could be easily obtained after calculating the median (or the quantile) rank for each tuple. We denote this algorithm as *T-MQRank*.

Example 4.2. *We consider tuple t_4 for the tuple-level uncertainty model example in Table 4.4. For t_4 the $R^{-}_{i-1,j}$s, where $i = 4$ and $j = 0, 1, 2, 3$ in this case, are $\{(0,0), (1,0.6), (2, 0.4), (3,0)\}$, the $R^{-i+}_{N-1,j}$'s are $\{(0,0), (1,0), (2,0.3), (3,0.2)\}$, and the $R^{-i-}_{N-1,j}$s are all zero as the probability that neither t_2 or t_4 appear is zero. Then,* rank(t_4) *is $\{(0,0), (1,0.3), (2,0.5), (3,0.2)\}$.*

4.7.6 Complexity of *T-MQRank*

In the basic case where each rule contains only one tuple, the cost of the dynamic program to compute the $R_{i,j}$s is $O(N^2)$, and one can compute $R^{-i}_{N-1,j}$s for all t_is similarly in $O(N^2)$ time. After which, for each t_i, obtaining the entire distribution rank(t_i) given $R_{i,j}$s and $R^{-i}_{N-1,j}$ is linear. Hence, the overall cost is $O(N^2)$.

In the second case when each rule may contain multiple tuples, we have to invoke the dynamic programming formulation for $R^{-}_{i-1,j}$ based on $\bar{D}^{-}_{i-1}$, which is different for each t_i. However, the size of the table for the dynamic programming formulation is only $O(M^2)$, where $M \leq N$ is the number of rules in the database. The cost of calculating $R^{-i-}_{N-1,j}$s and $R^{-i+}_{N-1,j}$s is also $O(M^2)$ for every tuple t_i. Hence, the overall cost of this algorithm is $O(NM^2)$.

4.8 Other Issues

4.8.1 Variance of the rank distribution: connection of expected ranks, median ranks, and quantile ranks

Implicit in all our discussions so far is that for any uncertain tuple t, there is a well-defined probability distribution rank(t) over its possible ranks within the uncertain relation. Within this setting, the expected rank, median rank, and quantile rank of t are the expectation, median, and quantile of this distribution, respectively. It is certainly meaningful to study other properties of these distributions. In particular, we next discuss the variance of the rank distribution rank(t), which we denote as var(t). A first observation is that var(t) is a good indicator to gauge the difference among $r(t)$ (expected rank), $r_m(t)$ (median rank), and $r_\phi(t)$ (quantile rank) for any quantile value ϕ. Intuitively, a small var(t) suggests that

$r(t)$ and $r_m(t)$ will be similar (in the extreme case, an extremely small var(t) suggests that even $r_\phi(t)$ for any ϕ value will be similar to $r(t)$). If all tuples in the database have small var(t)s, then ranking by expected ranks probably is a good choice. On the other hand, if a large number of tuples have large var(t) values, then ranking by expected ranks does not provide a good reflection on the ranks of various tuples. Rather, ranking by median ranks or some quantile ranks should be used.

It remains to show how to compute var(t) given an uncertain tuple t. Clearly, this task is trivial if one is provided with the rank distribution rank(t). Recall that our algorithms for ranking by median ranks or quantile ranks in Section 4.7, in both uncertain models, work by firstly computing the rank distribution for any given tuple in the database, then deriving the median (or the quantile) from the computed rank distribution. So, as a straightforward extension, these algorithms can easily support the calculation of var(t) for any tuple t as well. It is an open problem to find more direct ways to compute var(t), but the existence of correlations between tuples suggests that this may not be more efficient than simply computing the rank distribution.

4.8.2 Scoring functions

Our analysis has assumed that the score is a fixed value. In general, the score can be specified at query time by a user-defined function. Note that all of our offline algorithms (for expected ranks, median, and quantile ranks) also work under this setting, as long as the scores can be computed. If the system has some interface that allows us to retrieve tuples in the score order (for the tuple-level order) or in the expected score order (for the attribute-level model), our pruning algorithms for expected ranks are applicable as well.

A main application of a query-dependent scoring function is k-nearest-neighbor queries, which is the top-k query instantiated in spatial databases. Here, the score is implicitly the distance of a data point to a query point. When the data points are uncertain, the distance to the query is a random variable, which can be modeled as an attribute-level uncertainty relation. Existing works [105,115] essentially adopt U-kRanks semantics to define k-nearest-neighbor queries in spatial databases. We believe that our ranking definition makes a lot of sense in this context, and may have similar benefits over previous definitions of uncertain nearest neighbors.

When a relation has multiple (certain and uncertain) attributes on which a ranking query is to be performed, the user typically will give some function that combines these multiple attributes together and then rank on the output of the function. When at least one of the attributes is uncertain, the output of the function is also uncertain. This gives us another instance where our ranking semantics and algorithms could be applied.

4.8.3 Continuous distributions

When the input data in the attribute-level uncertainty model are specified by a continuous distribution (e.g., a Gaussian or Poisson), it is often hard to compute the probability that one variable exceeds another. However, by discretizing the distributions to an appropriate level of granularity (i.e., represented by a histogram), we can reduce to an instance of the discrete pdf problem. The error in this approach is directly related to the granularity of the discretization. Moreover, observe that our pruning-based methods initially require only information about expected values of the distributions. Since continuous distributions are typically described by their expected value (e.g., a Gaussian distribution is specified by its mean and variance), we can run the pruning algorithm on these parameters directly.

4.8.4 Further properties of a ranking

It is certainly reasonable to define further properties and analyze when they hold. However, formulating the right properties can be tricky. For example, Zhang and Chomicki [87] defined a property of "faithfulness," which demands that (in the tuple-level model), given two tuples $t_1 = (v_1, p(t_1))$ and $t_2 = (v_2, p(t_2))$ with $v_1 < v_2$ and $p(t_1) < p(t_2)$, then $t_1 \in R_k \Rightarrow t_2 \in R_k$. This intuitive property basically says that if t_2 "dominates" t_1, then t_2 should always be ranked at least as high as t_1. It was claimed that the Global-Top$k$ definition satisfies this property; however, this only holds in the absence of exclusion rules. There are cases with exclusions where all existing definitions fail on faithfulness. Consider the following example:

t_i	t_1	t_2	t_3	t_4	t_5
v_i	1	2	3	4	5
$p(t_i)$	0.4	0.45	0.2	0.2	0.2

with rules $\tau_1 = \{t_1, t_3, t_4, t_5\}, \tau_2 = \{t_2\}$. The possible worlds of this relation is shown in Table 4.6. Here, t_2 "dominates" t_1, but all of the previous definitions (U-Top$k$, U-kRanks, Global-Top$k$, and PT-k) will select t_1 as the top-1. On this example, ranking by expected ranks will rank t_2 as the top-1, hence satisfying the "faithfulness" requirement. However, it is easy to construct other examples where the expected rank will also rank a dominating tuple lower than a dominated tuple. Consider the example shown in Table 4.7, the expected rank of t_1 is $0.105 \times 1 + 0.245 \times 1 + 0.455 \times 1 = 0.805$. However, the expected rank of t_2 is $0.195 \times 2 + 0.455 \times 1 = 0.845$. Hence, t_2 ranks after t_1 even though t_2 dominates t_1. Our

Table 4.6. Possible worlds where previous definitions do not satisfy faithfulness. ©2011 IEEE

0.18	
t_2	2
t_1	1

0.09	
t_3	3
t_2	2

0.09	
t_4	4
t_2	2

0.09	
t_5	5
t_2	2

0.22	
t_1	1

0.11	
t_3	3

0.11	
t_4	4

0.11	
t_5	5

Table 4.7. An example where the expected rank does not satisfy the faithfulness. ©2011 IEEE

tuples	score	$p(t)$
t_1	1	0.3
t_2	2	0.35
t_3	0.5	0.65

rules	
τ_1	$\{t_1\}$
τ_2	$\{t_2, t_3\}$

0.105	
t_2	2
t_1	1

0.195	
t_1	1
t_3	0.5

0.245	
t_2	2

0.455	
t_3	0.5

initial study suggests that "faithfulness" defined this way may not be achievable, and one has to somehow take rules (i.e., correlations) into consideration in order to make it a viable.

4.8.5 Rank of missing tuples

In the tuple-level uncertainty model, we chose to determine the rank of tuples not present in a world W as $|W|$. This is an intuitive choice (the missing tuples were all ranked "equal last"), but other choices are possible. A different approach is to compute the rank of a tuple only over those worlds W where it does appear, and then to scale this by the probability that it appears. This can be understood in terms of conditional probabilities: for the expected rank definition, we compute the rank of a tuple t by summing the probabilities that other tuples appear which score more highly than t. This can be viewed as the probability that each tuple t' outranks t and t appears; dividing this by $p(t)$ gives the conditional probability t' outranks t given that t appears. Accordingly, we can call this the "conditional expected rank" of t_i, $e(t_i)$. Formally, adopting the convention that the tuples are indexed in decreasing order of their score, $e(t_i) = \frac{1}{p(t_i)}(1 + \sum_{j<i, t_j \bar{\diamond} t_i} p(t_j))$, which can be computed in constant time for each t_i after computing prefix sums on the $p(t_i)$'s and on the probabilities associated with each rule. On the example in Table 4.4, we obtain $e(t_1) = 1/0.4 = 2.5$, $e(t_2) = (1 + 0.4)/0.5 = 2.8$, $e(t_3) = (1 + 0.4 + 0.5)/1 = 1.9$ and $e(t_4) = (1 + 0.4 + 1)/0.5 = 4.8$. This yields the ranking (t_3, t_1, t_2, t_4), the same as under the expected rank semantics. Likewise, the properties of exact-k, containment, unique-rank, and value-invariance follow immediately. Stability also follows easily: increasing the score of a tuple t cannot increase the sum of probabilities of tuples with lower scores, while

increasing its probability drives down $1/p(t)$, so either way $e(t)$ cannot increase, ensuring that if it was in the top-k before, it will remain so after. We leave further study of this alternate semantics to future work.

4.8.6 Parametrized ranking function

In parallel to this work, a new ranking framework for probabilistic data was proposed [93], namely, the parametrized ranking function (PRF). It is interesting to note that PRF adopts a similar basis to that shown in this work for ranking probabilistic data. Essentially, the basis of ranking in PRF is also the rank distributions for different tuples. However, instead of using the expectations, medians, or quantiles of these ranking distributions to derive a total ordering of tuples, Li et al. proposed that any parametrized function may be defined over the rank distributions of tuples, i.e., the final rank value of a tuple $t \in \mathcal{D}$, where $|\mathcal{D}| = N$, could be defined as $\sum_{i=0}^{N-1} \omega(t,i) \Pr[R(t) = i]$, where $\{\omega(t,0), \ldots, \omega(t, N-1)\}$ is a set of $(N-1)$ user-defined parametrized functions. Clearly, the basis for the above ranking definition is $\Pr[R(t) = i]$ for $i = \{0, \ldots, N-1\}$, which is nothing else but the rank distribution of t, $\text{rank}(t)$. Note that the PRF is a framework for ranking, but not a ranking definition by itself. Many ranking definitions are possible to be defined in the PRF framework; for example, it is indeed possible to define the expected rank in this paper under the PRF framework. However, it is not feasible to directly define the median and quantile ranks using the PRF mechanism. Nevertheless, one may extend the PRF framework to support the median and quantile ranks when its ranking definitions are no longer constrained by using only parametrized functions.

Since the ranking basis is the rank distributions for tuples, when $\omega(t,i)$ is independent from t for all $i \in \{0, \ldots, N-1\}$ (which is prevalent as also noted in [93]), all such rankings within the PRF framework necessarily follow value-invariance. Different instantiations of the parametrized function $\omega(t,i)$ in the PRF framework introduces quite different definitions. It is an intriguing and challenging open problem to further study the rich semantics and properties the PRF framework imposes on various ranking definitions extended from it.

4.8.7 Approximate expected ranks

The rank distribution of an uncertain tuple follows the Poisson Binomial distribution. It follows from standard results in statistics [116, 117] the expected rank of a tuple t is approximated by the sum of probabilities of the tuples ranked before t in the tuple-level model. Since the focus of this work is to rank the uncertain tuples based on their *exact* expected ranks (or median or quantile ranks for that purpose), i.e., we are interested at computing the *exact* top-k results in each framework (be it expected ranks, median ranks, or quantile ranks), we do not pursue this observation further in this presentation. We leave further study of approximate top-k ranks of uncertain data to future work. We observe the key challenges to address are to design approximation schemes for broader models of uncertain data, and to quantify the quality of the approximation over all possible uncertain relations, or else to describe particular cases of uncertain relations which are guaranteed to be well-approximated. The hope is an approximation which compromises on finding the exact answer can compensate with computational efficiency.

4.9 Experiments

We implemented our algorithms in GNU C++. All experiments were executed on a Linux machine with a 2GHz CPU and 2GB main memory. We utilized synthetic datasets to study the impact of datasets with different characteristics on both the score and the probability distribution. We additionally tested our algorithms on a real dataset, *movie*, from the MystiQ project. *movie* consists of data integrated from IMDB and Amazon. Each record in *movie* consists of an IMDB id (*imdbid*), IMDB string, Amazon id (*aid*), Amazon string, and a probability, where the pair (*imdbid*,*aid*) is unique. We converted *movie* to the attribute-level and tuple-level models by grouping records by the *imdbid* attribute. To obtain an attribute-level representation, we created a single tuple for each *imdbid* group where the pdf consists of all of the *aid* in the group. To obtain a tuple-level representation, we created a tuple for each *aid* in an *imdbid* group and we also created an exclusion rule (τ) for each *imdbid* group consisting of all *aid* in the group. For both models, we rank tuples by the *aid* attribute. The *movie* dataset consists of 246,816 unique (*imdbid*, *aid*) pairs and there are 56,769 unique *imdbid* groups with an average of 4.35 *aid* per group. For the experiments, we vary only N and k for *movie*, where N is varied by uniformly and randomly selecting tuples. To generate synthetic datasets, we developed several data generators for both models. Each generator controls the distribution on the score value as well as the probability. For both models, these distributions refer to the *universe of score values and probabilities* when we take the union of all tuples in $\mathcal{D}$. The distributions used include *uniform, Zipfian*, and *correlated bivariate*. They are abbreviated as *u, zipf*, and *cor*. For each tuple, we draw a score and probability value independently from the score distribution and probability distribution, respectively. We refer to the result of drawing from these two distributions by the concatenation of the short names for each distribution for score then probability, i.e. *zipfu* indicates a Zipfian distribution of scores and uniform distribution of probabilities. The default skewness for the Zipfian distribution is 1.2, and other default values are $k = 100$, $N = 10,000$, $s = 5$, $\psi = 5$, and $\zeta = 30\%$.

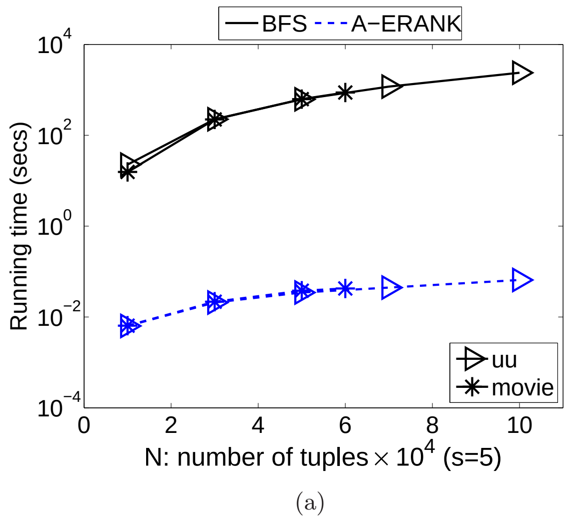
4.9.1 Expected ranks

4.9.1.1 Attribute-level uncertainty model

We first studied the performance of the exact algorithm A-ERank by comparing it to the basic brute-force search (BFS) approach on *uu* and *movie*. The distribution on the probability universe does not affect the performance of either algorithm, since both algorithms calculate the expected ranks of all tuples. The score value distribution has no impact on BFS, but does affect A-ERank: the uniform score distribution results in the worst performance given a fixed number of tuples, as it leads to a large set of possible values. So, amongst the synthetic datasets, we only consider *uu* for this experiment, to give the toughest test for this algorithm.

Figure 4.1(a) shows the total running time of these algorithms as the size of $\mathcal{D}$ (the number of tuples, N) is varied, up to $100,000$ tuples. Note we can only vary *movie* up to $N = 56,769$ for this and the following attribute-level experiments. A-ERank outperforms BFS by up to 6 orders of magnitude. This gap grows steadily as N gets larger. A-ERank has very low query cost: it takes only about 10ms to find the expected ranks of all tuples for $N = 100,000$, while the brute force approach takes 10 minutes. Results are similar for other values of s.

As discussed in Section 4.5.2, A-ERank-Prune is an approximate algorithm, in that it may not find the exact top-k. Figure 4.1(b) reports its approximation quality on various datasets using the standard *precision* and *recall* metrics. Since A-ERank-Prune always returns k tuples, its recall and precision are always the same. Figure 4.1(b) shows it achieves high approximation quality: recall and precision are near the 100th percentile for *movie* and in the 90th percentile when the score is distributed uniformly for the synthetic datasets. The worst case occurs when the data are skewed on both dimensions, where the potential for pruning is greatest. The reason for this is that as more tuples are pruned, these unseen tuples have a greater chance to affect the expected ranks of the observed tuples. Even though the pruned tuples all have low expected scores, they could still have values with high probability to be ranked above some seen tuples, because of the heavy tail of their distribution. Even in this worst case, the recall and precision of T-ERank-Prune is about 80%.



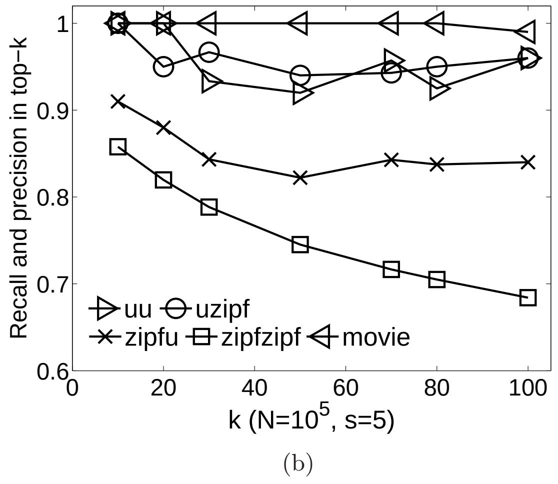


Figure 4.1. Attribute-level model performance analysis: (a) running time of exact algorithms and (b) A-ERank-Prune's precision/recall. ©2011 IEEE

Figure 4.2 shows the pruning power of A-ERank-Prune. In this experiment $N = 100,000$, $s = 5$, and k is varied from 10 to 100. It shows we often only need to materialize a small number of tuples of $\mathcal{D}$ (ordered by expected score) before we can be sure we have found the top-k, across a variety of datasets. Intuitively, a more skewed distribution on either dimension should increase the algorithm's pruning power. This intuition is confirmed by results in Figure 4.2. When both distributions are skewed, A-ERank-Prune could halt the scan after seeing less than 20% of the relation. For the *movie* dataset, A-ERank-Prune is also effective and prunes almost 50% of the tuples. Even for more uniform distributions such as *uu*, expected scores hold enough information to prune.

