Efficient Processing of Top-k Queries on Spatial and Temporal Data

by

Zhitao Shen

B.E. Shanghai Jiao Tong Univeristy, 2006

M.E. University of Tsukuba, 2009

A THESIS SUBMITTED IN FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SCHOOL
OF
COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF
NEW SOUTH WALES

SYDNEY · AUSTRALIA

December, 2012

Originality Statement

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Signed .....

Name: Zhitao Shen

Abstract

Spatial and temporal databases play a vital role in many applications in different areas such as Geographic Information Systems (GIS), stock market, wireless sensor network, traffic monitoring and internet applications, etc. Due to their importance, a huge amount of work has focused on efficiently computing various spatial and temporal queries. Among the applications, end-users are more interested in the most important query answers in the potentially enormous answer space. Therefore, different types of information systems use various techniques to rank query answers and return the most important query results to users. Observed in real world scenario, top-k results are more interesting to users and a top-k query is a natural way to be asked to reflect a user's preference in terms of a user-defined scoring function. In this thesis, we provide efficient solutions for the top-k queries under various settings and different criteria for users' preferences. Specifically, we tackle three types of top-k queries in a systematic way. Below is a brief description of our contributions.

We are the first to study the efficient monitoring of top-k pairs queries over data streams. We present the first approach to answer a broad class of top-k pairs and top-k objects queries over sliding windows. Our framework handles multiple top-k queries and each query is allowed to use a different scoring function, a different value of k and a different size of the sliding window. Furthermore, the framework allows

the users to define arbitrarily complex scoring functions and supports out-of-order data streams.

We are the first to study the top-k loyalty queries. We propose a measure named loyalty that reflects how persistently an object satisfies the criteria. Formally, the loyalty of an object is the total time (in past T time units) it satisfied the query criteria. We propose an optimal approach to monitor the loyalty queries over sliding windows that continuously report k objects with the highest loyalties. We also experimentally verify the effectiveness of the proposed approach by comparing it with a classic sweep line algorithm.

We are the first to study the I/O efficient solution for depth-related problems which can be used for retrieving the top-k objects with linear scoring functions. Half-plane depth of a plane is the number of objects lying in the plane. Location depth of a point p is the minimum half-plane depth of any plane that is bounded by any line passing through p. We propose disk-based algorithms for a few important depth-related problems namely *k-depth contour*, *k-snippet* and *k-upper envelope*. We show that one of our proposed algorithms is I/O optimal for k-snippet and k-upper envelope problems.

Publications Involved in Thesis

- Zhitao Shen, Muhammad Aamir Cheema, Xuemin Lin, Wenjie Zhang, Haixun Wang. Efficiently Monitoring Top-k Pairs over Sliding Windows. in 28th IEEE International Conference on Data Engineering (ICDE), Washington DC, USA, 2012. (Chapter 3)

- Zhitao Shen, Muhammad Aamir Cheema, Xuemin Lin, Wenjie Zhang, Haixun Wang. A Generic Framework for Top-k Pairs and Top-k Objects Queries over Sliding Windows. in IEEE Transactions on Knowledge and Data Engineering (TKDE), (accepted in Sep 2012). (Special issue of IEEE-TKDE on the "Best Papers of ICDE 2012") (Chapter 3)

- Zhitao Shen, Muhammad Aamir Cheema, Xuemin Lin. Loyalty-based Selection: Retrieving Objects that Persistently Satisfy Criteria. in The 21st ACM International Conference on Information and Knowledge Management (CIKM), Maui, Hawaii, USA, 2012. (Chapter 4)

- Muhammad Aamir Cheema, Zhitao Shen, Xuemin Lin. A Unified Framework for Efficiently Processing Various Ranking Related Queries. Submitted to SIGMOD 2013. (Chapter 5)

Acknowledgements

First of all, I would like to deliver sincere gratitude to Prof. Xuemin Lin for supervising me during the past three and half years. I thank him for guiding me constantly through the road of my research and encouraging me to overcome the difficult times. I have been deeply impressed by his insight and breadth of knowledge in his research area as well as his continuous dedication to research work. Without his significant encouragements and supports, my research years will never be able to run smooth.

I would like to acknowledge the support from my senior colleague, Muhammad Aamir Cheema, together with whom I conducted research on many research topics and thus had a lot of fun. I would like to give my thanks to the senior colleagues, Dr. Ying Zhang and Dr. Wenjie Zhang who are always willing to provide their help whenever I am faced with difficulties in conducting research work.

Part of the work in this thesis is conducted in collaboration with Dr. Haixun Wang. I deliver my thanks to him for supporting the work presented in Chapter 3.