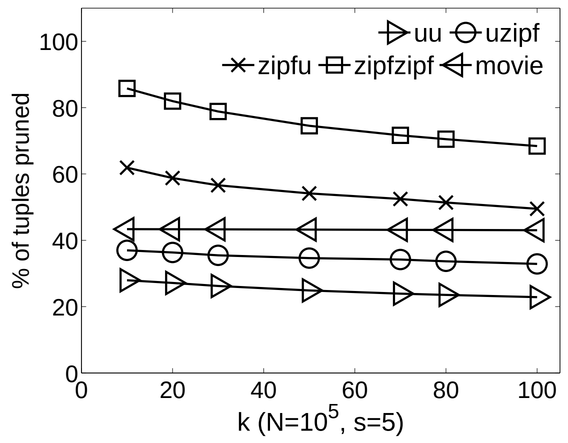


Figure 4.2. Attribute-level: Pruning of A-ERank-Prune. ©2011 IEEE

4.9.1.2 Tuple-level uncertainty model

For our experiments in the tuple-level model, we first investigate the performance of our algorithms. As before, there is a brute-force search-based approach which is much more expensive than our algorithms, so we do not show these results.

A notable difference in this model is that the pruning algorithm is able to output the exact top-k, provided $\mathbf{E}[|W|]$, the expected number of tuples of $\mathcal{D}$, is known. Figure 4.3(a) shows the total running time for the T-ERank and T-ERank-Prune algorithms on *uu* and *movie*. Both algorithms are extremely efficient. For $100,000$ tuples, the T-ERank algorithm takes less than 100 milliseconds to compute the expected ranks of all tuples; applying pruning, T-ERank-Prune finds the same k smallest ranks in just 1 millisecond. However, T-ERank is still highly efficient, and is the best solution when $\mathbf{E}[|W|]$ is unavailable.

Figure 4.3(b) shows the pruning power of T-ERank-Prune for different datasets. We fix $N = 100,000$ and vary k. T-ERank-Prune prunes more than 98% of tuples for all k for *movie*, which shows T-ERank-Prune may be very effective at pruning real data for certain applications. We also see a skewed distribution on either dimension increases the pruning capability of T-ERank-Prune. Even in the worst case of processing *uu*, T-ERank-Prune is able to prune more than 90% of tuples.

Our next experiments study the impact of correlations between a tuple's score value and probability. We say the two are positively correlated when a tuple with a higher score value also has a higher probability; a negative correlation means a tuple with a higher score value has a lower probability. Such correlations have no impact on the performance of T-ERank as

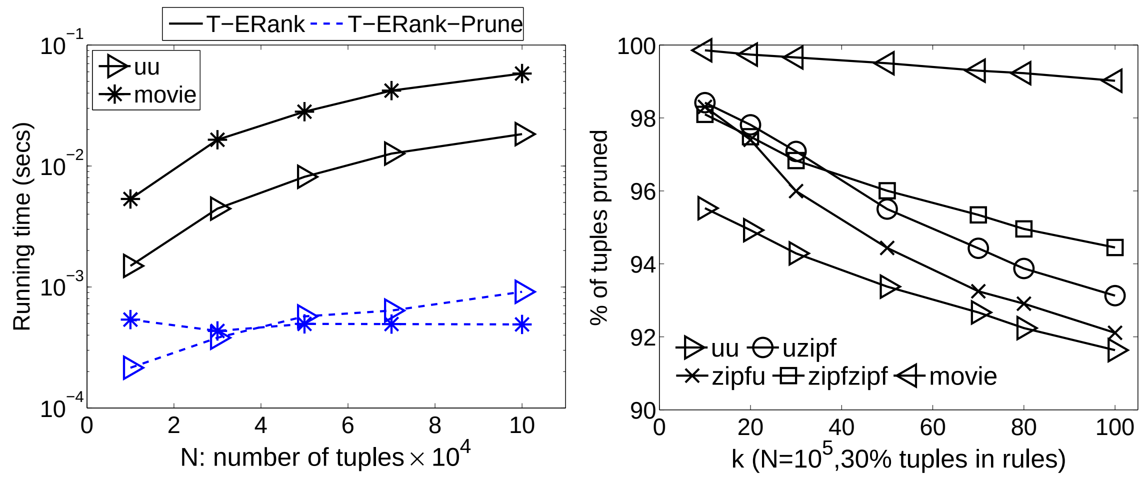


Figure 4.3. Tuple-level model performance analysis: (a) running time and (b) pruning of T-ERank-Prune. ©2011 IEEE

it computes the expected ranks for all tuples. However, correlation does have an interesting effect on the pruning capability of T-ERrank-Prune. Using correlated bivariate datasets of different correlation degrees, Figure 4.4(a) repeats the pruning experiment for T-ERank-Prune with $N = 100,000$. The strongly positively correlated dataset with a $+0.8$ correlation degree allows the highest amount of pruning, whereas the strongly negatively correlated dataset with a -0.8 correlation degree results in the worst pruning power. However, even in the worst case, T-ERank-Prune still pruned more than 75% of tuples. Figure 4.4(b) reflects the running time of the same experiment. T-ERank-Prune consumes between 0.1 and 5 milliseconds to process $100,000$ tuples.

4.9.2 Median and quantile ranks

Our primary concern when evaluating A-MQRank, A-MQRank-Prune, and T-MQRank was the time necessary to retrieve the top-k from an uncertain database. To evaluate the query time for A-MQRank and T-MQRank, we utilize *uu* and *movie*. The results for other datasets are similar as the computational effort imposed by A-MQRank and T-MQRank are the same regardless of the distribution as both algorithms compute the median or quantile rank for every tuple in the database. We utilize *movie*, *uu*, *uzipf*, *zipfzipf*, and *zipfu* to show the effect different distributions have on the pruning power of A-MQRank-Prune. We also analyze the effect positively correlated and negatively correlated datasets have on the processing time of A-MQRank-Prune. For all experiments, we present the results from calculating the median ranks of all tuples, as other general quantiles perform similarly.

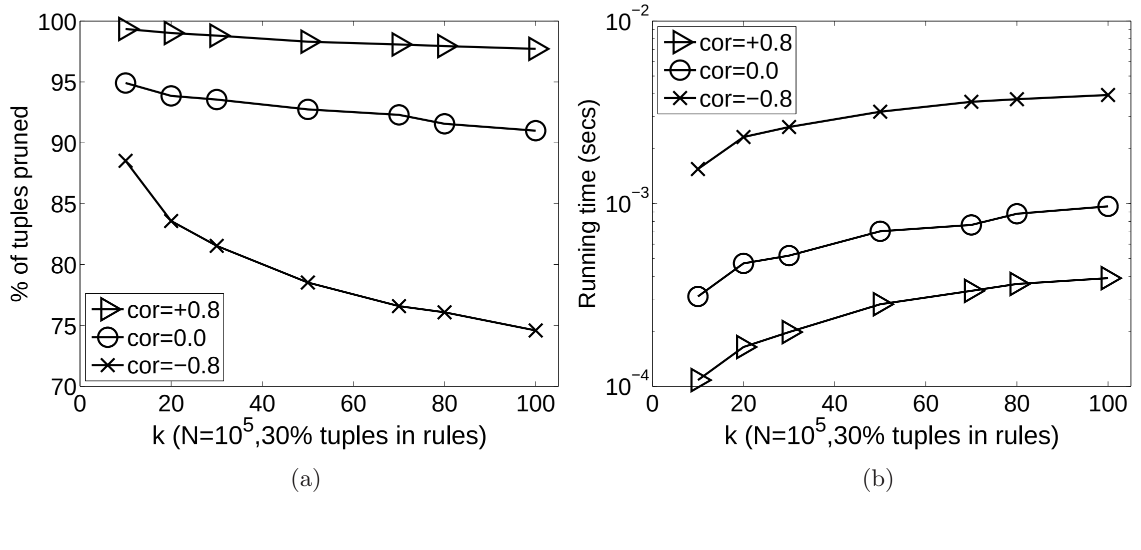


Figure 4.4. Impact of different correlations on T-ERank-Prune: (a) pruning power and (b) running time. ©2011 IEEE

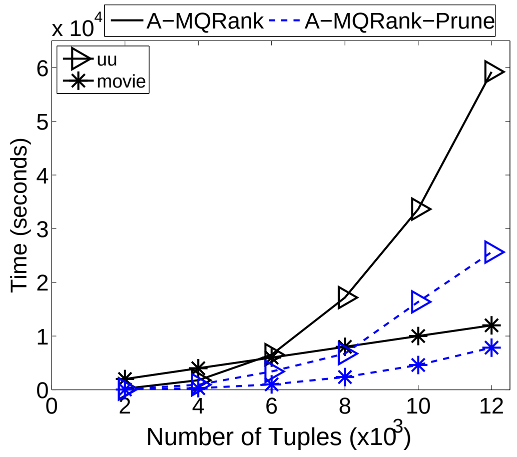
4.9.2.1 Attribute-level uncertainty model

We first analyze the effects of varying the number of tuples N with respect to the processing time for A-MQRank and A-MQRank-Prune in Figure 4.5(a). Recall A-MQRank and A-MQRank-Prune are both $O(sN^3)$. However, the pruning techniques utilized in A-MQRank-Prune could allow the algorithm to avoid performing the $O(N^2)$ dynamic program for all the s unique entries in the pdf of a tuple. The effects of this pruning are apparent in Figure 4.5(a). We utilize *movie* and *uu* and vary N from 2,000 to 12,000. The processing time for *movie* is less than that of *uu* for both algorithms as s is on average 4 in *movie* whereas in *uu*, $s = 5$ for all tuples. In all cases A-MQRank-Prune requires less processing time. In Figure 4.5(b) we analyze the effect of varying s for A-MQRank and A-MQRank-Prune. Here we use only *uu* as we cannot vary s in *movie*. In this experiment, s is varied from 2 to 10. Again we see the pruning of A-MQRank-Prune performs very well, effectively reducing the constant in the $O(sN^3)$ complexity of A-MQRank.

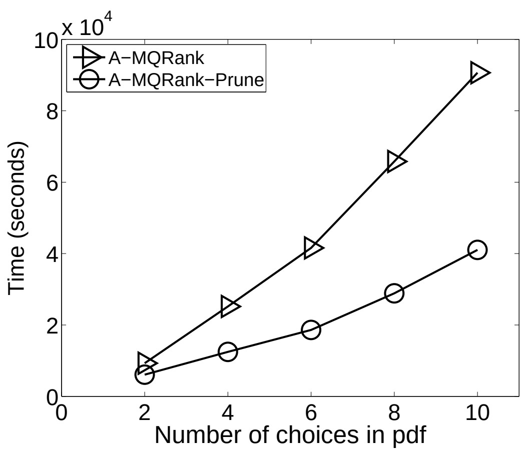
We next analyze the effect different distributions have on the pruning power of A-MQRank-Prune. In Figure 4.5(c), we study the effects of varying k over *movie*, *uu*, $zipfzipf$, $uzipf$, and $zipfu$ on A-MQRank-Prune. We see in general the processing time grows almost linearly with respect to k. It is apparent the pruning utilized by A-MQRank-Prune works best for scores which are uniformly distributed and probabilities which follow a zipfian distribution. Regardless of the distribution A-MQRank-Prune shows excellent scalability with respect to k. In Figure 4.5(d), we analyze the effect of varying k over the positively and negatively correlated datasets. In general, we see correlations of the dataset do not affect the pruning of A-MQRank-Prune.

4.9.2.2 Tuple-level uncertainty model

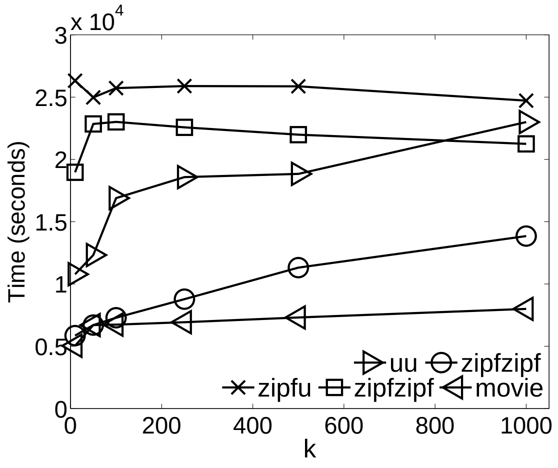
We evaluate the effect of varying the number of tuples N in the database and the percentage of tuples from the database which share a rule with another tuple ζ in Figure 4.6. Note the value selected for k is irrelevant as the quantile rank for every tuple is computed. In Figure 4.6(a), we see the amount of time required to determine the median ranks increases quadratically with respect to the number of exclusion rules M in the database, since M clearly increases as N increases. This quadratic relationship makes sense as the running time for T-MQRank is $O(NM^2)$. Also, note *movie* has about 3 times as many rules as *uu*, which explains why it is about 9 times more expensive. We also evaluate the effect of varying the percentage of tuples which share a rule with another tuple in Figure 4.6(b). As expected, the amount of time required to calculate the median ranks drops quadratically as the percentage increases. This is clear since as the percentage of tuples which share a rule



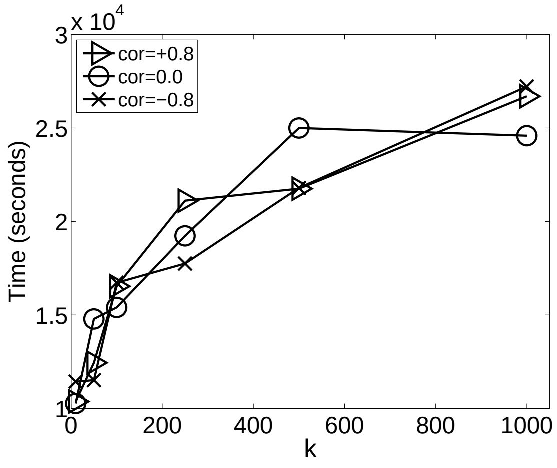
(a)



(b)

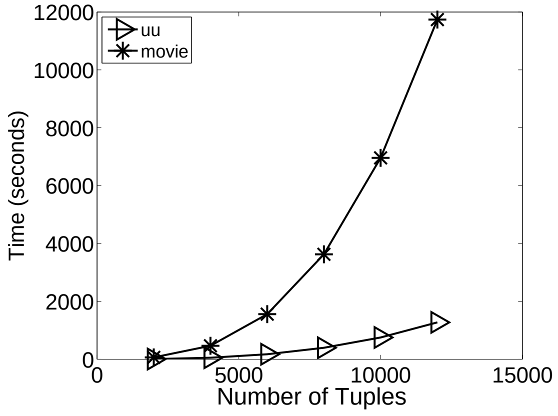


(c)

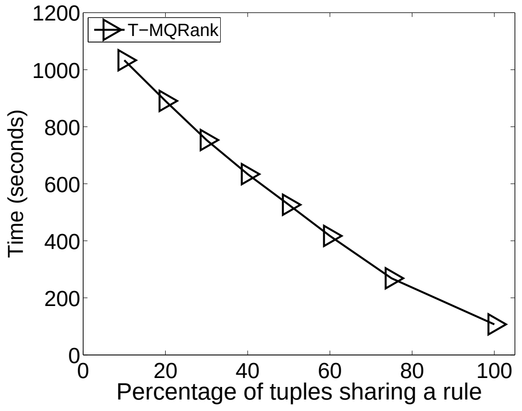


(d)

Figure 4.5. Attribute-level model median and quantile ranks performance: effect of (a) N, (b) s, (c) k while varying distributions on A-MQRank-Prune, (d) k while varying correlated distributions on A-MQRank-Prune. ©2011 IEEE



(a)



(b)

Figure 4.6. Tuple-level model median and quantile ranks performance: effect of (a) N and (b) ζ. ©2011 IEEE increases, there are fewer tuples in rules by themselves, and therefore, M decreases.

Figure 4.7 shows that increasing the number of tuples in a rule, while still requiring $\zeta = 30\%$, also reduces the amount of time to compute median ranks. This is explainable by the fact that increasing the number of tuples in any rule τ will clearly cause the total number of exclusion rules in the database to decrease. It follows that the processing time for T-MQRank will be very dependent on N, ζ, and ψ. It is likely M will only be a constant factor smaller than N for any uncertain database. However, as N increases, this reduction factor will become increasingly significant in terms of the computational effort required by T-MQRank. In any case, T-MQRank still has a low polynomial time cost with respect to N and M.

4.9.3 Comparison of expected and median ranks

We have shown retrieving the top-k from an uncertain database following either the attribute-level or tuple-level model may be computed efficiently in $O(N \log N)$ time when ranking by expected ranks. We have also presented algorithms to compute median and quantile ranks for a database following the attribute-level and tuple-level model in $O(sN^3)$ and (NM^2) time, respectively. It is evident not only from our experiments but also from the corresponding complexities for the expected rank and median and quantile rank algorithms that retrieving the top-k tuples from an uncertain database using expected ranks may require much less computational effort than median or quantile ranks. It is clear from Figure 4.5 and Figure 4.6 that determining the top-k from both attribute-level and tuple-level uncertain databases using median ranks requires on the order of 10^4 seconds. In comparison,

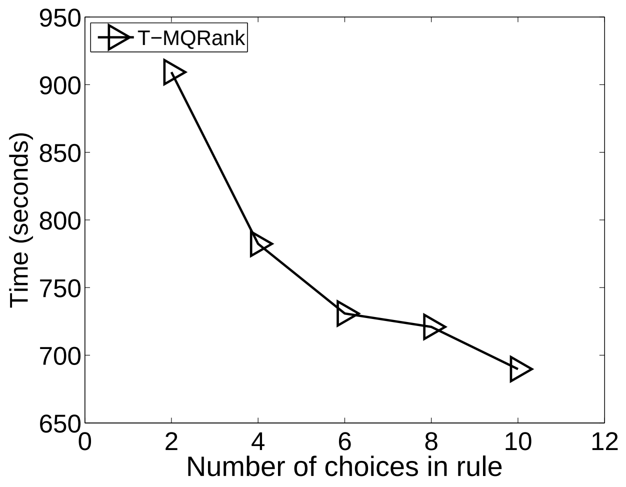


Figure 4.7. Tuple-level: Effect of ψ for median and quantile ranks. ©2011 IEEE

from Figure 4.1 and Figure 4.3, we can see we need less than 1 second to determine the top-k for both attribute-level and tuple-level uncertain databases utilizing the expected ranks. It is not surprising that ranking by median ranks requires more computational effort than ranking by expected ranks since we must compute the rank distribution $\text{rank}(t_i)$ for every $t_i \in \mathcal{D}$ in order to determine the median ranks. To do this, we rely on dynamic programs with quadratic complexities. However, it has been commonly observed that calculating the median or quantile values for a distribution is more expensive than computing the expectation of the distribution.