Besides, my thanks also go to my group members: Dr. Haichuan Shang, Dr. Chuan Xiao, Dr. Gaoping Zhu, Ke Zhu, Xiang Zhao, Weiren Yu, Liming Zhang, Jin Xu, Jianfen Zhang, Chengyuan Zhang. It is a great honor to work with all these talented people.

Last but not the least, I would also like to thank my parents and my wife,

Yiqiong, for their constant love, support and encouragement during my PhD study. Although they are not involved in this thesis, they are everywhere in my life supporting me with love and care.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In many applications domains, end-users are more interested in the most important query answers in the potentially enormous answer space. Therefore, different types of information systems use various techniques to rank query answers and return the most important query results (top-k results) to users. For instance, in the contents of spatial databases, users are interested in ranking query answers based on the distances to the query location. Similar applications exist in the context of data stream processing systems. Most of these applications compute queries that involve a certain ranking model to provide users with top-k results.

One common way to identify the top-k results is scoring all instances of the result based on some *scoring function*. A score of the instance acts as a valuation for that result according to its characteristics (e.g. distance-based score in spatial databases, or price and volume in the stock transaction databases). For data objects queries, the score is corresponding to one data object, while for data pairs queries, the score is corresponding to a pair of objects.

The spatial objects are composed of one or more points, lines / polygons which present the location information of the object. Figure 1.1 shows a map from Google

Maps (http://maps.google.com) obtained by entering the query "Find hotels near the University of New South Wales". The query results are shown in red circles and balloons labelled from A to J. The lines represent the facilities of moving through space or connections in space (i.e., roads, rivers). A region represents the spatial object for which its spatial extent is also important. A region may consist of disjoint pieces each containing many polygons. In Figure 1.1, University of New South Wales and Prince of Wales Hospital are represented by regions.

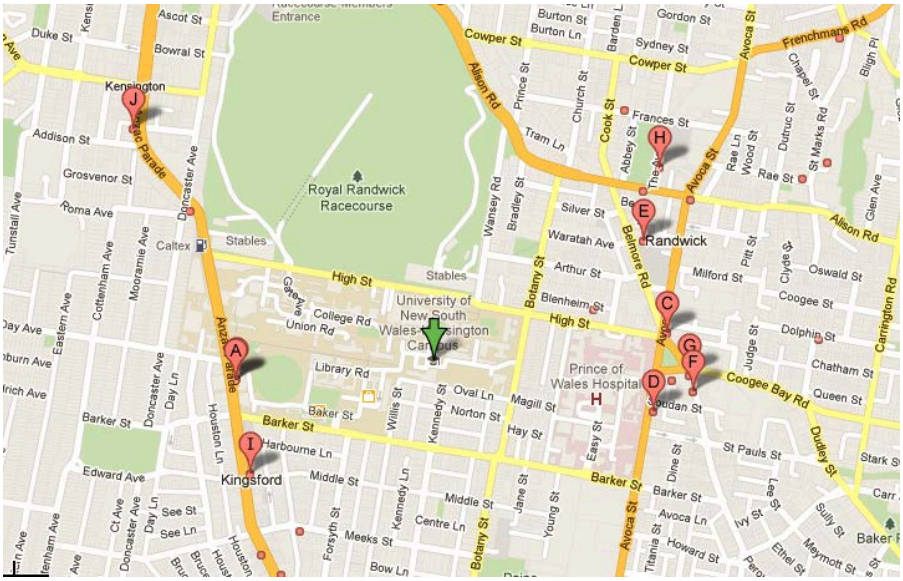


Figure 1.1: A top-k queries example in Google Map

As observed in Figure 1.1, a top-k query is processed to generate the most interesting results among the numerous relevant answers. Moreover, the example implies that the scoring function used by the top-k query is related to both the distance to the university and the textual relevance of an object. Thereby, the results shown on the map are ranked accordingly. The balloons illustrate the top-10 results with a ranking order indicated by the labels and the red circles without balloons describe another 10 less important results.

The temporal objects store the time related to the objects. In a more natural

way, the objects are usually collected chronologically in the form of data streams. In database area, A data stream is a massive real-time continuous sequence of data objects. The typical applications of data streams include sensor network, stock tickers, network traffic measurement, click streams and telecom call records. The main challenge of these applications is that the data objects arrive continuously and the volume of the data is difficult to store the entire data set in main memory. Therefore, the system sometimes need to drop some of the data objects due to high arrival speed.

1.1 A major challenge

One of the main challenges in answering the top-k queries on spatial and temporal data is that there is no nature total ordering among these kinds of data sets. A simple example is that a user wants to find 5 restaurants nearest to her location. A naive approach is to compute the distances of all the restaurants from her current location and then report the 5 closest restaurants. Therefore, one way to answer such a spatial top-k query is by sequentially scanning all database objects, computing the score of each instance according to the certain feature. However, this approach suffers from scalability problems with respect to database size and the number of features of objects. Assuming that the scoring functions are monotonic, an alternative way is to map the query into a join query that joins the output of multiple single-feature queries, and then sorts the joined results based on combined score. This approach also does not scale with respect to both number of features and database size since all join results have to be computed then sorted.