We also compared the similarity between the top-k lists returned by ranking with the median ranks and ranking with the expected ranks. We adopted the techniques from [112] for this purpose. Specifically, for two top-k lists τ_1 and τ_2, we use the *averaging Kendall distance* to measure their similarity, denoted as $K_{avg}(\tau_1, \tau_2)$. $K_{avg}(\tau_1, \tau_2)$ is computed as

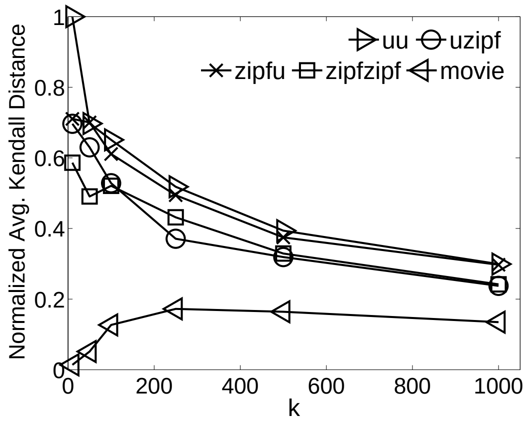
$$K_{avg}(\tau_1, \tau_2) = \sum_{\{i,j\} \in P(\tau_1, \tau_2)} \bar{K}^{(p)}_{i,j}(\tau_1, \tau_2), \text{for } p = 0.5 \tag{4.17}$$

where $\bar{K}^{(p)}_{i,j}(\tau_1, \tau_2)$ is defined as a penalty over the pairs in the set $P(\tau_1, \tau_2) = \{\{i, j\} | i \neq j \text{ and } i, j \in \tau_1 \cup \tau_2\}$, i.e., $P(\tau_1, \tau_2)$ is the set of unordered pairs of distinct elements in $\tau_1 \cup \tau_2$. The exact details of how the penalty $\bar{K}^{(p)}_{i,j}(\tau_1, \tau_2)$ is assigned for different pairs in the set P are found in [112]. A larger $K_{avg}(\tau_1, \tau_2)$ value indicates a higher *dissimilarity* between two top-k lists τ_1 and τ_2. By examining the assignment of the penalty to possible pairs in P, we can show that $K_{avg}(\tau_1, \tau_2) \in [0, k^2 + \binom{k}{2}]$ for any two top-k lists τ_1 and τ_2. The smallest value for $K_{avg}(\tau_1, \tau_2)$ happens when τ_1 and τ_2 are identical as two ordered sets; and the largest value for $K_{avg}(\tau_1, \tau_2)$ happens when τ_1 and τ_2 are completely disjoint. Hence, a meaningful way to represent the similarity between any two top-k lists, τ_1 and τ_2, is to use the *normalized averaging Kendall distance*, which is defined as:

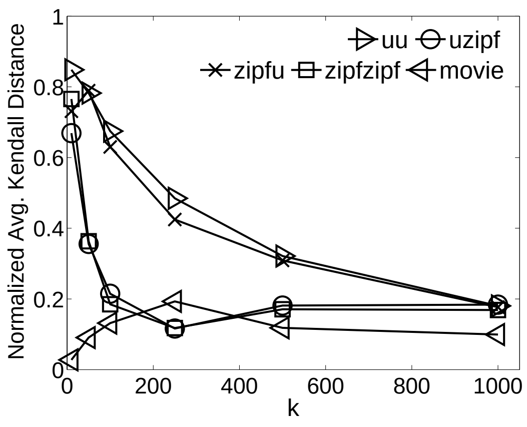
$$K_{navg}(\tau_1, \tau_2) = \frac{K_{avg}(\tau_1, \tau_2)}{k^2 + \binom{k}{2}} \tag{4.18}$$

Clearly, $K_{navg}(\tau_1, \tau_2) \in [0, 1]$. Smaller $K_{navg}(\tau_1, \tau_2)$ values indicate higher similarity between τ_1 and τ_2, and larger values indicate lower similarity.

In Figures 4.8(a) and 4.8(b), we compare the similarity between the top-k results returned from the median ranks and expected ranks for both the attribute-level and tuple-level uncertainty models, using the normalized averaging Kendall distance. It is clear from the results in Figure 4.8(a) and 4.8(b) that the top-k lists produced by the median ranks and the expected ranks are rather different for both the attribute-level and the tuple-level uncertainty models, especially when k is small for the synthetic datasets. In general, the



(a)

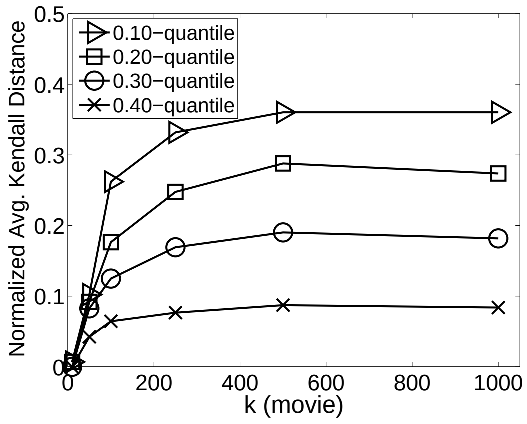


(b)

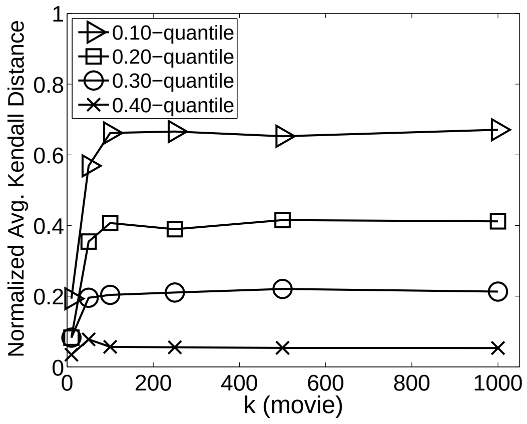
Figure 4.8. $K_{avg}(\tau_1, \tau_2)$ for expected and median ranks with different k values: following (a) attribute-level uncertainty model and (b) tuple-level uncertainty model. ©2011 IEEE

similarity between their top-k lists increases while k increases, but still maintains a clear difference. This shows that ranking by median and quantile ranks or by expected ranks will arrive at a different view of the top-k. This result is quite natural since median (quantile) ranks and expected ranks characterize different characteristics of the rank distributions rank(t_i) for all $t_i \in \mathcal{D}$, i.e., the 0.50-quantile (or other quantile values) and expectation of rank(t_i).

In Figures 4.9(a) and 4.9(b) we compare the similarity between the top-k results returned from the median ranks and different quantile ranks for the *movie* dataset for both the



(a)



(b)

Figure 4.9. $K_{avg}(\tau_1, \tau_2)$ for median and quantile ranks with different k values: following (a) attribute-level uncertainty model and (b) tuple-level uncertainty model. ©2011 IEEE attribute-level and tuple-level uncertainty models, again using the normalized averaging Kendall distance. From these results, we see that the similarity of the top-k lists produced by different quantile ranks and median ranks behaves in a very stable manner. As k increases from 0 to about 200, the similarity decreases quadratically between the different quantile ranks and the median ranks and when $k > 200$, we see that the similarity between median ranks and different quantile ranks remains roughly the same. Notice for all k values, as the quantile approaches the median, the normalized averaging Kendall distance approaches 0.

4.10 Conclusion

In this chapter, we study the challenge of *uncertain data*, which is becoming ubiquitous in many applications, and how to summarize it using the *ranking operator*. We have studied semantics of ranking queries in probabilistic data. We adapt important properties that guide the definition of ranking queries in deterministic databases and analyze characteristics of existing top-k ranking queries for probabilistic data. These properties naturally lead to the ranking approach that is based on the rank distribution for a tuple across all possible worlds in an uncertain domain. Efficient algorithms for two major uncertainty models ensure the practicality of our approach. Our experiments demonstrate that ranking by expected ranks, median ranks and quantile ranks is efficient in both attribute-level and tuple-level uncertainty models.

CHAPTER 5

RANKING DISTRIBUTED PROBABILISTIC DATA

5.1 Introduction

In Chapter 4, we studied the challenge of *uncertain data* by observing its semantics, which drove us to propose novel *ranking operators* to summarize it, including the expected, median, and quantile ranks, where we focused only on the centralized case. In this chapter, we study the emerging challenge of *distributed data*, extending our study in Chapter 4 to the distributed and parallel setting using the *expected ranks* ranking operator to summarize massive distributed uncertain data, e.g., such as data collected in sensor networks as well as data generated by scientific applications and stored in distributed file systems within clusters.

Data are increasingly stored and processed distributively as a result of the wide deployment of computing infrastructures and the readily available network services [28, 119–124]. More and more applications collect data in a distributive fashion and derive results based on the collective view of all of the distributed data. Examples include sensor networks, data integration from multiple data sources, information retrieval from geographically separated data centers, as well as data partitioned and stored in a distributed file system within a cluster. In the aforementioned application domains, it is often very expensive, impractical, or sometimes even impossible to communicate the distributed dataset entirely to a centralized server for processing, e.g., by using techniques discussed in Chapter 4 at the centralized server, due to the large amounts of data available nowadays and the network delay incurred, as well as the economic cost associated with such communication [27]. Fortunately, query semantics in many such applications rarely require reporting every piece of data in the system. Instead, only a fraction of data that are the most relevant to the user's interest

*The work in this chapter appears in [118] ©2009 Association for Computing Machinery, Inc. Reprinted by permission.

will appear in the query results. Typically, a ranking mechanism is implemented and only the top-k records are needed [27, 72], e.g., displaying the sensor ids with the k highest temperature readings [125,126], or retrieving the images with the k largest similarity scores to a predefined feature [127].

This observation deems that expensive communication is unnecessary and could be avoided by designing communication efficient algorithms. Indeed, a lot of efforts have been devoted to this subject, from both the database and the networking communities [27, 28, 72, 119–123]. However, none of these works deals with distributed queries on probabilistic data, which has emerged as a principled data type in many applications. Interestingly enough, many cases where uncertainty arises are distributed in nature, e.g., distributed sensor networks with imprecise measurements [80], multiple data sources for information integration based on fuzzy similarity scores [81, 85]. Probabilistic data encode an exponential number of possible instances, and ranking queries, by focusing attention on the most representative records, are arguably more important and useful as a data summarization technique in such contexts. Not surprisingly, top-k and ranking queries on probabilistic data have quickly attracted a lot of interest, as discussed in Chapter 4. However, prior to our work in this chapter, no work addressed the problem of ranking uncertain data in a parallel and distributed setting. To address this important issue, our focus in this chapter is to answer top-k queries in a communication-efficient manner on probabilistic data from multiple, distributed sites using the *expected ranks* ranking operator.

The focus of uncertain query processing is (1) how to "combine" the query results from all the possible worlds into a meaningful result for the query; and (2) how to process such a combination efficiently without explicitly materializing the exponentially many possible worlds. Additionally, in a distributed environment, we also face the third challenge: (3) how to achieve (1) and (2) with the minimum amount of communication. This chapter concentrates on retrieving the top-k tuples with the smallest expected ranks from m distributed sites that collectively constitute the uncertain database $\mathcal{D}$. The challenge is to answer these queries both computation- *and* communication-efficiently.

5.1.1 Our contributions.

We study ranking queries for distributed probabilistic data. We design communication efficient algorithms for retrieving the top-k tuples with the smallest ranks from distributed sites, with computation overhead also as a major consideration as well. In summary, our contributions are as follows:

- We formalize the problem of distributed ranking queries in probabilistic data (Section 5.2), and argue that the straightforward solution is communication-expensive.
- We first provide a basic approach using only a tuple's local rank (Section 5.3). Then, we introduce the sorted-access framework based on expected scores (Section 5.4). The Markov inequality is first applied, followed by an improvement that formulates a linear programming optimization problem at each site, which results in significantly less communication.
- We next propose the notion of approximate distributions in probabilistic data used for ranking (Section 5.5) to alleviate computation cost from distributed sites. We present a sampling algorithm that optimally minimizes the total error in the approximation at each site. By transmitting them to the server at the beginning of the algorithm, distributed sites are freed from any further expensive computation. These approximate distributions are used by the server to compute the terminating condition conservatively, so that we can still guarantee that the final top-k results returned by the server are exact and correct. Furthermore, these approximations can be incrementally updated by the server after seeing a tuple from the corresponding distributed site, improving their approximation quality as the algorithm progresses.
- We also extend our algorithms to deal with issues on latency, continuous distributions, scoring function, and multiple attributes (Section 5.6).
- We present a comprehensive experimental study that confirms the effectiveness of our approach (Section 5.7).

Finally, we survey the related works (Section 5.8) before concluding the chapter.

5.2 Problem Formulation

Many models for describing uncertain data have been proposed in the literature, as we discuss in Chapter 4. We focus on the attribute-level model of uncertainty that has been used frequently within the database community. In this section, we will only introduce the basics as well as our assumptions regarding an attribute-level uncertain database in both the centralized and distributed settings; for a more in-depth discussion on the attribute-level uncertainty model and its use within the community, please refer to Section 4.3. Without loss of generality, we assume a probabilistic database $\mathcal{D}$ contains only one attribute-level uncertain relation (also commonly referred to as a *relational table*, or *table* for short).

5.2.1 Uncertainty data model

The probabilistic database $\mathcal{D}$ is a table of N tuples. Each tuple has one attribute whose value is uncertain (together with other certain attributes). This uncertain attribute has a discrete pdf describing its value distribution. When instantiating this uncertain relation to a certain instance, each tuple draws a value for its uncertain attribute based on the associated pdf and the choice is independent among tuples. For ranking queries, the important case is when the uncertain attribute represents the score for the tuple, and we would like to rank the tuples based on this score attribute. Let X_i be the random variable denoting the score of tuple t_i. We assume that X_i has a discrete pdf with bounded size (bounded by b_i). The general, continuous pdf case is discussed in Section 5.6 as well as in our experimental study. In this model, we are essentially ranking the set of independent random variables $X_1, \ldots, X_N$. In the sequel, we will not distinguish between a tuple t_i and its corresponding random variable X_i. This model is illustrated in Table 5.1. For tuple t_i, the score takes the value $v_{i,j}$ with probability $p_{i,j}$ for $1 \leq j \leq b_i$, and for $\forall i$, $b_i \leq b$, where b is an upper bound on the size of any pdf.

5.2.2 The possible world semantics

In the above uncertainty model, an uncertain relation $\mathcal{D}$ is instantiated into a *possible world* by taking one value for each tuple's uncertain attribute independently according to its distribution. Denote a possible world as W and the value for t_i's uncertain attribute in W as w_{t_i}. The probability that W occurs is $\Pr[W] = \prod_{j=1}^{N} p_{j,x}$, where x satisfies $v_{j,x} = w_{t_j}$. It is worth mentioning that in this case, we always have $\forall W \in \mathcal{W}, |W| = N$, where $\mathcal{W}$ is the space of all the possible worlds. The example in Table 5.2 illustrates the possible worlds for an uncertain relation in this model.

5.2.3 The ranking definition

As we have argued in Chapter 4, many definitions for ranking queries in probabilistic data exist. Among them, the expected rank approach is particularly useful as it is an

Table 5.1. The uncertainty data model.

tuples	score
t_1	$\{(v_{1,1}, p_{1,1}), (v_{1,2}, p_{1,2}), \ldots, (v_{1,b_1}, p_{1,b_1})\}$
t_2	$\{(v_{2,1}, p_{2,1}), \ldots, v_{2,b_2}, p_{2,b_2})\}$
$\vdots$	$\vdots$
t_N	$\{(v_{N,1}, p_{N,1}), \ldots, (v_{N,b_N}, p_{N,b_N})\}$

Table 5.2. An example of possible worlds.

tuples	score
t_1	$\{(120, 0.8), (62, 0.2)\}$
t_2	$\{(103, 0.7), (70, 0.3)\}$
t_3	$\{(98, 1)\}$

world W	$\Pr[W]$
$\{t_1 = 120, t_2 = 103, t_3 = 98\}$	$0.8 \times 0.7 \times 1 = 0.56$
$\{t_1 = 120, t_3 = 98, t_2 = 70\}$	$0.8 \times 0.3 \times 1 = 0.24$
$\{t_2 = 103, t_3 = 98, t_1 = 62\}$	$0.2 \times 0.7 \times 1 = 0.14$
$\{t_3 = 98, t_2 = 70, t_1 = 62\}$	$0.2 \times 0.3 \times 1 = 0.06$

important statistical value that satisfies a number of intuitive properties. We extend our study of the expected ranks operator to the distributed and parallel setting in this chapter.

For ease of exposition, we reintroduce the formal definition of the expected ranks operator (as originally defined in Chapter 4):

Definition 5.1 (Expected Rank). *The rank of a tuple t_i in a possible world W is defined to be the number of tuples whose score is higher than t_i (so the top tuple has rank 0), i.e.,*

$$\text{rank}_W(t_i) = |\{t_j \in W | w_{t_j} > w_{t_i}\}|.$$

The expected rank $r(t_i)$ is then defined as:

$$r(t_i) = \sum_{W \in \mathcal{W}} \Pr[W] \cdot \text{rank}_W(t_i) \tag{5.1}$$

For the example in Table 5.2, the expected rank for t_1 is $r(t_1) = 0.56 \times 0 + 0.24 \times 0 + 0.14 \times 2 + 0.06 \times 2 = 0.4$. Similarly $r(t_2) = 1.1$, $r(t_3) = 1.5$. So the final ranking is (t_1, t_2, t_3).

5.2.4 Distributed top-k in uncertain data

Given m distributed sites $S = \{s_1, \ldots, s_m\}$, each holding an uncertain database $\mathcal{D}_i$ with size n_i, and a centralized server H, we denote the tuples in $\mathcal{D}_i$ as $\{t_{i,1}, \ldots, t_{i,n_i}\}$ and their corresponding score values as random variables $\{X_{i,1}, \ldots, X_{i,n_i}\}$. Extending the notation from Table 5.1, $X_{i,j}$'s pdf is $\{(v_{i,j,1}, p_{i,j,1}), \ldots, (v_{i,j,b_{ij}}, p_{i,j,b_{ij}})\}$. We would like to report at H the top-k tuples with the lowest $r(t_{i,j})$s as in Definition 5.1 among all tuples in the unified uncertain database $\mathcal{D} = \mathcal{D}_1 \bigcup \mathcal{D}_2 \cdots \bigcup \mathcal{D}_m$ of size $N = \sum_{i=1}^{m} n_i$. The main objective is to minimize the total communication cost in computing the top-k list, which is the same for many problems on distributed data [27, 122].

5.2.5 The straightforward solution

Obviously, one can always ask all sites to forward their databases to H and solve the problem at H locally. If we have a centralized uncertain database $\mathcal{D} = \{t_1, \ldots, t_N\}$, we can find the top-$k$ tuples using our proposed centralized algorithm *A-ERrank* from Section 4.5.1, which we will briefly review here as a warm-up before introducing our more efficient distributed algorithms. We have observed previously, in Section 4.5.1, that $r(t_i)$ can be written as:

$$r(t_i) = \sum_{\ell=1}^{b_i} p_{i,\ell}\big(q(v_{i,\ell}) - \Pr[X_i > v_{i,\ell}]\big), \tag{5.2}$$

where $q(v) = \sum_j \Pr[X_j > v]$. Let U be the universe of all possible values of X_i, $i = 1, \ldots, N$. We have $|U| \leq |bN|$. When b is a constant, we have $|U| = O(N)$. Let $\Lambda(v) = \sum_{v_{i,j}=v} p_{i,j}$ for $\forall i, j$, then $q(v) = \sum_{v' \in U \wedge v' > v} \Lambda(v')$. Then, *A-ERrank* first precomputes $q(v)$ for all $v \in U$ with a linear pass over the input after sorting U (summing up $\Lambda(v')$'s for $v' > v$) which can be done in $O(N \log N)$. Following (5.2), exact computation of the expected rank for a single tuple can be done in constant time given $q(v)$ for all $v \in U$. *A-ERrank* computes the expected ranks for all tuples to find the top-k and has an overall cost of $O(N \log N)$ (retrieving the top-k has an inferior cost $O(N \log k)$ by maintaining a priority queue of size k).

This approach, however, is communication-expensive. In this case, the total communication cost is $|\mathcal{D}| = \sum_{i=1}^{m} |\mathcal{D}_i|$. This will be the baseline we compare against.

5.3 Sorted Access on Local Rank

One common strategy in distributed query processing is to first answer the query within each site individually, and then combine the results together. For our problem, this corresponds to first compute the local ranks of the tuples at the sites to which they belong. In this section we present an algorithm following this strategy.

Consider an uncertain database $\mathcal{D}_i$ in a local site s_i and any $t_{i,j} \in \mathcal{D}_i$. We define $r(t_{i,j}, \mathcal{D}_i)$ as the *local rank* of $t_{i,j}$ in $\mathcal{D}_i$:

$$r(t_{ij}, \mathcal{D}_i) = \sum_{W \in \mathcal{W}(\mathcal{D}_i)} \Pr[W] \cdot \text{rank}_W(t_{ij}), \tag{5.3}$$

where $\mathcal{W}(\mathcal{D}_i)$ is the space of possible worlds of $\mathcal{D}_i$.

Following the algorithm *A-ERrank*, let the universe of values at the site s_i be U_i. We first compute $q_i(v) = \sum_j \Pr[X_{ij} > v]$ for all $v \in U_i$ in $O(n_i \log n_i)$ time. Then, we can efficiently compute the local ranks of all tuples in $\mathcal{D}_i$ using (5.2), i.e.,

$$r(t_{ij}, \mathcal{D}_i) = \sum_{\ell=1}^{b_{ij}} p_{i,j,\ell}\big(q_i(v_{i,j,\ell}) - \Pr[X_{ij} > v_{i,j,\ell}]\big). \tag{5.4}$$

We also extend the local rank definition of $t_{i,j}$ to a $\mathcal{D}_y$ where $y \neq i$. Since $t_{i,j} \notin \mathcal{D}_y$, we define its local rank in $\mathcal{D}_y$ as its rank in $\{t_{ij}\} \bigcup \mathcal{D}_y$. We can calculate $r(t_{ij}, \mathcal{D}_y)$ as

$$\begin{aligned} r(t_{ij}, \mathcal{D}_y) &= \sum_{Y \in \mathcal{D}_y} \Pr[Y > X_{ij}] \\ &= \sum_{Y \in \mathcal{D}_y} \sum_{\ell=1}^{b_{ij}} p_{i,j,\ell} \Pr[Y > v_{i,j,\ell}] \\ &= \sum_{\ell=1}^{b_{ij}} p_{i,j,\ell} \left(\sum_{Y \in \mathcal{D}_y} \Pr[Y > v_{i,j,\ell}] \right) \\ &= \sum_{\ell=1}^{b_{ij}} p_{i,j,\ell} q_y(v_{i,j,\ell}). \end{aligned} \tag{5.5}$$

Note that in the last step of the derivation above, we use the fact that $t_{ij} \notin \mathcal{D}_y$; hence, X_{ij} does not contribute to U_y and $q_y(v)$.

An important observation on the (global) expected rank of any tuple t_{ij} is that its expected rank could be calculated cumulatively from all the sites. More precisely, we have the following.

Lemma 5.1. *The (global) expected rank of t_{ij} is*

$$r(t_{ij}) = \sum_{y=1}^{m} r(t_{ij}, \mathcal{D}_y),$$

where $r(t_{i,j}, \mathcal{D}_y)$ is computed using (5.4) if $y = i$, or (5.5) otherwise.

Proof of Lemma 5.1: First, since $\mathcal{D} = \mathcal{D}_1 \bigcup \mathcal{D}_2 \cdots \bigcup \mathcal{D}_m$, we have $U = U_1 \bigcup U_2 \cdots U_m$. Furthermore, $q(v) = \sum_{i=1}^m q_i(v)$ by definition. Then, we have

$$\begin{aligned} \sum_{y=1}^{m} r(t_{ij}, \mathcal{D}_y) &= \sum_{y=1, y\neq i}^{m} \sum_{\ell=1}^{b_{ij}} p_{i,j,\ell} q_y(v_{i,j,\ell}) \\ &\quad + \sum_{\ell=1}^{b_{ij}} p_{i,j,\ell}\big(q_i(v_{i,j,\ell}) - \Pr[X_{ij} > v_{i,j,\ell}]\big) \end{aligned}$$

$$\begin{aligned}
&= \sum_{\ell=1}^{b_{ij}} p_{i,j,\ell}\big(\sum_{y=1}^{m} q_y(v_{i,j,\ell}) - \Pr[X_{ij} > v_{i,j,\ell}]\big) \\
&= \sum_{\ell=1}^{b_{ij}} p_{i,j,\ell}\big(q(v_{i,j,\ell}) - \Pr[X_{ij} > v_{i,j,\ell}]\big) \\
&= r(t_{ij}). \qquad \text{(By equation (5.2))}.
\end{aligned}$$

■

An immediate corollary of Lemma 5.1 is that any tuple's local rank is a lower bound on its global rank. Formally,

Corollary 5.1. *For any tuple* t_{ij}, $r(t_{ij}) \geq r(t_{ij}, \mathcal{D}_i)$.

Lemma 5.1 indicates that by forwarding only the tuple t_{ij} itself from the site s_i to all other sites, we can obtain its final global rank. This naturally leads to the following idea for computing the global top-k at the central server H. We sort the tuples at site s_i based on their local ranks $r(t_{ij}, \mathcal{D}_i)$. Without loss of generality, we assume $r(t_{i1}, \mathcal{D}_i) \leq r(t_{i2}, \mathcal{D}_i) \ldots \leq r(t_{in_i}, \mathcal{D}_i)$ for any site s_i. The central server H accesses tuples from the m sites in the increasing order of their local ranks. More precisely, H maintains a priority queue L of size m in which each site s_i has a representative local rank value and the tuple id that corresponds to that local rank value, i.e., a triple $\langle i, j, r(t_{i,j}, \mathcal{D}_i)\rangle$. The triples in the priority queue are sorted by the local rank value in ascending order. L is initialized by retrieving the first tuple's id and local rank from each site.

In each step, H obtains the first element from L, say $\langle i, j, r(t_{i,j}, \mathcal{D}_i)\rangle$. Then, H asks for tuple t_{ij} from site s_i as well as $r(t_{i,j+1}, \mathcal{D}_i)$, the local rank of the next tuple from s_i. The triple $\langle i, j+1, r(t_{i,j+1}, \mathcal{D}_i)\rangle$ will be inserted into the priority queue L. In order to compute the exact global rank of $t_{i,j}$ that H has just retrieved, H broadcasts $t_{i,j}$ to all sites except s_i and asks each site $\mathcal{D}_y$ to report back the value $r(t_{ij}, \mathcal{D}_y)$ (based on equation (5.5)). By Lemma 5.1, H obtains the exact global rank of tuple t_{ij}. This completes a round.

Let the set of tuples seen by H be $\mathcal{D}_H$. H dynamically maintains a priority queue for tuples in $\mathcal{D}_H$ based on their global ranks. In the λth round, let the kth smallest rank from $\mathcal{D}_H$ be r^+_λ. Clearly, the local rank value of any unseen tuples by H from all sites is lower bounded by the head element from L. This in turn lower bounds the global rank value of any unseen tuples in $\mathcal{D} - \mathcal{D}_H$ by Corollary 5.1. Let r^-_λ be the local rank of the head element of L. It is safe for H to terminate the search as soon as $r^+_\lambda \leq r^-_\lambda$ at some round λ and output the top-k from the current $\mathcal{D}_H$ as the final result. We denote this algorithm as *A-LR*.

5.4 Sorted Access on Expected Score

Sorted access on local rank has limited pruning power as it simply relies on the next tuple's local rank from each site to lower bound the global rank of any unseen tuple. This is too pessimistic an estimate. This section introduces the framework of sorted access on expected score, which incurs much less communication cost than the basic local rank approach.

5.4.1 The general algorithm

The general algorithm in this framework is for H to access tuples from all the sites in the descending order of their expected scores. Specifically, each site sorts its tuples in the decreasing order of the expected score, i.e., for all $1 \leq i \leq m$ and $1 \leq j_1, j_2 \leq n_i$, if $j_1 < j_2$, then $\mathbf{E}[X_{ij_1}] \geq \mathbf{E}[X_{ij_2}]$. H maintains a priority queue L of triples $\langle i, j, E[X_{ij}]\rangle$, where the entries are sorted in the descending order of the expected scores. L is initialized by retrieving the first tuple's expected score from each of the m sites. In each round, H pops the head element from L, say $\langle i, j, E(X_{ij})\rangle$, and requests the tuple t_{ij} (and its local rank value $r(t_{ij}, \mathcal{D}_i)$) from site s_i, as well as the expected score of the next tuple at s_i, i.e., $E[X_{i,j+1}]$. Next, H inserts the triple $\langle i, j+1, E[X_{i,j+1}]\rangle$ into L. Let τ be the expected score of the top element from L after this operation. Clearly, τ is an upper bound on the expected score for any unseen tuple.

Similarly to the algorithm *A-LR*, H broadcasts t_{ij} to all sites (except s_i) to get its local ranks and derive the global rank for t_{ij}. H also maintains the priority queue for all tuples in $\mathcal{D}_H$ (the seen tuples by H) based on their global ranks and r^+_λ is similarly defined as in *A-LR* for any round λ. The key issue now is to derive a lower bound r^-_λ for the global rank of any unseen tuple t from $\mathcal{D} - \mathcal{D}_H$. H has the knowledge that $\forall t$ with a random variable X for its score attribute, $E(X) \leq \tau$. We will show two methods in the sequel to derive r^-_λ based on τ. Once r^-_λ is calculated, the round λ completes. Then H either continues to the next round or terminates if $r^+_\lambda \leq r^-_\lambda$.

5.4.2 Markov inequality-based approach

Given the expectation of a random variable X, the Markov inequality could bound the probability that the value of X is above a certain value. Since τ is an upper bound for the expected score of any unseen tuple t with the score attribute X, we have $E(X) \leq \tau$ and for a site s_i:

$$\begin{aligned}
r(t, \mathcal{D}_i) &= \sum_{j=1}^{n_i} \Pr[X_j > X] = n_i - \sum_{j=1}^{n_i} \Pr[X \geq X_j] \\
&= n_i - \sum_{j=1}^{n_i} \sum_{\ell=1}^{b_{ij}} p_{i,j,\ell} \Pr[X > v_{i,j,\ell}] \\
&\geq n_i - \sum_{j=1}^{n_i} \sum_{\ell=1}^{b_{ij}} p_{i,j,\ell} \frac{\mathbf{E}[X]}{v_{i,j,\ell}}. \qquad \text{(Markov Ineq.)} \\
&\geq n_i - \sum_{j=1}^{n_i} \sum_{\ell=1}^{b_{ij}} p_{i,j,\ell} \frac{\tau}{v_{i,j,\ell}} = r^-(t, \mathcal{D}_i). \qquad (5.6)
\end{aligned}$$

This leads to the next lemma that lower bounds the global rank of any unseen tuple.

Lemma 5.2. *Let τ be the expected score for the head element from L at round λ. Then, for any unseen tuple t:*

$$r(t) \geq \sum_{i=1}^{m} r^-(t, \mathcal{D}_i) = r^-_\lambda,$$

for $r^-(t, \mathcal{D}_i)$ defined in equation (5.6).

Proof of Lemma 5.2: By equation (5.6) and the Lemma 5.1. ■

By Lemma 5.2 and the general algorithm in Section 5.4.1, our algorithm could terminate as soon as $r^+_\lambda \leq r^-_\lambda$ at some round λ. This is denoted as the algorithm *A-Markov*. Since both n_i and $sm_i = \sum_{i=1}^{n_i} \sum_{\ell=1}^{b_{ij}} \frac{p_{i,j,\ell}}{v_{i,j,\ell}}$ are invariants for different rounds λ's in equation (5.6), a notable improvement to *A-Markov* is to have each site s_i transmit its n_i and sm_i to H at the beginning of the algorithm, once. Then, at each round λ, $r^-(t, \mathcal{D}_i)$ could be computed locally at H. Note that in order to compute the exact rank of seen tuples and derive r^+_λ as well as producing the final output, the server still needs to broadcast each new incoming tuple to all sites and collect its local ranks.

5.4.3 Optimization with linear programming

The Markov inequality in general gives a rather loose bound. In this section, we give a much more accurate lower bound on $r(t, \mathcal{D}_i)$ for any tuple $t \notin \mathcal{D}_H$. Again, let X be the uncertain score of t, and we have $E[X] \leq \tau$. Our general idea is to let H send τ to all sites in each round and ask each site to compute a lower bound *locally* on the rank of any unseen tuples (from H's perspective), i.e., a lower bound on $r(t, \mathcal{D}_i)$ for all $\mathcal{D}_i's$. All sites then send back these lower bounds and H will utilize them to compute the global lower bound on the rank of any unseen tuple, i.e., r^-_λ.

The computation for $r(X, \mathcal{D}_i)$ is different depending on whether $X \in \mathcal{D}_i$ or $X \notin \mathcal{D}_i$. We first describe how to lower bound $r(X, \mathcal{D}_i)$ if $X \notin \mathcal{D}_i$. The problem essentially is, subject to the constraint $E[X] \leq \tau$, how to construct the pdf of X such that $r(X, \mathcal{D}_i)$ is minimized. The minimum possible $r(X, \mathcal{D}_i)$ is obviously a lower bound on $r(X, \mathcal{D}_i)$. Let U_i be the universe of possible values taken by tuples in $\mathcal{D}_i$. Suppose the pdf of X is $\Pr[X = v_\ell] = p_\ell, v_1 < v_2 < \cdots < v_\gamma$ for some γ. Let $q_i(v) = \sum_{Y \in \mathcal{D}_i} \Pr[Y > v]$; note that we always have $q_i(-\infty) = \sum_{v \in U_i} \Lambda(v)$ where $\Lambda(v) = \sum_{Y \in \mathcal{D}_i \wedge Y.v_j = v} Y.p_j$, and $q_i(v_L) = 0$ where v_L is the largest value in U_i. Since $X \notin \mathcal{D}_i$, by equation (5.5), the rank of X in $\mathcal{D}_i$ is

$$r(X, \mathcal{D}_i) \quad = \sum_{Y \in \mathcal{D}_i} \Pr[Y > X] = \sum_{\ell=1}^{\gamma} p_\ell q_i(v_\ell). \tag{5.7}$$

Note that $q_i(v)$ is a nonincreasing, staircase function with changes at the values of U_i. (We also include $-\infty$ in U_i.) We claim that to minimize $r(X, \mathcal{D}_i)$, we only need to consider values in U_i to form the v_ℓs, the values used in the pdf of X. Suppose the pdf uses some $v_\ell \notin U_i$. Then, we decrease v_ℓ until it hits some value in U_i. During this process, $E[X]$ decreases so the constraint $E[X] \leq \tau$ is still satisfied. As we decrease v_ℓ while not passing a value in U_i, $q_i(v_\ell)$ does not change (see the example in Figure 5.1). So (5.7) stays unchanged during this transformation of the pdf. Note that this transformation will always reduce the number of v_ℓs that are not in U_i by one. Applying this transformation repeatedly will thus arrive at some pdf of X with all $v_\ell \in U_i$ without changing $r(X, \mathcal{D}_i)$.

Therefore, we can assume without loss of generality that the pdf of X has the form $\Pr[X = v_\ell] = p_\ell$ for each $v_\ell \in U_i$, where

$$0 \leq p_\ell \leq 1, \qquad \ell = 1, \ldots, \gamma = |U_i|; \tag{5.8}$$

$$p_1 + \cdots + p_\gamma = 1. \tag{5.9}$$

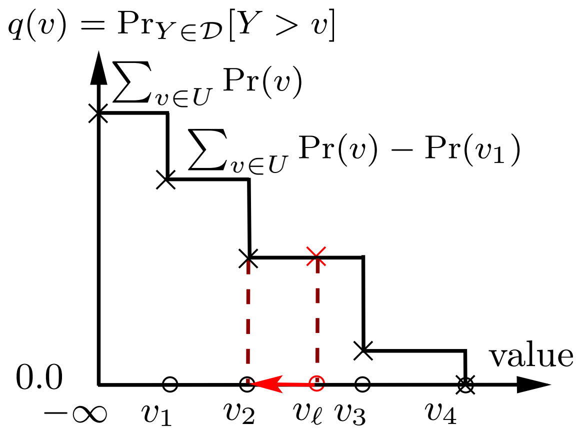

$q(v) = \Pr_{Y \in \mathcal{D}}[Y > v]$
$\sum_{v \in U} \Pr(v)$
$\sum_{v \in U} \Pr(v) - \Pr(v_1)$
value
0.0
$-\infty$
v_1
v_2
v_ℓ
v_3
v_4

Figure 5.1. Transform values in an unseen tuple X.

The constraint $E[X] \leq \tau$ becomes

$$E[X] = p_1 v_1 + \cdots + p_\gamma v_\gamma \leq \tau. \tag{5.10}$$

Therefore, the problem is to minimize (5.7) subject to the linear constraints (5.8) (5.9) (5.10), which can be solved using linear programming.

Next, consider the case $X \in \mathcal{D}_j$ for some j. Then, $r(X, \mathcal{D}_j)$ can be computed as in (5.4), i.e.,

$$\begin{aligned} r(X, \mathcal{D}_j) &= \sum_{\ell=1}^{\gamma} p_\ell (q_j(v_\ell) - \Pr[X > v_\ell]) \\ &= \sum_{\ell=1}^{\gamma} p_\ell q_j(v_\ell) - \sum_{\ell=1}^{\gamma} p_\ell \Pr[X > v_\ell] \\ &\geq \sum_{\ell=1}^{\gamma} p_\ell q_j(v_\ell) - \sum_{\ell=1}^{\gamma} p_\ell = \sum_{\ell=1}^{\gamma} p_\ell q_j(v_\ell) - 1, \end{aligned}$$

where $q_j(v_\ell) = \sum_{Y \in \mathcal{D}_j} \Pr[Y > v_\ell]$. Therefore, we can still minimize as we do for any other $\mathcal{D}_i$, but simply subtract one from the final lower bound on $r(X)$ that we obtain after aggregating the minimum of (5.7) from all $\mathcal{D}_i$s. These observations are summarized in the next lemma.

Lemma 5.3. *Let X be an arbitrary unseen tuple by H. For $\forall i \in \{1, \ldots, m\}$, suppose $r^-(X, \mathcal{D}_i)$ is the optimal minimum value from the linear program using (5.7) as the objective function and (5.8), (5.9), (5.10) as the constraints for each site s_i respectively. Then,*

$$r(X) \geq \sum_{i=1}^{m} r^-(X, \mathcal{D}_i) - 1 = r^-(X).$$

Proof of Lemma 5.3: By Lemma 5.1 and the optimal minimum local rank returned by each linear programming formulation. ■

This naturally leads to an optimization to the sorted by expected score framework. H maintains the current kth tuple's rank among all the seen tuples at round λ as r^+_λ and $r^-(X)$ at round λ as r^-_λ. As soon as $r^+_\lambda \leq r^-_\lambda$, H stops the search and outputs the current top-k from $\mathcal{D}_H$. This is the *A-LP* algorithm.

5.5 Approximate $q(v)$: Reducing Computation at Distributed Sites

In many distributed applications (e.g., sensors), the distributed sites often have limited computation power or cannot afford expensive computation due to energy concerns. Algorithm *A-LP* finds the optimal lower bound for the local rank at each site, but at the expense of solving a linear program each round at all sites. This is prohibitive for some applications. This section presents a method to approximate the $q(v)$, which enables the site to shift almost all the computation costs to the server H while still keeping the communication cost low.

5.5.1 $q^*(v)$: an approximate $q(v)$

Given a database $\mathcal{D}$ and its value universe U, $q(v)$ represents the aggregated cumulative distribution $\Pr_{X\in\mathcal{D}}[X > v]$, which is a staircase curve (see Figure 5.1). Let $U = \{v_0, v_1, \dots, v_\gamma\}$ where $v_0 = -\infty$ and $\gamma = |U|$. Then, $q(v_i) = \sum_{j>i} \Lambda(v_j)$ where $\Lambda(v) = \sum_{X\in\mathcal{D}\wedge X.v_i=v} X.p_i$. $q(v)$ is decided by a set of points $\{(v_0, q(v_0)), (v_1, q(v_1)), \dots, (v_\gamma, 0)\}$, but it is well-defined for any value v even if it is not in U, i.e., for $v \notin U$, $q(v) = \sum_{v_j>v\wedge v_j\in U} \Lambda(v_j)$.

In the *A-LP* approach, the computation of $r^-(X, \mathcal{D}_i)$ only depends on $q_i(v)$. If each site s_i sends its $q_i(v)$ to H at the beginning of the algorithm, then the server could compute $r^-(X, \mathcal{D}_i)$ locally at each round without invoking the linear programming computation at each site in every round. However, $q_i(v)$ is expensive to communicate if $|U_i|$ is large. In the worst case when tuples in $\mathcal{D}_i$ all take distinct values, $|q_i(v)| = |\mathcal{D}_i|$ and this approach degrades to the straightforward solution of forwarding the entire $\mathcal{D}_i$ to H.