The main problem with sort-based approaches is that sorting is a blocking operation that requires full computation of the join results, which performs not

well for online processing. Moreover, although the input to the join operation is sorted on individual features, this order is not exploited by conventional join algorithms. Hence, sorting the join results becomes necessary to produce the top-k answers.

Additionally, for online processing of the temporal data, the input data varies dynamically. It is even more challenging to maintain the sorted join results over such data stream model (e.g. sliding window model), in which the objects may appear and expire dynamically. Therefore, embedding rank-awareness in top-k query processing techniques provides a more efficient and scalable solution.

1.2 Various Problem Settings

In this thesis, we mainly tackle three types of top-k queries under different problem settings in a systematic way. In Chapter 3, we study the top-k objects and pairs queries in a data stream model. Chapter 4 studies the problem of continuously monitoring top-k loyalty objects. Chapter 5 presents the techniques for answering top-k objects queries using linear scoring functions over large static databases. Below, we discuss the various problem settings investigated in the thesis.

1.2.1 Static Databases v.s. Data Streams Processing

The query processing over static databases has been extensively studied since the invention of database systems. Recently, "Big Data" has been a hot topic in both research and industry communities due to the recognition that the data set collected by databases could be extremely enormous. Therefore, query optimizers are demanded to provide efficient algorithms for the (top-k) query processing. Typically, for the design of any algorithm over large static databases, two types of

costs in terms of the number of I/O operations required and the CPU requirements should be carefully considered.

Beside this, in recent years, we have witnessed the widely recognised phenomenon of high speed data streams. A data stream is a massive real-time continuous sequence of data elements. The typical applications include sensor network, stock tickers, network traffic measurement, click streams and telecom call records. The main challenge of these applications is that the data element arrives continuously and the volume of the data is so large that they can hardly be stored in the main memory (even on the local disk) for online processing, and sometimes the system has to drop some data elements due to the high arriving speed. The data in the traditional database applications are organized on the hard disk by the Database Management System(DBMS) so the queries from the users can be answered by scanning the indices or the whole data set. Considering of the characteristics of the stream applications, it is not feasible to simply load the arriving data elements onto the DBMS and operate on them because the traditional DBMS's are not designed for rapid and continuous loading of individual data element and they do not directly support continuous queries that are typical of data stream applications

In the thesis, we focus on efficient data stream processing over the sliding window model. The sliding window model is extensively used in querying streaming data as well as analyzing the spatial and temporal data [BBD$^+$02, MBP06, LMT$^+$05, LYWL05, MP07a, BO79]. A sliding window maintains the most recent data which may be most interesting to the users. As the contents of the sliding windows evolve over time, it makes sense for user to ask a query once and receive the updated answers over time. For a fixed length of time period T, a sliding window contains all the query results within last T time units.

1.2.2 Objects Queries v.s. Pairs Queries

The traditional top-k objects queries compute the ranking of scores based on each object. Given a scoring function $s(o_i)$ that computes the score of one object, a top-k objects query returns k objects with the smallest scores.

Moreover, in many applications, we may consider the ranking of scores based on each pair of objects. Given a scoring function $s(o_i, o_j)$ that computes the score of a pair of objects (o_i, o_j), a top-k pairs query returns k pairs with the smallest scores among all possible pairs of objects. k closest pairs queries, k furthest pairs queries and their variants are some well studied examples of top-k pairs queries that rank the pairs on distance functions.

In the thesis, we study the problem of top-k pairs queries over data streams based on the sliding window model. Top-k pairs queries over sliding windows have many interesting applications. Consider the example of a data stream of ATM transactions. Assume that there are two transactions such that both the transactions belong to the same bank account and both are made within a small duration of time. The transactions may indicate a fraud if the distance between the ATM machines where these transactions were made is large (e.g., the transactions are made within 15 minutes of each other but are made from two different cities or even countries). Analysing such top-k pairs may not only help to detect fraud but may also help to understand the spatial and temporal relations between the transactions. The query below shows a simple example of such top-k pairs query over the transactions issued in last 24 hours.

1.2.3 Snapshot Queries v.s. Continuous Queries

Note that the set of objects in the sliding window changes dynamically as the new objects arrive and the old objects expire from the sliding window. Hence, some

users may be interested in continuous update of the results. In contrast, some users may only be interested in retrieving the top-k pairs from the current sliding window and may not be interested in the updates of the results. The queries that require continuous updates of the results are called continuous queries and the queries that compute the results only once are called snapshot queries. In this thesis, we consider both snapshot queries and continuous queries for top-k pairs and objects queries over sliding windows.