This motivates us to consider finding an approximate $q^*(v)$ for a given $q(v)$, such that $|q^*(v)|$ is small and adjustable, and provides a good approximation to $q(v)$. The approximation error ε is naturally defined to be the area enclosed by the approximate and the original curves, i.e.,

$$\varepsilon = \int_{v\in[-\infty,+\infty]} |q(v) - q^*(v)| dv. \tag{5.11}$$

We use such a definition of error because the site does not know beforehand how $q^*(v)$ is going to be used by the server. If we assume that each point on $q(v)$ is equally likely to be probed, then the error ε defined in (5.11) is exactly the expected error we will encounter.

The approximation $q^*(v)$ is naturally a staircase curve as well. Thus, the problem is, given a tunable parameter $\eta \le |U|$, how to obtain a $q^*(v)$ represented by η points while minimizing ε.

However, not all approximations meet the problem constraint. We need to carefully construct $q^*(v)$ so that given an upper bound value τ on the expected score, the solution from the linear program w.r.t $q^*(v)$ is still a lower bound for $r(X, \mathcal{D})$ for any unknown tuple X with $E(X) \le \tau$. In other words, let $r^*(X, \mathcal{D})$ be the optimal value identified by the linear program formulated with $q^*(v)$, and $r^-(X, \mathcal{D})$ be the optimal value from the linear program using $q(v)$ directly. We must make sure that $r^*(X, \mathcal{D}) \le r^-(X, \mathcal{D})$ so that the lower bounding framework still works, and the returned top-k results are still guaranteed to be exact and correct. Intuitively, we need $q^*(v)$ below $q(v)$ in order to have this guarantee. In what follows, we first present an algorithm that finds the optimal $q^*(v)$ below $q(v)$ that minimizes the error ε; then, we show that such a $q^*(v)$ indeed gives the desired guarantee.

There are still many possible choices to construct a $q^*(v)$ as there are infinite number of decreasing staircase curves that are always below $q(v)$ and are decided by η turning points. The next question is, among the many possible choices, which option is the best in minimizing the error ε? The insight is summarized by the next Lemma.

Lemma 5.4. *Given any $\eta \le |U|$ and $q(v)$, $q^*(v)$, s.t., $q^*(v) \le q(v)$ for $\forall v \in [-\infty, +\infty]$ and $|q^*(v)| = \eta$, the approximation error ε is minimized iff $q^*(v)$'s right-upper corner points only sample points from the set of right-upper corner points in the staircase curve of $q(v)$, i.e., $q^*(v)$ is determined by a subset of $\Delta_{q(v)} = \{\alpha_1 : (v_1, q(v_0)), \ldots, \alpha_\gamma : (v_\gamma, q(v_{\gamma-1})\}$.*

Proof of Lemma 5.4: We concentrate on the necessary condition; the sufficient condition can be argued similarly. We prove this by contradiction. Suppose this is not true; then, we have a $q^*(v)$ with the smallest approximation error that contains a right-upper corner point $\alpha' \notin \Delta_{q(v)}$. Since $q^*(v)$ is always below $q(v)$ for $\forall v \in [-\infty, +\infty]$, moving α' towards the $\alpha_i \in \Delta_{q(v)}$ that is the first to its right will only reduce the area enclosed by $q^*(v)$ and $q(v)$. Please see Figure 5.2(a) for an example where we move α' to α_3 and α'' to α_4. This conflicts with the fact that $q^*(v)$ minimizes the approximation error ε and completes the proof. ■

A Corollary for constructing $q^*(v)$ is that the two boundary points from $\Delta_{q(v)}$ should always be sampled; otherwise, the error ε could be unbounded.

Corollary 5.2. *Both $\alpha_1 : (v_1, q(v_0))$ and $\alpha_\gamma : (v_\gamma, q(v_{\gamma-1}))$ from $\Delta_{q(v)}$ should be included in $q^*(v)$'s right corner points in order to have a finite error ε.*

Lemma 5.4 is illustrated in Figure 5.2(a) where the $\times$s are the points (the lower-left corner points) in a $q(v)$ and the $\triangle$s are the right-upper corner points that should be used to determine $q^*(v)$ in order to minimize the approximation error ε. The dashed line denotes a possible curve for $q^*(v)$ with $\eta = 2$. The two extreme points $(v_1, q(-\infty))$ and $(v_5, q(v_4))$ are automatically included, plus α_3 and α_4. Once we have found the optimal set of α points from $\Delta_{q(v)}$, the $\times$ points in $q^*(v)$ could be easily constructed. As a convention, we do not include the two boundary $\triangle$ points in the budget η.

With Lemma 5.4 and Corollary 5.2, we are ready to present the algorithm that obtains an optimal $q^*(v)$ given a budget η. Note that $q^*(v)$ always have the two boundary $\triangle$ points from $q(v)$. The basic idea is to use dynamic programming.

Let $\Delta^{\sharp}_{q(v)} = \{\alpha_2, \ldots, \alpha_{\gamma-1}\}$. Let $A(i,j)$ be the approximation error corresponding to the optimal $q^*(v)$ with i points selected from the first j points in $\Delta^{\sharp}_{q(v)}$ for all $1 \leq i \leq j \leq \gamma - 2$ (since $\Delta^{\sharp}_{q(v)}$ has $\gamma - 2$ number of points), together with the two boundary Δ points (α_1 and α_γ). The optimal curve achieving $A(i,j)$ is denoted as $q^*(i,j)$. Next, let $\delta^{j+1}_{q^*(i,j)}$ be the area reduced by adding the $(j+1)$-th point from $\Delta^{\sharp}_{q(v)}$, i.e., α_{j+2}, to $q^*(i,j)$. Now, we have:

$$A(i,j) = \min \begin{cases} \min_{x \in [i-1, j-1]} \{A(i-1, x) - \delta^{j}_{q^*(i-1,x)}\}; \\ \min_{x \in [i, j-1]} \{A(i, x)\}. \end{cases} \tag{5.12}$$

Our goal is to find $A(\eta, \gamma - 2)$ and the corresponding $q^*(\eta, \gamma - 2)$ will be $q^*(v)$ with the minimum approximation error to $q(v)$ using only η right-upper corner points (plus α_1 and α_γ).

Given any $q^*(i,j)$ and α_{j+2}, $\delta^{j+1}_{q^*(i,j)}$ could be easily and efficiently computed using only subtraction and multiplication since $q^*(i,j)$ is a staircase curve. Suppose the last $\triangle$ point, except α_γ, in $q^*(i,j)$ is α_x, recall that α_{j+2} is $(v_{j+2}, q(v_{j+1}))$ and α_x is $(v_x, q(v_{x-1}))$, then:

$$\delta^{j+1}_{q^*(i,j)} = (v_{j+2} - v_x) \times (q(v_{j+1}) - q(v_{\gamma-1})). \tag{5.13}$$

The base case is when $i = j = 0$. This simply corresponds to having only the two boundary Δ points in $q^*(0,0)$ and

$$A(0,0) = \sum_{i \in [2, \gamma-1]} (v_i - v_{i-1})(q(v_{i-1}) - q(v_{\gamma-1})).$$

For example, in Figure 5.2(b), $q^*(0,0)$'s right-upper corner points are $(v_1, q(-\infty))$ and $(v_5, q(v_4))$, $A(0,0)$ corresponds to the the gray area in Figure 5.2(b), and $\delta^1_{q^*(0,0)}$ is the area reduced by adding α_2 to $q^*(0,0)$ (marked by the gray dotted lines in Figure 5.2(b)). This gives us a dynamic programming formulation for finding the right-upper corner points in optimal $q^*(v)$ for any η.

This dynamic programming algorithm requires only subtraction, addition, and multiplication, and only needs to be carried out once per distributed site. Hence, even a site with limited computation power is able to carry out this procedure. Compared with the linear programming approach in Section 5.4.3, each site has shifted the expensive linear programming computation to the server.

It still remains to argue that such a $q^*(v)$ computed as above guarantees that $r^*(X, \mathcal{D}) \leq r^-(X, \mathcal{D})$, such that our lower bounding framework is still correct. This is formalized in the following theorem.

Theorem 5.1. *If $q^*(v)$ is constructed only using upper-right corners from the set of points $\Delta_{q(v)}$, then for any unknown tuple X with $E(X) \leq \tau$, $r^*(X, \mathcal{D}) \leq r^-(X, \mathcal{D})$.*

Proof of Theorem 5.1: The unknowns in the linear program of Section 5.4.3 are the p_ℓs for $\ell = 1, \ldots, \gamma$, where $\gamma = |U|$. The v_ℓs in the constraint from equation (5.10) only take values from U (or equivalently, the x-coordinates of the turning points of $q(v)$).

Suppose $q^*(v)$ has η points. Then, the LP constructed from $q^*(v)$ has η unknowns. In the following, we will transform this LP into one also with γ unknowns, with the same constraints as the LP constructed from $q(v)$, while having a smaller objective function.

Denote the set of x-coordinates of the turning points in $q^*(v)$ as U^*. For any value $\bar{v} \in U - U^*$, we have $q^*(\bar{v}) \leq q(\bar{v})$. Now, we add the value $\bar{v}$ to U^* and a corresponding unknown to the LP. Note this transformation does not change the staircase curve defined by $q^*(v)$. We apply such transformations for all values from $U - U^*$. Now we obtain a LP that has the same set of unknowns and the same constraints (5.8)(5.9)(5.10) as the LP generated from $q(v)$. The objective function (5.7) of this transformed LP has smaller or equal coefficients. Thus, the optimal solution to this transformed LP is no larger than that of the original LP constructed from $q(v)$.

Finally, we need to argue that this transformed LP is actually the same as the LP constructed from $q^*(v)$, namely, this transformation is merely conceptual and we do not need to actually do so. Indeed, since all the new unknowns that are added during the transformation are not at the turning points of $q^*(v)$, by the same reasoning of Section 5.4.3, we know that in the optimal solution of this transformed LP, these new unknowns will be zero anyway. Thus, we do not need to actually include these unknowns in the LP, and it suffices to solve the simpler LP that is constructed just from $q^*(v)$. ■

Theorem 5.1 indicates that by using $q_i^*(v)$s, the server is able to find a lower bound for the local rank of any unseen tuple and check the terminating condition by solving the linear programming formulation *locally.* Note that the server still forwards every seen tuple to all sites to get its exact global rank based on the $q_i(v)$s stored at individual sites. This, together with Theorem 5.1, guarantees that the final top-k are exact answers. There is the overhead of communicating $q_i^*(v)$s to H at the beginning of the algorithm. However, it is a one-time cost. Furthermore, in each subsequent round, the communication of passing τ from H to all sites and sending lower bound values from all sites back to the server in the *A-LP* algorithm is saved. This will compensate the cost of sending $q_i^*(v)$s as evident from our experiments.

5.5.2 Updating the $q_i^*(v)$s at the server

Initially, each site computes the approximate $q_i^*(v)$ with a budget η for $q_i(v)$ and sends it to the server H. A nice thing about these approximate $q_i^*(v)$s is that they can be incrementally updated locally by H after seeing tuples from the $\mathcal{D}_i$s so that the approximation quality keeps improving after each update. In our framework, the budget η is only important for the initial transmission of $q_i^*(v)$. After that, the server has no constraint to keep only η number of points in $q_i^*(v)$. As the algorithm progresses and the sites send in their tuples, the server can also use these tuples to improve the quality of $q_i^*(v)$ "for free." Intuitively, when H receives a tuple from some site s_i, H's knowledge about $q_i(v)$ for the database $\mathcal{D}_i$ should be expanded; hence, a better approximation for $q_i(v)$ should be possible.

The general problem is the following. Assume the server H has an initial $q^*(v)$ with a budget η for a database $\mathcal{D}$ that H does not possess. When a tuple $X \in \mathcal{D}$ is forwarded to H, we would like to update $q^*(v)$ with X such that the approximation error ε between $q^*(v)$ and $q(v)$ could be reduced.

Recall that $q^*(v)$ is represented by the set of right-upper corners obtained at the end of the dynamic programming. Suppose H gets a new tuple $X = \{(v_{x_1}, p_{x_1}), \dots, (v_{x_z}, p_{x_z})\}$ for some z. Below we show how to update $q^*(v)$ for one pair $(v_\ell, p_\ell) \in X$; the same procedure is applied to all the pairs one by one.

We call the upper-right corner points in the initial $q^*(v)$ the *initial points*. Note that the two boundary points from $q(v)$ are always initial points. An obvious observation is that all the initial points should not be affected by any update since they are accurate points from the original $q(v)$. Consider an update (v_ℓ, p_ℓ), and the first initial point to its left, denoted Δ^L. Another observation is that this update will not affect the curve $q^*(v)$ outside the interval $[\Delta^L.v, v_\ell]$, where $\Delta^L.v$ is the value of Δ^L. This is because the update (v_ℓ, p_ℓ) will only raise the curve on the left side of v_ℓ, while the initial point Δ^L already incorporates all the information on the right side of Δ^L on the original curve $q(v)$.

We are now ready to describe how to update $q^*(v)$ with (v_ℓ, p_ℓ). By the definition of $q(v)$, the part of the curve of $q^*(v)$ to the left of v_ℓ should be raised by an amount of p_ℓ, if such a contribution has not been accounted for. As observed from above, this will raise $q^*(v)$ from v_ℓ all the way to the left until we hit Δ^L.

An example of this procedure is shown in Figure 5.2(c) with two updates. Suppose the initial points in $q^*(v)$ are α_1, α_4 and α_5. We first update with (v_2, p_2). This will raise the portion $(\Delta^L.v = v_1, v_2)$ by p_2. This corresponds to adding a new upper-right corner point α_2 to $q^*(v)$. Next, we update $q^*(v)$ with (v_ℓ, p_ℓ). As reasoned above, this will raise the

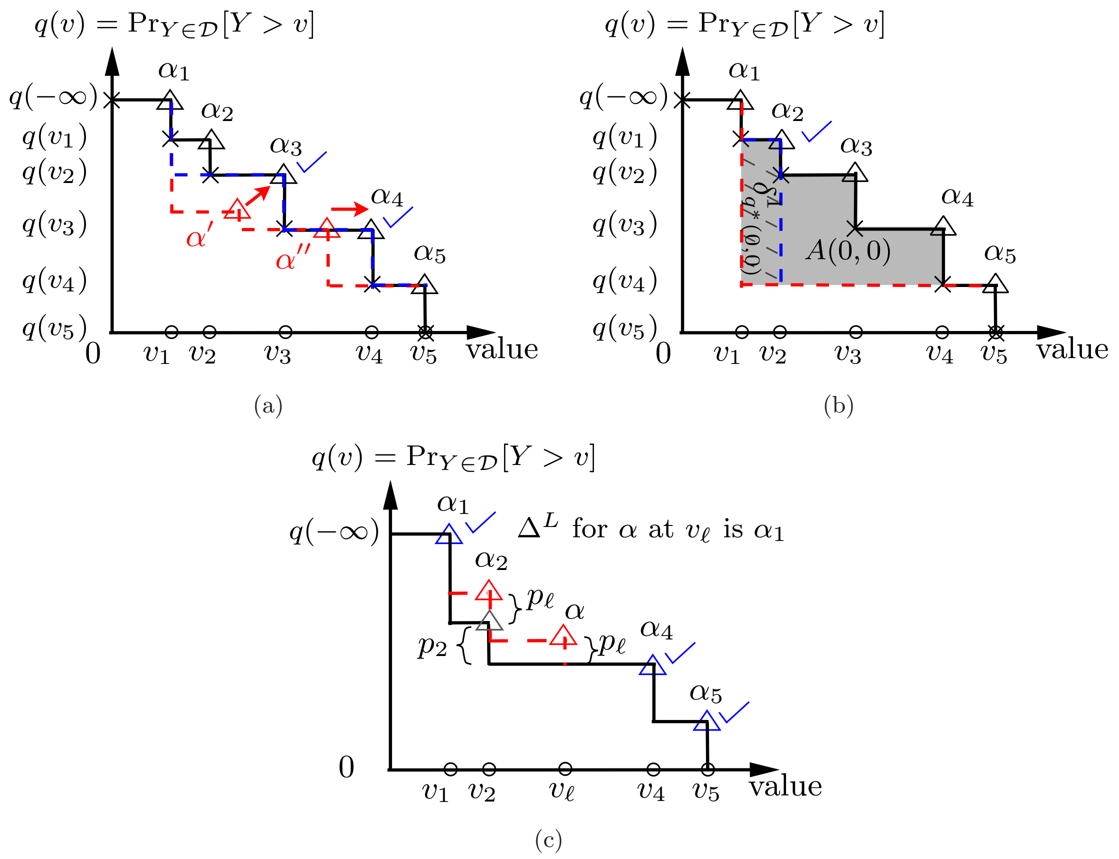


Figure 5.2. $q^*(v)$: definition, optimal computation and update: (a) approximate $q(v)$, (b) base case and $\delta^{j+1}_{q^*(i,j)}$, (c) update $q^*_{\lambda-1}(v)$.

portion ($\Delta^L.v = v_1, v_\ell$) by p_ℓ. To record such a raise, we need to add a new upper-right corner α to $q^*(v)$, and then raise all the $\triangle$ points between $\Delta^L.v$ and v_ℓ by p_ℓ. In this example, we will raise α_2 by p_ℓ.

5.6 Reducing Latency and Other Issues

5.6.1 Reducing latency

All of our algorithms presented so far process one tuple from some site s_j in a single round. The latency of obtaining the final result could be high if there are many rounds. However, there is an easy way to reduce latency. Instead of looking up one tuple at a time, our algorithms could process β tuples before running the lower bounding calculation, for some parameter β. Such a change could be easily adopted by all algorithms. The overall latency will be reduced by a factor of β. However, we may miss the optimal termination point, but by at most β tuples. In Section 5.7, we will further investigate the effects of β empirically.

5.6.2 Continuous distributions

When the input data in the uncertainty model is specified by a continuous distribution (e.g., Gaussian or Poisson), it is often hard to compute the probability that such a random variable exceeds another (e.g., there is no closed formula for Gaussian distributions). However, by discretizing the distributions to an appropriate level of granularity (i.e., represented by a histogram), we can reduce to an instance of the discrete pdf problem. The error in this approach is directly related to the granularity of the discretization.

5.6.3 Scoring function and other attributes

Our analysis has assumed that the score is an attribute. In general, the score can be specified at query time by a user-defined function that could even involve multiple uncertain attributes. Our algorithms also work under this setting, as long as the scores can be computed, by treating the output of the scoring function as an uncertain attribute. Finally, users might be interested in retrieving attributes other than the ranking attribute(s) by the order of the scoring function. We could modify our algorithms to work with trimmed tuples that only contain the necessary attribute(s) for the ranking purpose. When the algorithm has terminated, we retrieve the top-k tuples from distributed sites with user-interested attributes based on the ids of the top-k truncated tuples at server H.

5.7 Experiments

We implemented all the algorithms proposed in this chapter: *A-LR*, *A-Markov*, *A-LP*, *A-BF* (the straightforward solution that sends all $\mathcal{D}_i$s to H and process $\mathcal{D}$ locally using the *A-ERank*), *A-ALP* (the algorithm using approximate $q_i(v)$s). For *A-LP* and *A-ALP*, we used the GNU linear programming kit library (GLPK) [128] to solve LPs.

5.7.1 Datasets

We used three real datasets and one synthetic dataset. The first real dataset is the *movie* dataset from the MystiQ project [129], which contains probabilistic records as a result of information integration of the movie data from IMDB and Amazon. The *movie* dataset contains roughly 56,000 tuples. Each tuple is uniquely identified by the movie id. We rank tuples by the *ASIN* attribute, which varies significantly in different movie records, and may have up to 10 different choices, each associated with a probability.

The second real dataset is the lab readings of 54 sensors from the Intel Research, Berkeley lab [80]. This dataset contains four sets of sensor readings corresponding to the temperature, light intensity, humidity, and voltage of lab spaces over a period of 8 days.

These datasets exhibit similar results in all of our experiments, so we only report the results on the *temperature* dataset. To reflect the fuzzy measurement in sensor readings, we put near-by g sensors (e.g., in the same room) into a group where g is a number between 1 and 10. We treat these g readings as a uniformly distributed discrete pdf of the temperature. The *temperature* dataset has around $67,000$ such records. We rank tuples by their temperature attribute.

The third real dataset is the *chlorine* data from the EPANET project that monitors and models the hydraulic and water quality behavior in water distribution piping systems [130]. This dataset records the amounts of chlorine detected at different locations in the piping system collected over several days. The measurements were inherently fuzzy and usually several monitoring devices were installed at the same location. Hence, we process this dataset in a similar way as the *temperature* dataset. The *chlorine* dataset has approximately $140,000$ records and each record has up to 10 choices on the value of the chlorine amount. We rank tuples by their chlorine amount attribute.

Finally, we also generated the synthetic *Gaussian* dataset where each record's score attribute draws its values from a Gaussian distribution. For each record, the standard deviation σ is randomly selected from $[1, 1000]$ and the mean μ is randomly selected from $[5\sigma, 100000]$. Each record has g choices for its score values where g is randomly selected from 1 to 10. This dataset can be also seen as a way to discretize continuous pdfs.

5.7.2 Setup

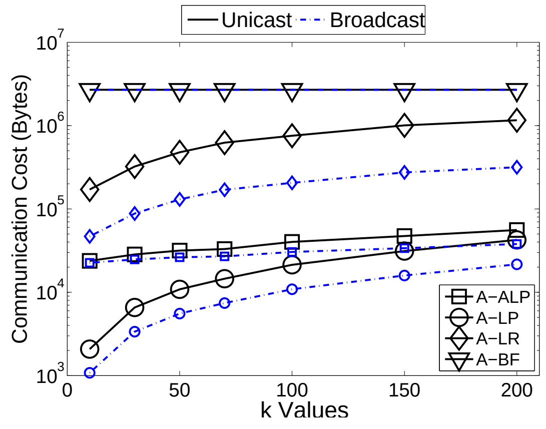
In each experiment, given the number of sites m, each record from the uncertain database $\mathcal{D}$ is assigned to a site s_i chosen uniformly at random. Once the $\mathcal{D}_i$s are formed, we apply all algorithms on the same set of $\mathcal{D}_i$s.

We measure the total communication cost in terms of bytes, as follows. For each choice of the score attribute, both the value and the probability are four bytes. The tuple id is also four bytes. We do not send attributes other than the score attribute and the tuple id. The expected score value is considered to be four bytes as well. We distinguish communication costs under either the broadcast or the unicast scenario. In the broadcast case, whenever the server sends a tuple or an expected score value to all sites, it is counted as one tuple or one value regardless of the number of sites. In the unicast case, such communication is counted as m tuples or m values being transmitted. In either case, all site-to-server communication is unicast.

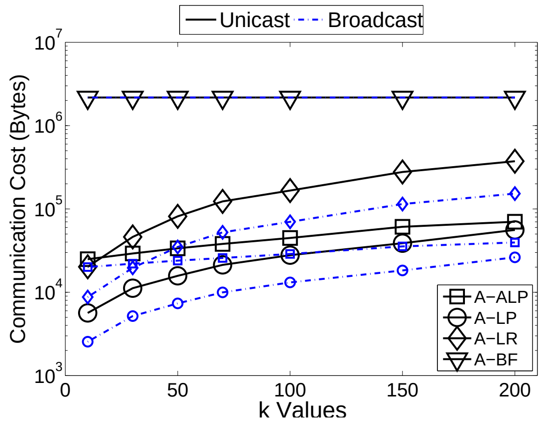
We truncated all datasets to $N = 56,000$ tuples, the size of the *movie* dataset. The default number of sites is $m = 10$ and the default budget η in the *A-ALP* algorithm is set to be 1% of $|q_i(v)|$. The default value of k is 100.

5.7.3 Results with different k

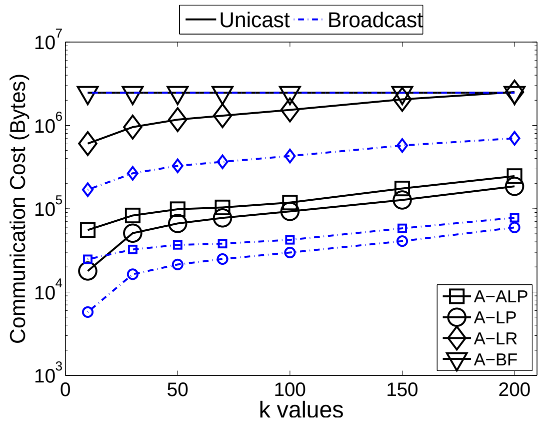
We first study the performance of all the algorithms for different k values from 10 to 200. Figure 5.3 shows the communication costs of the algorithms for the four datasets under both the broadcast and unicast settings; note $N = 56,000$, $m = 10$, and $\eta = 1\% \times |q_i(v)|$ for this experiment. Clearly, *A-LP* and *A-ALP* save the communication cost by at least one to two orders of magnitude compared with *A-BF* in all cases. *A-LR* does provide communication savings over the brute-force approach, but as k increases, it quickly approaches *A-BF*. This indicates that the simple solution of using the local rank alone to characterize the global rank of a tuple is not good enough. *A-Markov* is consistently much worse than the *A-LP* and *A-ALP* algorithms in the access by expected score framework. In some cases, it actually



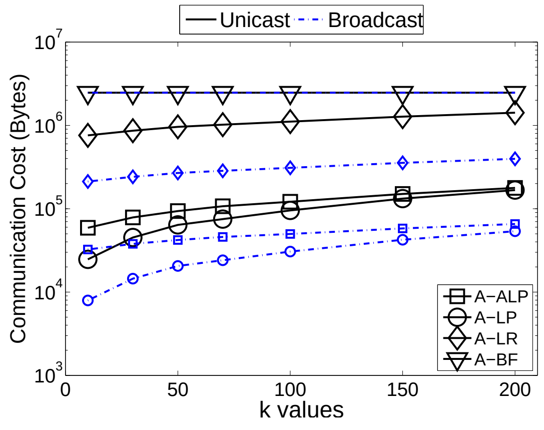
(a)



(b)



(c)



(d)

Figure 5.3. Communication cost while varying k: effect on (a) Synthetic Gaussian, (b) Movie, (c) Temperature, (d) Chlorine.

retrieves almost all tuples from all sites. So we omit *A-Markov* from this and all remaining experiments. All algorithms have increasing communication costs as k gets larger (except *A-BF* of course). The costs of *A-LP* and *A-ALP* gradually increase as k gets larger. A useful consequence of this is that *A-LP* and *A-ALP* are able to return the top tuples very quickly to the user, and then return the remaining tuples progressively. We would like to emphasize that these results were from relatively small databases ($N = 56,000$). In practice, N could be much larger and the savings from *A-LP*, *A-ALP* comparing to *A-BF* could be from several to tens of orders of magnitude .

Another interesting observation is that *A-ALP* achieves similar communication cost as *A-LP*, while with very little computation overhead on the distributed sites. Recall that other than the initial computation of the $q_i^*(v)$'s, the distributed sites have little computation cost during the subsequent rounds in *A-ALP*. A reason is that *A-ALP* does not require the server to send the expected score value τ to all sites and collect the lower bounds on the local ranks based on τ. This results in some communication savings that compensate the needs of communicating $q_i^*(v)$s at the beginning. We also show the number of rounds required in Figure 5.4; again note $N = 56,000$, $m = 10$, $\eta = 1\% \times |q_i(v)|$. Note that for all values of k, *A-LP* and *A-ALP* need only slightly more than k rounds.

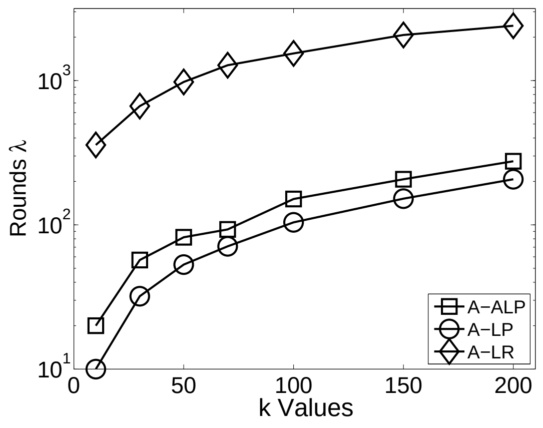
Finally, it is not surprising that our algorithms perform better in the broadcast case. In the rest, we only show the unicast scenario as the other case can only be better.

5.7.4 Results with different N

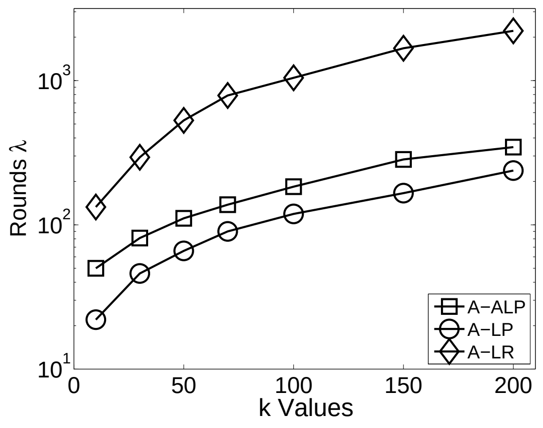
We next study the effects of N, the total number of records in the database $\mathcal{D}$, using the *chlorine* dataset as it is the largest real dataset. Not surprisingly, Figure 5.5 (note $m = 10$, $\eta = 1\%$, $k = 100$) shows that the communication cost of the *A-BF* approach linearly increases with N (note that it is shown in log scale). On the other hand, both the communication cost and the number of rounds for *A-LP* and *A-ALP* increase at a much slower rate. For example, when $k = 100$, both algorithms only access less than 300 tuples (or rounds) even for the largest $N = 140,000$. This means that *A-LP* and *A-ALP* have excellent scalability w.r.t the size of the distributed database while others do not. The gap between *A-LP*, *A-ALP* comparing to *A-BF* quickly increases as N becomes larger.

5.7.5 Results with different m

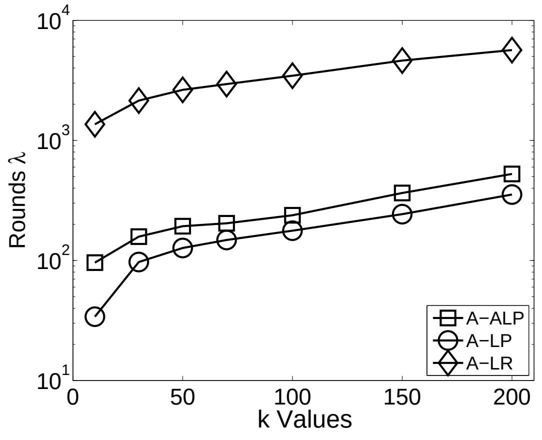
Our next goal is to investigate the effects of m, the number of sites. Figure 5.6 (note $N = 56,000$, $\eta = 1\%$, $k = 100$) shows the experimental results on the *movie* dataset where we varied m from 5 to 30 but kept N, the total database size (the union of all



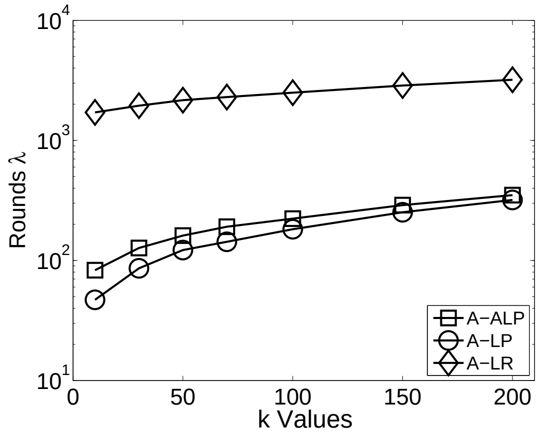
(a)



(b)



(c)



(d)

Figure 5.4. Number of rounds while varying k: effect on (a) Synthetic Gaussian, (b) Movie, (c) Temperature, (d) Chlorine.

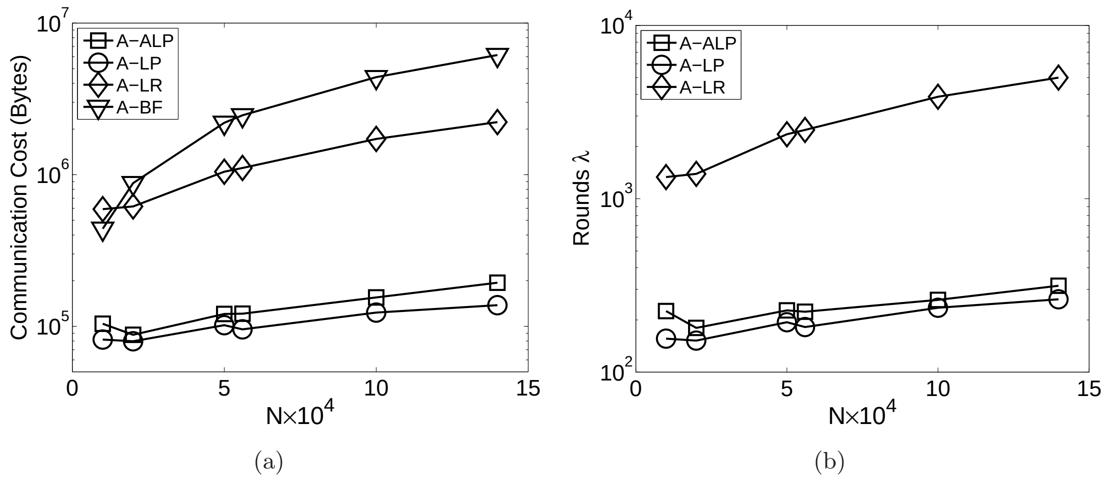


Figure 5.5. Varying N on Chlorine dataset: effect on (a) communication and (b) number of rounds.

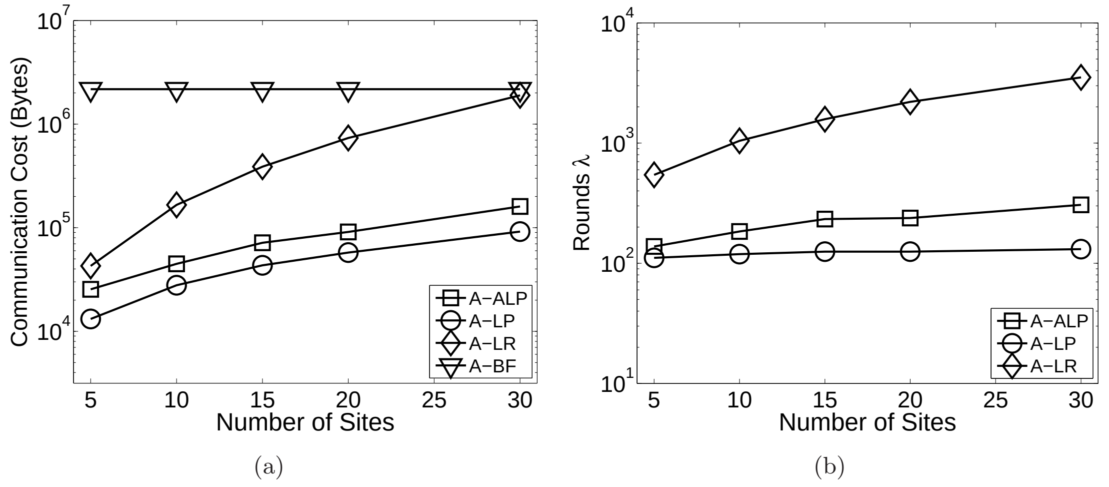


Figure 5.6. Varying m on Movie dataset: effect on (a) communication and (b) number of rounds.

sites) fixed. Since we use unicast, as expected, the communication cost for our algorithms increase as m gets larger. Nevertheless, even with 30 sites, *A-LP* and *A-ALP* are still an order of magnitude better than the basic *A-BF* solution (Figure 5.6(a)). We would like to emphasize that this is the result from the *smallest database* with only $56,000$ tuples and N is kept as a constant. In practice, when m increases, N will increase as well and *A-LP*, *A-ALP* will perform much better than *A-BF*. Finally, the number of sites does not have an obvious impact on the number of rounds required for *A-LP* and *A-ALP* (Figure 5.6(b)), which primarily depends on N and k.

5.7.6 Results with different b

We next study the effects of b, the upper bound on the size of each individual pdf. Recall that for continuous pdfs, we discretize them into discrete pdfs with up to b choices. The larger b is, the better we can approximate the original continuous pdfs. For this purpose, we use the synthetic *Gaussian* dataset in which we can control b. The results are shown in Figure 5.7 (note $N = 56,000$, $m = 10$, $\eta = 1\%$, $k = 100$). As we can see from Figure 5.7(a), the communication costs of all algorithms increase roughly linearly with b. The relative gap among the algorithms basically stay the same. Figure 5.7(b) indicates that the number of rounds λ is almost not affected by b. This is because the dominant factor that determines λ is the expected score value. Changing the number of choices in a pdf does not shift its expected score value too much. Note that with $b = 20$, a continuous pdf and its discrete version is already very close in most cases.

5.7.7 Results with different skewness of the pdfs

We also study the effects of the skewness of the pdfs. Recall that by default for the *temperature* and *chlorine* datasets, the probabilities for a pdf are set uniformly at random to reflect the scenario that the reading is randomly selected in a group of sensors. In practice, we may have a higher priority to collect the reading from one specified senor in a group, resulting in a skewed distribution. For this purpose, for a group of sensors, we always give a probability of ρ to one of them, while dividing the remaining probability equally among the rest of the sensors. Obviously, the larger ρ is, the higher the skewness.

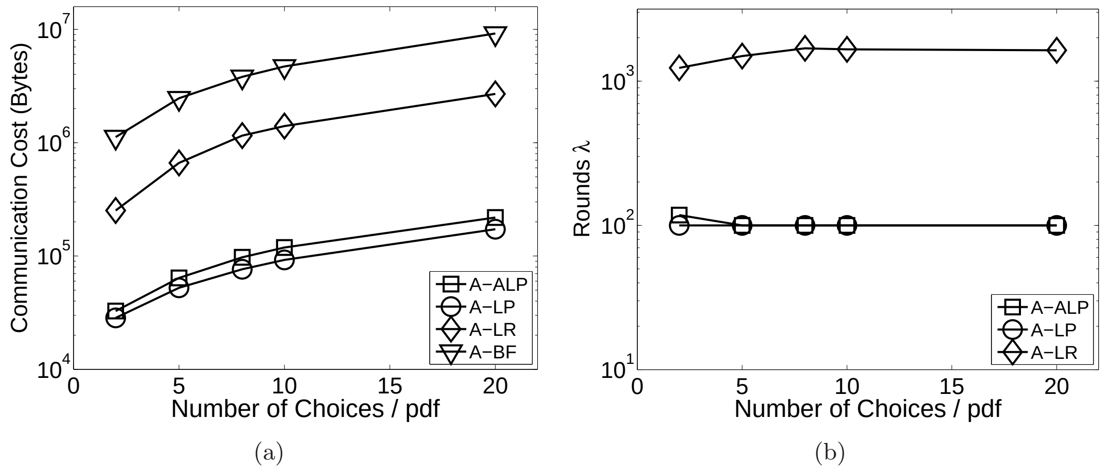
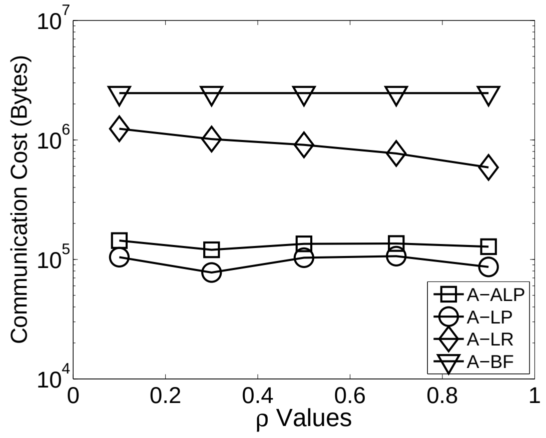


Figure 5.7. Varying b on Synthetic Gaussian dataset: effect on (a) communication and (b) number of rounds.