1.2.4 Main Memory-based Approach v.s. Disk-based Approach

For data stream processing, we focus on the main memory-based approach for efficient online processing. This is because that the stream data changes frequently and dynamically, and it is infeasible to record the high volume stream data in a disk storage.

In contract, for very large static databases, it is difficult to fit the entire data sets into main memory. However, for the depth-related queries, all of the existing algorithms assume that the data can fit into main memory. This assumption does not always hold because massive data sets have become quite common in almost all disciplines ranging from financial markets to human biology to sociology. Unfortunately, the concept of data depth has not received sufficient attention from the database community. The absence of disk-based algorithms for the depth-related problems opens up a new area that needs to be explored.

Motivated by this, we also propose efficient and I/O optimal disk-based algorithms for solving some important depth-related problems over large data sets.

1.3 Contributions

In this section, we summarize our contributions in this thesis. We proposed efficient techniques for three important problems related top-k queries on spatial and temporal data. For each of these problems, we describe our contributions below.

1.3.1 Top-k Pairs and Objects Queries over Data Streams

In this thesis, we study top-k pairs and objects queries over data streams. Below is a summary.

- To the best of our knowledge, we are first to study a broad class of top-k pairs queries over sliding windows. The server maintains a K-skyband for most recent N objects and we can answer any top-k query over most recent n objects for any $k \leq K$ and any $n \leq N$. We introduce a novel concept of K-staircase to efficiently maintain the K-skyband.

- We provide a generic framework for querying top-k pairs and objects over sliding windows. The proposed framework can handle arbitrary scoring functions, supports queries with any window size and works for out-of-order data streams.

- We provide detailed complexity analysis for our techniques and show that the expected cost of our approach is reasonably close to the lower bound cost. We show that the expected cost of our skyband maintenance algorithm is $O(N \cdot (log(log\ N) + log\ K))$ where $O(N)$ is a lower bound cost for the skyband maintenance. Given a K-skyband, the expected cost of our algorithm to answer a query $Q_{(k,n,s)}$ is $O(log(log\ n) + log\ K + k)$ where $O(k)$ is a lower bound cost for query answering.

- We conduct extensive experiments on both the real and synthetic data sets to demonstrate the efficiency and the scalability of our framework. Our algorithm demonstrates more than three orders of magnitude improvement over a naïve algorithm. Furthermore, we demonstrate its efficiency by comparing it with a specially designed *supreme* algorithm that uses an oracle to conduct certain steps.

1.3.2 Top-k Loyalty Queries

We study a novel query operator, called loyalty queries. We propose a measure named loyalty that reflects how persistently an object satisfies the criteria. Formally, the loyalty of an object is the total time (in past T time units) it satisfied the query criteria. The major contributions are shown below.

- To the best of our knowledge, we are the first to study continuous loyalty queries. In this thesis, we formalize the definition of loyalty queries and present a framework that efficiently solves the loyalty queries.

- We study the problem in a continuous time domain where the updated results are reported as soon as the results change as opposed to the *time-stamp* model where the results are updated after every u time units. Note that the time-stamp model suffers from either high computational cost or low accuracy. More specifically, if u is small, the computation cost increases because the results are to be updated more often. On the other hand, if u is large, the accuracy is reduced because the results may have become invalid between two successive time-stamps. The continuous updates provided by our algorithm do not have these limitations.

- An object issues an update if it starts satisfying the query criteria or if it

stops satisfying the query criteria. Note that the top-k loyal objects may change whenever an object issues an update. Let N be the total number of object updates issued in the last T time units. We prove that our algorithm is optimal by showing that the cost of our algorithm is the lower bound update cost for top-k loyalty queries..

- We theoretically analyse the complexity of our algorithm and prove that it meets the lower bound cost. We also conduct experiments to show the effectiveness and the efficiency of our proposed approach. We compare our algorithm with the Bentley-Ottmann sweep line algorithm [BO79].

1.3.3 Depth-related Queries for Top-k Objects

We study a series of depth-related problems over very large databases in the thesis. Half-plane depth of a plane is the number of objects lying in the plane. Location depth of a point p is the minimum half-plane depth of any plane that is bounded by any line passing through p. The concept of location depth is crucial for ranking the objects in a multi-dimensional space. In this thesis, we study three problems involving location depth: k-depth contour, k-snippet and k-upper envelope query. These queries have a wide range of applications in ranking systems as well as in various other domains such as in outlier detection, clustering, Voronoi diagrams etc. Summarily, we make the following contributions in this work.