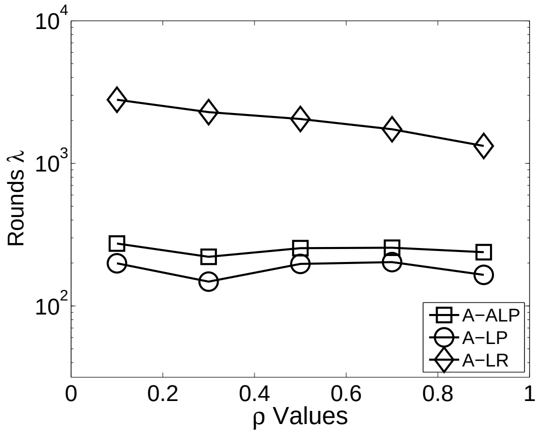
Figure 5.8 (note $N = 56,000$, $m = 10$, $\eta = 1\%$, $k = 100$) studies how this affects the algorithms using the *chlorine* dataset. Obviously, this has no effect on the *A-BF* algorithm. Interestingly, Figure 5.8(a) and 5.8(b) indicate that both the communication cost and the number of rounds required for *A-LR*, *A-LP* and *A-ALP* algorithms actually reduce on more skewed distributions.

5.7.8 Results with different η

We then study the impact of the budget η of the approximate $q_i^*(v)$ for the *A-ALP* algorithm. Intuitively, smaller ηs reduce the communication cost of transmitting these approximate $q_i^*(v)$s, but also reduce their quality of approximation leading to larger number of rounds. So there is expected to be some sweet spot for the choice of η. It turns out that a fairly small η already reaches the sweet spot. Figure 5.9 (note $N = 56,000$, $m = 10$, $k = 100$) shows the results on all four datasets. As seen in Figure 5.9(a), the communication cost drops sharply when η increases from 0% to 1% on all datasets. Note that $\eta = 0\%$ means that $q_i^*(v)$ contains only the two boundary points from $q_i(v)$. This indicates that by just adding a small number of points into $q_i^*(v)$, it does a very good job at representing $q_i(v)$ and hence gives a pretty tight lower bound $r^*(X, \mathcal{D}_i)$ on the estimated local rank of any unseen tuple X using the upper bound τ for the expected score of X. This is also evident from Figure 5.9(b) where the number of rounds drops significantly when η changes from 0% to 1%. As η gets larger, the overhead of communicating $q_i^*(v)$ starts to offset the communication savings. These results confirm that our algorithm does an excellent job in finding the

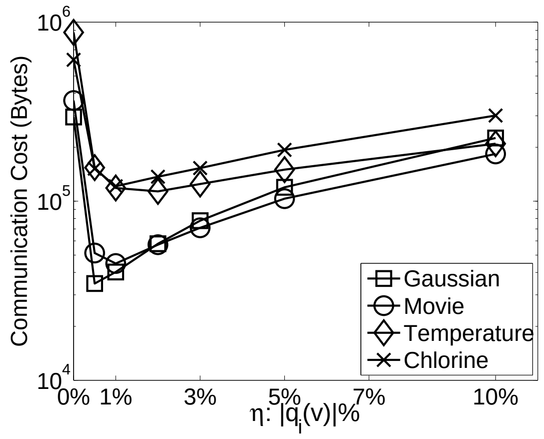


(a)

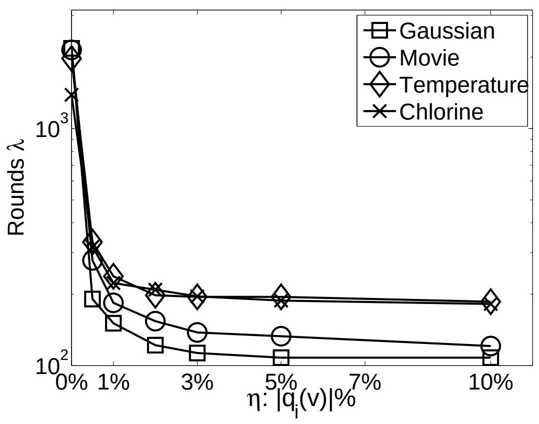


(b)

Figure 5.8. Varying ρ (skewness of the pdf) on Chlorine dataset: effect on (a) communication and (b) number of rounds.



(a)



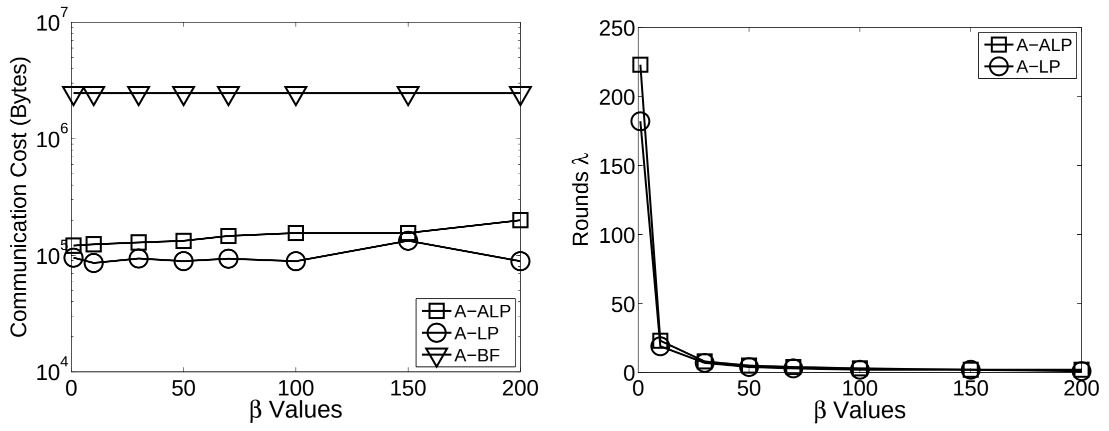
(b)

Figure 5.9. Effect of η on approximate $q_i(v)$s: effect on (a) communication and (b) number of rounds.

optimal representation of $q_i(v)$s given a limited budget. In practice, a tiny budget as small as 1% already seems good enough. A small η also reduces the computation cost for both the server and the distributed sites. Thus, the algorithm *A-ALP* is both computation- and communication-efficient. It requires minimal computational resources from the distributed sites, since only addition, subtraction, and multiplication operations are needed to compute the $q_i^*(v)$s. Subsequently, solving LPs is only done at the server.

5.7.9 Results with different β

In all the experiments above, in each round, our algorithms process only one tuple and then immediately check if it is safe to terminate. This causes a long latency if there are network delays. However, as argued in Section 5.6, we can easily reduce the latency by processing β tuples per round before checking the termination condition. This will reduce the number of rounds by a factor of β while only incurring an *additive* communication overhead of at most β tuples. In the last set of experiments, we empirically study the effects of β. The results are shown in Figure 5.10 (note $N = 56,000$, $m = 10$, $k = 100$, $\eta = 1\%|q_i(v)|$). First, as expected, the number of rounds is greatly reduced as β gets larger. For $\beta = 100$, both *A-LP* and *A-ALP* just need 2 rounds to complete, i.e., 2 round trips of communication. On the other hand, in this particular case, since the β value is still quite small when the number of rounds have been reduced to just 2, the increase in the total communication cost is almost negligible. This can be explained by the fact that although having a larger β makes the algorithms send $\leq \beta$ more tuples, we also reduce the communication cost of checking the termination condition repeatedly. This results in a net



(a) (b)

Figure 5.10. Varying β on Chlorine dataset: effect on (a) communication and (b) number of rounds.

effect of quite flat curves that we see from Figure 5.10, meaning that the algorithms are both communication-efficient and fast.

In general, there is obviously a trade-off between the number of rounds and the total communication cost. In the extreme case, when β equals the total size of the distributed data sets (N) from all sites, this approach degrades to the *A-BF* approach. For any case with $\beta > 1$, the number of tuples retrieved by the server will be larger than the case with $\beta = 1$, resulting in a communication overhead. However, larger β values also imply that the server only has to communicate a single tuple to all sites to get the lower bounds back from every site after seeing β tuples, except in the case of *A-ALP* where the lower bounds are computed locally by the server. In the latter case, larger β values reduce the computation overhead at the server, i.e., the server only needs to do the LPs once for every β tuples. For other algorithms, for small values of β, the savings of only retrieving the lower bounds once after seeing every β tuples cancels off the overhead of retrieving more tuples over the "optimal" terminating point with $\beta = 1$, as they cannot miss the "optimal" point by more than β tuples. We run this experiment with small β values (up to 200) as this already reduces the total rounds to just 1 or 2 for all datasets. If we keep increasing β, there will be a cut-off value where the delayed termination (have to look at many more tuples beyond the "optimal" point) will eventually result in more communication overhead than the savings.

5.7.10 Computation cost of solving the linear programs

Our main focus in this chapter is to save the communication cost. However, in practice, the computation overhead should not be ignored. Our main algorithms, namely the *A-LP* and *A-ALP*, require solving the linear programs (LPs). It is interesting to examine the associated computation overhead. In our experiments, we found that such overheads are quite small. All of our experiments were executed on a linux machine with an Intel Xeon CPU 5130@2GHz and 4GB memory. On this machine, solving the LP in each round takes only a few seconds at most, and our best algorithm takes only two or tree rounds (the optimization with a β value that is larger than 1). The GLPK library is extremely efficient. Note that we cannot assume the distributed sites in real world applications are powerful, and this is precisely the reason why we wanted to migrate the computation cost to the server with the *A-ALP* algorithm.

5.8 Related Work

There has been a large amount of efforts devoted to modeling and processing uncertain data, so we survey only the works most relevant to ours. TRIO [82,94], MayBMS [109,131], and MystiQ [81] are promising systems that are currently being developed. Many query processing and indexing techniques have been studied for uncertain databases and the most relevant works to this chapter are top-k queries [85–87,90]—their definitions and semantics have been discussed in detail in Chapter 4 and the expected rank approach was shown to have important properties that others do not guarantee. Techniques used include the Monte Carlo approach of sampling possible worlds [85], AI-style branch-and-bound search of the probability state space [86], dynamic programming approaches [87], and applying tail (Chernoff) bounds to determine when to prune [90]. There is ongoing work to understand top-k queries in a variety of contexts. For example, the work of Lian and Chen [105] deals with ranking objects based on spatial uncertainty, and ranking based on linear functions. Recently, Soliman *et al.* [88] have extended their study on top-k queries [86] to Group-By aggregate queries.

To the best of our knowledge, this is the first work on query processing on distributed probabilistic data. Distributed top-k queries have been extensively studied in certain data, including both the initial computation of the top-k [27,28,72,126,132] and the incremental monitoring/update version [120,122]. Our work falls into the first category. Some works consider minimizing the scan depth at each site to be the top priority, i.e., the number of tuples a site has to access, such as the seminal work by Fagin *et al.* [72]. Arguably, the more important metric for distributed systems is to minimize the communication cost [27,28,122], which is our objective.

To capture more complex correlations among tuples, more advanced rules and processing techniques are needed in the uncertainty data model. Recent works based on graphical probabilistic models and Bayesian networks have shown promising results in both offline [107] and streaming data [108]. Converting prior probability into posterior probability also offers positive results [133]. In these situations, the general approaches are using Monte-Carlo simulations [85, 97] to obtain acceptable approximations or inference rules from graphical model and Bayesian networks, e.g., [133, 134].

5.9 Conclusion

In this chapter we study of the emerging challenge of *distributed data*, extending our study in Chapter 4 to the distributed and parallel setting using the *expected ranks* ranking operator to summarize massive distributed uncertain data. This is the first work that studies ranking queries for distributed probabilistic data. We show that significant communication costs can be saved by exploring the interplay between the probabilities and the scores. We also demonstrate how to alleviate the computation burden at distributed sites, e.g., such as sensors in a sensor network or compute nodes in a cluster, so that communication and computation efficiency are achieved simultaneously. Many ranking semantics are still possible in the context of probabilistic data, extending our framework to those cases is an important open work, e.g., the median and quantile ranks discussed in Chapter 4. Finally, when updates are present at distributed sites, how to incrementally track (or monitor) the top-k results is also an intriguing open problem.

CHAPTER 6

OTHER WORKS

Though not included as part of this thesis, there are several other works which we have contributed to, in collaboration with other students, during our PhD study, including: (1) studying an alternative data summary to the histogram for *large data*, namely the kernel density estimate [135]; (2) studying the similarity join operator over both uncertain string data and spatial data [136,137]; (3) and monitoring distributed probabilistic data [138]. We will briefly mention these works here.

We studied the challenge of dealing with the massive *size* of data and summarizing it in Chapter 2 using the wavelet histogram. We explore another data summary, the kernel density estimate in [135], where the kernel density estimate may be thought of as a smooth histogram. The construction of kernel density estimates has been well-studied. However, existing techniques are expensive on massive datasets and/or only provide heuristic approximations without theoretical guarantees. We propose randomized and deterministic algorithms with quality guarantees which are orders of magnitude more efficient than previous algorithms. We demonstrate how to implement our ideas in a centralized setting and in *MapReduce*, although our algorithms are applicable to any large-scale parallel and distributed data processing framework. Extensive experiments on large real datasets demonstrate the quality, efficiency, and scalability of our techniques.

When studying the similarity join operator, we begin by looking at similarity join between uncertain strings [136]. Uncertainties arising in strings are a natural phenomena and occur in many applications including data cleaning, data integration, and scientific computing. We noticed that despite intense efforts in processing (deterministic) string joins and managing probabilistic data, modeling and processing probabilistic string joins had been a largely unexplored territory. Therefore, we studied the string join problem in probabilistic string databases, using the expected edit distance (EED) as the similarity measure. Extensive experiments on real data demonstrated order-of-magnitude improvements of our approaches over the baseline.

We also study the similarity join operator with respect to spatial data in [137], where the problem of interest was to efficiently obtain the kNN join between two datasets over a spatial attribute, which is to produce the k nearest neighbors (NN), from a dataset S, of every point in a dataset R. Since it involves both the join and the NN search, performing kNN joins efficiently is a challenging task. As argued in Chapter 1, the scale of data we are seeing these days is soaring out of control and a popular model nowadays for large-scale data processing is using a MapReduce cluster. Hence, how to execute kNN joins efficiently on large data that are stored in a MapReduce cluster is an intriguing problem that meets many practical needs. In this work, we propose novel (exact and approximate) algorithms in MapReduce to perform efficient parallel kNN joins on large data. We demonstrate our ideas using Hadoop. Extensive experiments in large real and synthetic datasets, with tens or hundreds of millions of records in both R and S and up to 30 dimensions, have demonstrated the efficiency, effectiveness, and scalability of our methods.

Finally, we also study how to efficiently monitor distributed probabilistic data in [138]. In this particular distributed setting, a primary concern is monitoring the distributed data and generating an alarm when a user-specified constraint is violated. A particular useful instance is the threshold-based constraint, which is commonly known as the distributed threshold monitoring problem [119,124,139,140]. This work extends this useful and fundamental study to distributed probabilistic data that emerge in a lot of applications, where uncertainty naturally exists when massive amounts of data are produced at multiple sources in distributed, networked locations (as mentioned in Chapter 5). When dealing with probabilistic data, there are two thresholds involved, the score and the probability thresholds. One must monitor both simultaneously, as such techniques developed for deterministic data are no longer directly applicable. This work presents a comprehensive study to this problem. Our algorithms have significantly outperformed the baseline method in terms of both the communication cost (number of messages and bytes) and the running time, as shown by an extensive experimental evaluation using several, real large datasets.

CHAPTER 7

CONCLUSIONS

Our PhD study has focused on the end goal of taking massive datasets and making concise summaries on top of these huge data, as shown in Figure 1.3(a), and then using these data summaries to accelerate data analytics tasks, as shown in Figure 1.3(b). One of the most important aspects of our approach is to always look for methods to construct summaries which can still give quality guarantees on the end results of a user's query, and this guarantee should be customizable via user parameter. After constructing such summaries, data analytics tasks can become orders of magnitude more efficient and potentially drastically reduce heat inside clusters and data centers as well as communication cost in sensor networks.

Throughout our study, we identified and focused on four challenges emerging from massive data: (1)*size*; (2) *complex structure and rich semantics*; (3) *uncertain data*; (4) and *distributed data*. During our study, we proposed the use of the *histogram* and *ranking operator* to deal with these emerging challenges. An illustration of some of the emerging challenges from massive data, as well as our proposed data summaries to cope with these challenges, is summarized in Figure 1.2.

Throughout our research, it has become apparent that there is no one-size-fits-all solution for constructing data summaries. Sometimes it involves *smart* random sampling schemes, clever indexing schemes, adapting or modifying existing algorithms from the data mining and streaming communities, or developing novel algorithms completely from scratch. In fact, during our PhD study, we have used all of these schemes, and in some cases have combined schemes, in order to produce the *histogram* and *ranking operator* as data summaries to cope with emerging massive data challenges. These techniques are by no means all-inclusive and we believe only represent the tip of the iceberg in terms of data summary construction techniques.

One common misconception is that not all data summaries will be useful for all data analytics tasks or applications. For instance, oftentimes the *histogram* and *ranking operators* we have proposed in our research may potentially eliminate outliers, depending on how these data summaries are constructed, which may be the most interesting part of the data depending on the data analytics. Therefore, the data analytics required of an application or user as well as the quality guarantees will greatly shape the design of any data summary which may be used to accelerate these tasks. Given this insight, there is certainly a huge space remaining to be investigated in regards to data summaries and which data analytics tasks are supported by them.

REFERENCES

[1] B. I. Lowdown, “The 10 largest databases in the world,” Business Report, 2007, http://www.andymars.com/Papers_IT_Ed_Computers.htm or www.businessintelligencelowdown.com/2007/02/top_10_largest_.html.

[2] L. Dignan, “Techlines panelist profile: Nasa’s nicholas skytland on big data literacy,” Between the Lines, Report, 2012, http://www.zdnet.com/techlines-panelist-profile-nasas-nicholas-skytland-on-big-data-literacy-7000003452/.

[3] Bing, “Bing maps publishes equivalent of 100,000 dvds of birds eye imagery,” Business Report, 2013, http://www.bing.com/blogs/site_blogs/b/maps/archive/2013/06/11/largest-shipment-of-bird-s-eye-100-000-dvds-of-imagery.aspx.

[4] Google, “Facts about google and competition,” http://www.google.com/competition/howgooglesearchworks.html.

[5] J. Horel, M. Splitt, L. Dunn, J. Pechmann, B. White, C. Ciliberti, S. Lazarus, J. Slemmer, D. Zaff, and J. Burks, “Mesowest: cooperative mesonets in the western united states,” *Bull. Amer. Meteor. Soc.*, vol. 83, no. 2, pp. 211–226, 2002.

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The google file system,” in *SOSP*, 2003, pp. 29–43.

[7] Hadoop Project, http://hadoop.apache.org/.

[8] F. Li, B. Yao, M. Tang, and M. Hadjieleftheriou, “Spatial approximate string search,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 6, pp. 1394–1409, 2013.

[9] C. Li, B. Wang, and X. Yang, “Vgram: improving performance of approximate queries on string collections using variable-length grams,” in *VLDB*, 2007.

[10] H. Shatkay and S. B. Zdonik, “Approximate Queries and Representations for Large Data Sequences,” in *ICDE*, 1996, pp. 536–545.

[11] Y. Matias, J. S. Vitter, and M. Wang, “Wavelet-based histograms for selectivity estimation,” in *SIGMOD*, 1998, pp. 448–459.

[12] Y. Matias, J. V. Scott, and M. Wang, “Dynamic maintenance of wavelet-based histograms,” in *VLDB*, 2000, pp. 101–110.

[13] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss, “Surfing wavelets on streams: One-pass summaries for approximate aggregate queries,” in *VLDB*, 2001, pp. 79–88.

[14] G. Cormode, M. Garofalakis, and D. Sacharidis, "Fast approximate wavelet tracking on streams," in *EDBT*, 2006, pp. 4–22.

[15] MemeTracker Project, http://memetracker.org/.

[16] COAPS, "The Center for Ocean-Atmospheric Prediction Studies at Florida State University." http://www.coaps.fsu.edu/.

[17] J. Jestes, K. Yi, and F. Li, "Building wavelet histograms on large data in MapReduce," *PVLDB*, vol. 5, no. 2, pp. 109–120, 2012.

[18] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita, "Improved histograms for selectivity estimation of range predicates," in *SIGMOD*, 1996, pp. 294–305.

[19] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150.

[20] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *SIGMOD*, 2009, pp. 165–178.

[21] M. Garofalakis and P. B. Gibbons, "Wavelet synopses with error guarantees," in *SIGMOD*, 2002, pp. 476–487.

[22] K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim, "Approximate query processing using wavelets," *VLDBJ*, vol. 10, no. 2-3, pp. 199–223, 2001.

[23] S. Guha and B. Harb, "Wavelet synopsis for data streams: minimizing non-euclidean error," in *SIGKDD*, 2005, pp. 88–97.

[24] C. C. Aggarwal, "On effective classification of strings with wavelets," in *SIGKDD*, 2002, pp. 163–172.

[25] G. Sheikholeslami, S. Chatterjee, and A. Zhang, "WaveCluster: a wavelet-based clustering approach for spatial data in very large databases," *VLDBJ*, vol. 8, no. 3-4, pp. 289–304, 2000.

[26] R. Fagin, A. Lotem, and M. Noar, "Optimal aggregation algorithms for middleware," *Journal of Computer and System Sciences*, vol. 66, pp. 614–656, 2003.

[27] P. Cao and Z. Wang, "Efficient top-k query calculations in distributed networks," in *PODC*, 2004, pp. 206–215.

[28] S. Michel, P. Triantafillou, and G. Weikum, "KLEE: a framework for distributed top-k query algorithms," in *VLDB*, 2005, pp. 637–648.

[29] B. Patt-Shamir and A. Shafrir, "Approximate distributed top-k queries," *Distributed Computing*, vol. 21, pp. 1–22, 2008.

[30] V. N. Vapnik and A. Y. Chervonenkis, "On the uniform convergence of relative frequencies of events to their probabilities," *Theory of Probability and its Applications*, vol. 16, pp. 264–280, 1971.

[31] R. Srinivasan, *Importance sampling - Applications in communications and detection.* Springer-Verlag, 2002.

[32] Z. Huang, K. Yi, Y. Liu, and G. Chen, "Optimal sampling algorithms for frequency estimation in distributed data," in *IEEE INFOCOM*, 2011, pp. 1997–2005.

[33] Q. Zhao, M. Ogihara, H. Wang, and J. Xu, "Finding global icebergs over distributed data sets," in *PODS*, 2006, pp. 298–307.

[34] G. Cormode and M. Hadjieleftheriou, "Finding frequent items in data streams," in *VLDB*, 2008, pp. 1530–1541.

[35] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," in *STOC*, 1996, pp. 20–29.

[36] M. Arlitt and T. Jin, "Workload characterization of the 1998 world cup web site," IEEE Network, Tech. Rep., 1999.

[37] D. Jiang, B. C. Ooi, L. Shi, and S. Wu, "The performance of MapReduce: An in-depth study," *PVLDB*, vol. 3, no. 1, pp. 472–483, 2010.

[38] R. Vernica, M. J. Carey, and C. Li, "Efficient parallel set-similarity joins using MapReduce," in *SIGMOD*, 2010, pp. 495–506.

[39] F. N. Afrati and J. D. Ullman, "Optimizing joins in a map-reduce environment," in *EDBT*, 2010, pp. 99–110.

[40] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton, "MAD skills: New analysis practices for big data," *PVLDB*, vol. 2, no. 2, pp. 1481–1492, 2009.

[41] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "SCOPE: easy and efficient parallel processing of massive data sets," *PVLDB*, vol. 1, no. 2, pp. 1265–1276, 2008.

[42] Amazon EC2, http://aws.amazon.com/ec2/.

[43] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz, "HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads," *PVLDB*, vol. 2, no. 1, pp. 922–933, 2009.

[44] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, "Hadoop++: Making a yellow elephant run like a cheetah," *PVLDB*, vol. 3, no. 1, pp. 518–529, 2010.

[45] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce online," in *NSDI*, 2010, pp. 21–21.

[46] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy, and R. Sears, "Online aggregation and continuous query support in MapReduce," in *SIGMOD*, 2010, pp. 1115–1118.

[47] M. Son and H. Im, "Parallel top-k query processing using MapReduce," Pohang University of Science and Technology, Tech. Rep., 2010.

[48] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel, "Optimal histograms with quality guarantees," in *VLDB*, 1998, pp. 275–286.

[49] J. Jestes, J. Phillips, F. Li, and M. Tang, "Ranking large temporal data," *PVLDB*, vol. 5, no. 11, pp. 1412–1423, 2012.

[50] I. F. Ilyas, G. Beskales, and M. A. Soliman, "Survey of top-k query processing techniques in relational database systems," *ACM Computing Surveys*, vol. To Appear, 2008.

[51] F. Li, K. Yi, and W. Le, "Top-k queries on temporal data," *VLDB Journal*, vol. 19, no. 5, pp. 715–733, 2010.

[52] Q. Chen, L. Chen, X. Lian, Y. Liu, and J. X. Yu, "Indexable PLA for efficient similarity search," in *VLDB*, 2007, pp. 435–446.

[53] A. Anagnostopoulos, M. Vlachos, M. Hadjieleftheriou, E. Keogh, and P. S. Yu, "Global distance-based segmentation of trajectories," in *KDD*, 2006, pp. 34–43.

[54] E. J. Keogh, S. Chu, D. Hart, and M. J. Pazzani, "An online algorithm for segmenting time series," in *ICDM*, 2001, pp. 289–296.

[55] B. Jiang and J. Pei, "Online interval skyline queries on time series," in *ICDE*, 2009, pp. 1036–1047.

[56] T. Palpanas, M. Vlachos, E. Keogh, D. Gunopulos, and W. Truppel, "Online amnesic approximation of streaming time series," in *ICDE*, 2004, pp. 338–349.

[57] M. L. Lee, W. Hsu, L. Li, and W. H. Tok, "Consistent top-k queries over time," in *DASFAA*, 2009, pp. 51–65.

[58] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant, "Range queries in OLAP data cubes," in *SIGMOD*, 1997, pp. 73–88.

[59] H.-P. Kriegel, M. Pötke, and T. Seidl, "Managing intervals efficiently in object-relational databases," in *VLDB*, 2000, pp. 407–418.

[60] L. Arge and J. S. Vitter, "Optimal external memory interval management," *SICOMP*, vol. 32, no. 6, pp. 1488–1508, 2003.

[61] L. Arge and J. S. Vitter, "Optimal dynamic interval management in external memory," in *FOCS*, 1996, pp. 560–569.

[62] D. Zhang, A. Markowetz, V. J. Tsotras, D. Gunopulos, and B. Seeger, "On computing temporal aggregates with range predicates," *TODS*, vol. 33, no. 2, pp. 1–39, 2008.

[63] J. Yang and J. Widom, "Incremental computation and maintenance of temporal aggregates," *VLDB Journal*, vol. 12, no. 3, pp. 262–283, 2003.

[64] G. S. Brodal and J. Katajainen, "Worst-case efficient external-memory priority queues," in *SWAT*, 1998, pp. 107–118.

[65] J. Jestes, J. M. Phillips, F. Li, and M. Tang, "Ranking large temporal data," School of Computing, University of Utah, Technical Report, 2012, http://www.cs.utah.edu/∼jestes/ranktaggtr.pdf.

[66] J. Shieh and E. Keogh, "iSAX: indexing and mining terabyte sized time series," in *KDD*, 2008, pp. 623–631.

[67] L. Arge, O. Procopiuc, and J. S. Vitter, "Implementing I/O-efficient data structures using TPIE," in *ESA*, 2002, pp. 88–100.

[68] L. H. U, N. Mamoulis, K. Berberich, and S. Bedathur, "Durable top-k search in document archives," in *SIGMOD*, 2010, pp. 555–566.

[69] Y. Tao, D. Papadias, and C. Faloutsos, "Approximate temporal aggregation," in *ICDE*, 2004, pp. 190–201.

[70] Y. Tao and X. Xiao, "Efficient temporal counting with bounded error," *VLDB Journal*, vol. 17, no. 5, pp. 1271–1292, 2008.

[71] J. Jestes, G. Cormode, F. Li, and K. Yi, "Semantics of ranking queries for probabilistic data," *IEEE TKDE*, vol. 23, pp. 1903–1917, 2011.

©2011 IEEE. Reprinted, with permission, from J. Jestes, G. Cormode, F. Li, and K. Yi, "Semantics of ranking queries for probabilistic data", IEEE TKDE, December 2011.

[72] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," in *PODS*, 2001.

[73] C. Li, K. C.-C. Chang, I. Ilyas, and S. Song, "RankSQL: Query algebra and optimization for relational top-k queries," in *SIGMOD*, 2005.

[74] I. F. Ilyas, W. G. Aref, A. K. Elmagarmid, H. Elmongui, R. Shah, and J. S. Vitter, "Adaptive rank-aware query optimization in relational databases," *ACM TODS*, vol. 31, no. 4, pp. 1257–1304, 2006.

[75] D. Xin, J. Han, and K. C.-C. Chang, "Progressive and selective merge: Computing top-k with ad-hoc ranking functions," in *SIGMOD*, 2007.

[76] A. Halevy, A. Rajaraman, and J. Ordille, "Data integration: the teenage year," in *VLDB*, 2006.

[77] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani, "Robust and efficient fuzzy match for online data cleaning," in *SIGMOD*, 2003.

[78] M. A. Hernandez and S. J. Stolfo, "Real-world data is dirty: Data cleansing and the merge/purge problem," *Data Mining and Knowledge Discovery*, vol. 2, no. 1, pp. 9–37, 1998.

[79] R. Cheng, D. Kalashnikov, and S. Prabhakar, "Evaluating probabilistic queries over imprecise data," in *SIGMOD*, 2003.

[80] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong, "Model-driven data acquisition in sensor networks," in *VLDB*, 2004.

[81] N. Dalvi and D. Suciu, "Efficient query evaluation on probabilistic databases," *VLDB Journal*, vol. 16, no. 4, pp. 523–544, 2007.

[82] P. Agrawal, O. Benjelloun, A. Das Sarma, C. Hayworth, S. Nabar, T. Sugihara, and J. Widom, "Trio: A system for data, uncertainty, and lineage," in *VLDB*, 2006.

[83] S. Singh, C. Mayfield, S. Mittal, S. Prabhakar, S. E. Hambrusch, and R. Shah, "The orion uncertain data management system," in *COMAD*, 2008.

[84] L. Antova, C. Koch, and D. Olteanu, "10^{10^6} worlds and beyond: Efficient representation and processing of incomplete information," in *ICDE*, 2007.

[85] C. Re, N. Dalvi, and D. Suciu, "Efficient top-k query evaluation on probalistic databases," in *ICDE*, 2007.

[86] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang, "Top-k query processing in uncertain databases," in *ICDE*, 2007.

[87] X. Zhang and J. Chomicki, "On the semantics and evaluation of top-k queries in probabilistic databases," in *DBRank*, 2008.

[88] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang, "Probabilistic top-k and ranking-aggregate queries," *ACM TODS*, vol. 33, no. 3, pp. 13:1–13:54, 2008.

[89] K. Yi, F. Li, D. Srivastava, and G. Kollios, "Efficient processing of top-k queries in uncertain databases," AT&T Labs, Inc., Tech. Rep., 2007.

[90] M. Hua, J. Pei, W. Zhang, and X. Lin, "Ranking queries on uncertain data: A probabilistic threshold approach," in *SIGMOD*, 2008.

[91] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom, "ULDBs: databases with uncertainty and lineage," in *VLDB*, 2006.

[92] Y. Tao, R. Cheng, X. Xiao, W. K. Ngai, B. Kao, and S. Prabhakar, "Indexing multi-dimensional uncertain data with arbitrary probability density functions," in *VLDB*, 2005.

[93] J. Li, B. Saha, and A. Deshpande, "A unified approach to ranking in probabilistic databases," *PVLDB*, vol. 2, no. 1, pp. 502–513, 2009.

[94] A. D. Sarma, O. Benjelloun, A. Halevy, and J. Widom, "Working models for uncertain data," in *ICDE*, 2006.

[95] S. Singh, C. Mayfield, S. Mittal, S. Prabhakar, S. Hambrusch, and R. Shah, "Orion 2.0: native support for uncertain data," in *SIGMOD*, 2008.

[96] A. Fuxman, E. Fazli, and R. J. Miller, "ConQuer: efficient management of inconsistent databases," in *SIGMOD*, 2005.

[97] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. M. Jermaine, and P. J. Haas, "MCDB: a monte carlo approach to managing uncertain data," in *SIGMOD*, 2008.

[98] J. Pei, B. Jiang, X. Lin, and Y. Yuan, "Probabilistic skylines on uncertain data," in *VLDB*, 2007.

[99] Q. Zhang, F. Li, and K. Yi, "Finding frequent items in probabilistic data," in *SIGMOD*, 2008.

[100] V. Ljosa and A. Singh, "APLA: Indexing arbitrary probability distributions," in *ICDE*, 2007.

[101] S. Singh, C. Mayfield, S. Prabhakar, R. Shah, and S. Hambrusch, "Indexing uncertain categorical data," in *ICDE*, 2007.

[102] G. Beskales, M. A. Soliman, and I. F. Ilyas, "Efficient search for the top-k probable nearest neighbors in uncertain databases," in *VLDB*, 2008.

[103] V. Ljosa and A. K. Singh, "Top-k spatial joins of probabilistic objects," in *ICDE*, 2008.

[104] M. Hua, J. Pei, W. Zhang, and X. Lin, "Efficiently answering probabilistic threshold top-k queries on uncertain data," in *ICDE*, 2008.

[105] X. Lian and L. Chen, "Probabilistic ranked queries in uncertain databases," in *EDBT*, 2008.

[106] T. Ge, S. Zdonik, and S. Madden, "Top-k queries on uncertain data: on score distribution and typical answers," in *SIGMOD*, 2009.

[107] P. Sen and A. Deshpande, "Representing and querying correlated tuples in probabilistic databases," in *ICDE*, 2007.

[108] B. Kanagal and A. Deshpande, "Online filtering, smoothing and probabilistic modeling of streaming data," in *ICDE*, 2008.

[109] L. Antova, T. Jansen, C. Koch, and D. Olteanu, "Fast and simple relational processing of uncertain data," in *ICDE*, 2008.

[110] J. G. Shanthikumar and M. Shaked, *Stochastic Orders and Their Applications*. Academic Press, 1994.

[111] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar, "Rank aggregation methods for the web," in *WWW Conference*, 2001.

[112] R. Fagin, R. Kumar, and D. Sivakumar, "Comparing top k lists," in *ACM-SIAM Symposium on Discrete Algorithms*, 2003.

[113] S. Borzsonyi, D. Kossmann, and K. Stocker, "The skyline operator," in *ICDE*, 2001.

[114] T. Bernecker, H.-P. Kriegel, M. Renz, F. Verhein, and A. Zuefle, "Probabilistic frequent itemset mining in uncertain databases," in *KDD*, 2009.

[115] R. Cheng, J. Chen, M. Mokbel, and C.-Y. Chow, "Probabilistic verifiers: Evaluating constrained nearest-neighbor queries over uncertain data," in *ICDE*, 2008.

[116] L. L. Cam, "An approximation theorem for the poisson binomial distribution," *Pacific Journal of Mathematics*, vol. 10, no. 4, pp. 1181–1197, 1960.

[117] W. Hoeffding, "On the distribution of the number of successes in independent trials," *Annals of Mathematical Statistics*, vol. 27, no. 3, pp. 713–721, 1956.

[118] F. Li, K. Yi, and J. Jestes, "Ranking distributed probabilistic data," in *SIGMOD*, 2009, pp. 361–374, ©2009 ACM, Inc. http://doi.acm.org/10.1145/1559845.1559885.

[119] I. Sharfman, A. Schuster, and D. Keren, "A geometric approach to monitoring threshold functions over distributed data streams," in *SIGMOD*, 2006.

[120] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston, "Finding (recently) frequent items in distributed data streams," in *ICDE*, 2005.

[121] G. Cormode, M. Garofalakis, S. Muthukrishnan, and R. Rastogi, "Holistic aggregates in a networked world: distributed tracking of approximate quantiles," in *SIGMOD*, 2005.

[122] B. Babcock and C. Olston, "Distributed top-k monitoring," in *SIGMOD*, 2003.

[123] R. Huebsch, M. Garofalakis, J. M. Hellerstein, and I. Stoica, "Sharing aggregate computation for distributed queries," in *SIGMOD*, 2007.

[124] S. Jeyashanker, S. Kashyap, R. Rastogi, and P. Shukla, "Efficient constraint monitoring using adaptive thresholds," in *ICDE*, 2008.

[125] M. Wu, J. Xu, X. Tang, and W.-C. Lee, "Top-k monitoring in wireless sensor networks," *IEEE TKDE*, vol. 19, no. 7, pp. 962–976, 2007.

[126] A. S. Silberstein, R. Braynard, C. Ellis, K. Munagala, and J. Yang, "A sampling-based approach to optimizing top-k queries in sensor networks," in *ICDE*, 2006.

[127] S. Chaudhuri, L. Gravano, and A. Marian, "Optimizing top-k selection queries over multimedia repositories," *IEEE TKDE*, vol. 16, no. 8, pp. 992–1009, 2004.

[128] GLPK, "GNU Linear Programming Kit," http://www.gnu.org/software/glpk/.

[129] J. Boulos, N. Dalvi, B. Mandhani, S. Mathur, C. Re, and D. Suciu, "MYSTIQ: a system for finding more answers by using probabilities," in *SIGMOD*, 2005.

[130] S. Papadimitriou, J. Sun, and C. Faloutsos, "Streaming pattern discovery in multiple time-series." in *VLDB*, 2005.

[131] L. Antova, C. Koch, and D. Olteanu, "From complete to incomplete information and back," in *SIGMOD*, 2007.

[132] D. Zeinalipour-Yazti, Z. Vagena, D. Gunopulos, V. Kalogeraki, V. Tsotras, M. Vlachos, N. Koudas, and D. Srivastava, "The threshold join algorithm for top-k queries in distributed sensor networks," in *DMSN*, 2005.

[133] C. Koch and D. Olteanu, "Conditioning probabilistic databases," in *VLDB*, 2008.

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

[135] Y. Zheng, J. Jestes, J. M. Phillips, and F. Li, "Quality and efficiency for kernel density estimates in large data," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD, 2013.

[136] J. Jestes, F. Li, Z. Yan, and K. Yi, "Probabilistic string similarity joins," in *SIGMOD*, 2010, pp. 327–338.

[137] C. Zhang, F. Li, and J. Jestes, "Efficient parallel KNN joins for large data in mapreduce," in *EDBT*, 2012, pp. 38–49.

[138] M. Tang, F. Li, J. M. Phillips, and J. Jestes, "Efficient threshold monitoring for distributed probabilistic data," in *ICDE*, 2012, pp. 1120–1131.

[139] R. Keralapura, G. Cormode, and J. Ramamirtham, "Communication-efficient distributed monitoring of thresholded counts," in *SIGMOD*, 2006.

[140] G. Cormode, S. Muthukrishnan, and K. Yi, "Algorithms for distributed functional monitoring," in *Proc. ACM-SIAM Symposium on Discrete Algorithms*, 2008.