- To the best of our knowledge, we are the first to propose disk-based algorithms for a few important depth-related problems that have a wide range of applications in various domains.

- We present two efficient disk-based algorithms named SkyRider and KnightRider. We show that KnightRider algorithm is I/O optimal for k-

upper envelope and k-snippet problems. It is also I/O optimal for k-depth contour when k is smaller than the minimum number of objects in any leaf node of the data structure (e.g., R-tree). Although KnightRider is not I/O optimal for k-depth contour problem when k is large, our experimental results demonstrate that its I/O cost is almost the same as the lower bound cost even when k is very large.

- We extensively evaluate our algorithms on both synthetic and real data sets. The experimental results demonstrate that our algorithms do not only have low I/O cost but are also quite efficient. More specifically, we compare the CPU time of our algorithms with the CPU time of the best known main-memory algorithms [JKN98, EW86] assuming that their algorithms have sufficient main-memory to store the whole data set. The experimental results demonstrate that our algorithms are more than an order of magnitude faster.

1.4 Thesis Organization

This thesis focuses on three fundamental problems in analyzing spatial and temporal data: (1) Top-k Pairs and Objects Queries; (2) Loyalty Queries; and (3) Depth-related Queries. We organize the rest of the thesis as follows. Chapter 2 surveys the existing works related to this thesis. For each problem, we present the preliminaries, problem statement, problem solution and empirical study in an individual chapter. Specifically, Chapter 3 studies the problem of top-k pairs and top-k objects queries over data streams while Chapter 4 studies the problem of top-k and threshold loyalty queries. Chapter 5 studies the depth-related problems for large databases. Finally, Chapter 6 concludes the thesis and proposes feasible future works.

Chapter 2 provides a literature review of the existing works on the three problems. We organize this chapter in three parts. The first part summarizes the related works on top-k pairs and objects queries, while the rest two parts report related works on loyalty queries and depth-related queries, respectively.

Chapter 3 presents our approach to efficiently solve top-k pairs and objects queries over data stream. We first justify the motivations behind our proposed framework. Secondly, we develop efficient algorithms to maintain the minimum candidates, k-skyband, over data stream and to process top-k pairs and objects based on k-skyband. Finally, we evaluate the performance of our proposed algorithms by comparing with a naive approach and a supreme approach.

Chapter 4 describes our approach to efficiently solve loyalty queries over data streams. We first explain the motivations of the loyalty queries. Then, an optimal algorithm is presented to efficiently monitor the results of top-k loyalty queries. A detail analysis is provided to show the optimality of our algorithm. Finally, we evaluate the performance of the proposed algorithm by comparing with a classic sweep line algorithm.

Chapter 5 studies the depth-related problems for top-k queries over large databases. Two algorithms are proposed to solve the problems and one of them is I/O optimal. We show a detail proof of the optimality. Finally, we conduct extensive experiments to demonstrate the efficiency of our proposed algorithms.

Chapter 6 finally summarizes our work in this thesis and provides feasible novel directions regarding our future work.

Chapter 2

Related Work

In this chapter, we present an overview of the related work for each problem we studied in the thesis. More specifically, we firstly introduce the related work on top-k queries in Section 2.1 which is related to all three types of queries. Next, Section 2.2 presents the related work on the sliding window model and continuous queries. An overview of the sweep line algorithms is shown in Section 2.3. Finally, in Section 2.4, we provide a description of the existing techniques on depth-related problems.

2.1 Top-k Queries

First, in Section 2.1.1, we present related work on answering top-k objects queries. Then, in Section 2.1.2, we provide a description of the existing techniques to answer top-k pairs over static databases and data streams.

2.1.1 Top-k Objects Queries

Given a set of objects and a user defined scoring function, a top-k query retrieve the k objects with the smallest scores. The top-k objects queries have been extensively

studied [MBP06, FLN03, NR99]. See [IBS08] for a comprehensive survey of the top-k query processing techniques. Fagin's algorithm (FA) [FLN03], threshold algorithm (TA) (independently proposed in [FLN03, NR99, GBK00]) and no-random access (NRA) [FLN03] propose some of the top-k processing algorithms that combine multiple ranked lists and return the top-k objects.

A major problem with FA is that it uses unbounded buffer (i.e., the number of objects stored in the main memory may be arbitrarily large). On the other hand, the buffer size of TA is $O(k)$. Moreover, TA is optimal in number of objects accessed from the ranked lists. NRA algorithm is applicable to the case when the ranked lists can only be accessed in sorted order (i.e., the objects cannot be accessed using a random access). Mamoulis et al. [MYCC07] present several interesting observations and propose an algorithm LARA that significantly improves the performance of NRA.

Processing the top-k objects queries and k nearest neighbor queries [MBP06, BOPY07, DGKS07] on the data stream has received significant attention. Mouratidis et al. [MBP06] propose an efficient technique to compute top-k objects queries over sliding windows. They make an interesting observation that a top-k objects query can be answered from a small subset of the objects called k-skyband [PTFS05]. Our algorithm is similar in the sense that we also maintain the K-skyband to answer the top-k queries. However, we use a single K-skyband to answer multiple queries having different values of $k \leq K$ and different sizes of the sliding windows. Also, the previous techniques [MBP06, BOPY07] to maintain K-skyband are not applicable to our problem because the techniques rely on the fact that the newly arrived objects cannot be dominated by any of the existing objects. Hence, these techniques unconditionally include the newly arrived objects in the K-skyband. We remark that this observation does not hold for out-of-order

data streams which renders the existing techniques invalid for out-of-order streams. Furthermore, in our problem, even for the in-order streams, the newly formed pairs may or may not be dominated by the existing pairs, which make the request of online maintenance technically more challenging.

2.1.2 Top-k Pairs Queries

The database community has devoted significant research attention to the processing of k-closest pairs queries[HS98, CMTV00, YL02] and their variants [UMY07, AP05]. See [Smi97] for a nice survey that covers the research conducted by computational geometric community.

All of the above mentioned techniques are applicable only to the k-closest pairs queries or their variants. Cheema et al. [CLW+11] propose a unified framework to efficiently answer a broad class of the top-k pairs queries including the queries mentioned above. k-closest pairs queries on moving objects are studied in [ZZS+05, UMY07]. However, the extension of these techniques to answer k-closest (or top-k) pairs queries over sliding windows is either non-trivial or inefficient.

Cheema et al. [CLW+11] first generalize k-closest pairs problem to a top-k pairs problem. A top-k pairs query returns k pairs with the smallest scores where the score of each pair is computed by using a user specified scoring function. A unified approach is presented to answer a broad class of top-k pairs queries including the k closest pairs queries, the k furthest pairs queries and their variants. A detailed complexity analysis is presented to show that the expected performance of the proposed algorithms is optimal when the scoring functions involve less than three attributes. In this work, each dataset is indexed by an R-tree and a priority queue is used to store the intermediate entry pairs. While the proposed solution has a nice feature that it returns the pairs incrementally, its priority queue size may be

prohibitively large. For this reason, a part of the priority queue is kept in main memory and remaining elements are stored in secondary memory as a number of linked lists.

Hjaltason et al. [HS98] propose incremental distance joins where two data sets are joined and the closest pairs are returned to the users in incremental order of the distances between them. Corral et al. [CMTV00] propose several algorithms for k-closest pairs queries. Both of these algorithms index the data sets by R-trees and provide several pruning strategies to improve the performance. Yang et al. [YL02] propose a data structure to further improve the performance of k-closest pairs queries. However, their algorithm works for the case when all the pairs have unique distances [SZS03]. Several variants of k closest pairs queries have also been studied in [UMY07, AP05, SZS03].

Bohm et al. [BOPY07] maintain skyband for monitoring nearest neighbor queries. Techniques are presented for $k = 1$ and no optimization is considered for maintaining k-skyband. Das et al. [DGKS07] propose a technique for answering ad-hoc top-k query in data streams by using a linear scoring function. To the best of our knowledge, there does not exit any previous work to answer top-k pairs queries over sliding windows. Next, we describe previous work related to the top-k pairs query in conventional databases.

2.2 Continuous Queries over Sliding Windows

In Section 2.2.1, we review the techniques on querying data streams over sliding windows. Section 2.2.2 shows the related techniques on continuous queries for spatial and temporal data.

2.2.1 Queries Over Sliding Windows

Processing aggregate queries on data stream [LMT+05, ZKOS05, NNRS08, YLz+07, WRG+06, BDMM04, TZZ06] has been extensively studied. Li et al. [LMT+05] propose an efficient algorithm to compute aggregate queries over sliding windows. We may perform an aggregate query to count the occurrences of the query results over sliding windows in the discrete time domain. However, there are some disadvantages of using the time-stamp model in a discrete time domain for loyalty queries in Chapter 4. We discuss the reason below. The streaming data in the discrete time domain is usually retrieved by sampling the physical world every every u time units. If u is too small, then the size of data to be processed is large, which will affect the efficiency of the algorithm. If we choose a large u, the accuracy cannot be guaranteed because some value changes are lost in the processing. Therefore, it is either imprecise or inefficient to perform aggregate queries to find the loyal objects by sampling in the discrete time domain.

Alternatively, we may process the data stream of the result changes from the traditional queries. However, this involves the current time as an attribute in the aggregation operator, which makes the aggregation results (loyalties) are changeable from time to time. Therefore, a data stream processor has to monitor the aggregated values at every moment, which actually incurs enormous overhead. We remark that most algorithms [GBÖ06, LMT+05, ZKOS05, NNRS08] for computing aggregation over data streams do not specifically consider this point, and thus are not able to efficiently support loyalty queries. Additionally, the objects should be further ranked by their loyalties for processing top-k loyalty queries. This is also non-trivial to be implemented in a data stream processor due to the changeable loyalty values, while our focus is on efficiently processing the continuous updates and detecting loyalty query results in the continuous time domain.

2.2.2 Continuous Spatial and Temporal Queries

The database community has devoted significant research attention to continuously processing spatial and temporal queries [GL04, GWYL06, CBL+10, LYH04, XMA05, BOPY07, MXA04, SCL+12b, MBP06]. The difference between traditional continuous queries and our queries is that the traditional continuous queries return the query results at each timestamp, while our queries return the objects which appear in query results for a majority of the recent time. Continuous spatial and temporal queries such as the continuous range queries [GL04, GWYL06, CBL+10] and the k-nearest neighbour queries [LYH04, XMA05, BOPY07] are well studied. Mokbel et al.[MXA04] present an incremental evaluation paradigm for continuous queries in spatial and temporal databases and its variant [XMA05] can be used to solve continuous k-nearest neighbour queries. We argue the existing work of continuous queries can be used as the front end in our framework in Chapter 4. An example of the application is that loyalty queries can be used as filters to eliminate the noisy (low loyalty) results from continuous queries. The details will be shown in Chapter 4. Farrell et al. [FRC11] present a system to process continuous range queries considering spatiotemporal tolerance. However, their scheme is different from ours. We remark that the loyalty queries may help users discover interesting motion patterns based on a large number of existing techniques.

Besides the continuous queries, the spatial and temporal queries over historical data have also been extensively studied. Li et al [LYL10] present a solution to find top-k objects on temporal data. They use a B-tree based indexing structure for the historical data. Top-k objects are efficiently answered based on the index. Tao et al [TPP05] study the problem of processing spatial-temporal window aggregation queries over historical data. However, such offline algorithms [LYL10, TPP05] cannot be utilized to efficiently solve our problem because the loyalty queries report

the results on the fly and it is not efficient to build the index for online processing.

2.3 Sweep Line Algorithm

In Section 2.3.1, we present a classic sweep line algorithms, the Bentley-Ottmann algorithm. Next, in Section 2.3.2, we give an overview of the related techniques for kinetic data structures.

2.3.1 Bentley-Ottmann Algorithm

The Bentley-Ottmann algorithm is a sweep line algorithm for reporting all intersections between all line segments in the plane. The algorithm is initially proposed by Bentley and Ottmann [BO79] and discussed in detail by Preparata and Shamos [PS85]. Consider a vertical sweep line, first placed at the extreme left of the plane. Then, it will move to the right by jumping between endpoints of the line segments and intersections. The algorithm maintains the vertical ordering of the line segments intersecting the vertical line. An event is created when two adjacent line segments on the vertical line will possibly intersect in the future, namely, the two line segments will possibly exchange their vertical ordering. An event queue is organized for processing the future events. The Bentley-Ottmann algorithm can be used to retrieve the top-k loyalty objects as it always keeps the total ordering of the line segments. Our proposed algorithm also uses a sweep line approach. However, we create less events. The total cost of the Bentley-Ottmann algorithm is $O((N + M) \log N)$, where N is the number of line segments and M is the number of events (intersections). In the worst case, M can be $O(N^2)$. We improve the complexity of solving the top-k loyalty queries and the total cost of our proposed algorithm is $O(N \log N)$.

2.3.2 Kinetic Data Structure

For $k = 1$, the top-k loyalty query is equivalent to finding the upper envelope [Her89] of N line segments in the plane. The upper envelope computation can be done in $O(N \log N)$. Kinetic data structures [BGH99, RKGG07] can also be used to find the upper envelop in a sweep-line fashion. However, it is non-trivial to extend the existing variants of kinetic data structures such as the kinetic heap or the kinetic tournament to support the top-k objects queries (k-level arrangements [EW86, AS98]). Moreover, our proposed algorithm is theoretically more efficient even when $k = 1$. As it is necessary to maintain a priority queue for scheduling the events in the continuous time domain for these data structures, the total cost of the kinetic heap is $O(N \log^3 N)$ and the total cost of the kinetic tournament is $O(N \log^2 N)$, where N is the number of line segments to process. We remark that these techniques for finding the upper envelope can only handle the case when $k = 1$ and are not applicable for $k > 1$.

2.4 Depth-related Problems

Because of the wide range of applications, the depth related problems have received significant attention by the community of statistics and computational geometry.

Statistics Community

The term location depth was first introduced by Tukey who showed that k-depth contours can be used for data picturization [Tuk74]. The significance of data depth in multivariate analysis was further elaborated in various studies [Bar76, LPS99, Liu90, KZ10, Tuk77]. Ruts and Rousseeuw developed a series

of algorithms (ISODEPTH [RR96], HALFMED [RR98] and BAGPLOT [RRT99]) to compute the depth contours.

However, All of these algorithms assume that the data can fit into main-memory. Moreover, their main purpose is to compute the depth of all points, which makes their algorithms computationally difficult over very large database.

Computational Geometry Community

k-depth contour has also received significant research attention [CSY84, MRR$^+$01, KMV02] from the computational geometry community. Cole *et al.* [CSY84] proposed an algorithm to compute k-depth contour in 2-dimensional space. Later, the problem was solved in dual space using similar basic ideas [EW86]. In the following decades, significant research attention was put to improve the complexity of the problem (see [AS98] for a nice survey). Miller et al. [MRR$^+$01] proposed an algorithm to compute all depth contours using topological sweep of the dual arrangement of lines. Krishnan *et al.* [KMV02] introduced a hardware-assisted algorithm with approximate solution. research

Others

Inspired by the usefulness of k-depth contour in outlier detection, Johnson *et al.* [JKN98] proposed a main-memory algorithm to compute k-depth contours. They demonstrated that their proposed algorithm outperforms the existing algorithms. Surprisingly, the problem of depth contours did not receive sufficient attention from the database community. Although several papers touched similar concepts and contain the flavor of data depth, the focus has been different. There is no work solving the depth contour queries over large databases. Böhm and Kriegel [BK01] proposed an I/O efficient algorithm for computing convex hull which

is a special case of our problem where $k = 1$. We remark that solving k-depth contour is considerably more challenging and it is non-trivial to extend their techniques to solve this problem. Xin *et al.* [XCH06] proposed indexing schemes using the concept of location depth to efficiently answer top-k queries involving linear score functions. However, their focus is on building an index and they did not discuss any disk-based algorithm to compute the location depths. In [DGKS07, YAY12], k-level arrangement has been used to answer the top-k queries involving linear scoring functions. However, they use the existing techniques to compute k-level arrangements.

Chapter 3

Continuous Top-k Pairs and Objects Queries

In this chapter, we study a generalized version of the top-k pairs and objects query over data streams. We provide a unified framework to answer a broad class of top-k queries over sliding windows including top-k pairs queries and top-k objects queries. This chapter is based on our research reported in [SCL+12c].

3.1 Overview

Given a scoring function $s(o_i)$ that computes the score of an object o_i, a top-k objects query returns k objects with the smallest scores. Given a scoring function $s(o_i, o_j)$ that computes the score of a pair of objects (o_i, o_j), a top-k pairs query returns k pairs with the smallest scores among all possible pairs of objects. k closest pairs queries, k furthest pairs queries and their variants are some well studied examples of top-k pairs queries that rank the pairs on distance functions.

Due to the importance of the top-k queries, numerous algorithms have been proposed to answer several variants of the top-k objects and top-k pairs

queries [CLW+11, HS98, CMTV00, Smi97, YL02, MBP06]. Our focus is on developing efficient techniques for top-k queries over sliding windows. Top-k objects queries over sliding windows have many applications and have received significant research attention in the past few years [MP07b, MBP06, BOPY07]. However, Top-k pairs query over sliding windows has not been studied well. Therefore, our main focus in this chapter is on presenting the techniques for top-k pairs queries. Then, we show that the framework can be used to answer top-k objects queries.

Top-k pairs queries have many interesting applications in different areas such as wireless sensor network, stock market, traffic monitoring and internet applications etc. For instance, top-k pairs queries can be used for *pair-trading* [Vid04]. Pair-trading is a market neutral strategy according to which two correlated stocks that follow same day-to-day price movement (e.g., Coca-Cola and Pepsi) may be used to earn profit when the correlation between them weakens, i.e., one stock goes up and the other goes down. The profit can be earned by buying the underperforming stock and selling it when the divergence between the two stocks returns to normal. A top-k pairs query can be issued to obtain the pairs of stocks that are correlated (e.g., they belong to the same business sector and have similar fundamentals such as market caps, dividends etc.) and display different trends. Pair-trading can be profitable only if the trader is the first one to capitalize on the opportunity [Vid04]. Hence, the trader may want to continuously monitor the top-k pairs from the most recent data (e.g., a sliding window containing most recent n items).

Consider another example of an online auction website. A user may be interested in finding the pairs of products that have similar specifications but are sold at very different prices (i.e., different final bids). Such pairs may be used to understand the users behavior and market trends, e.g., suitable bidding time for buyers and suitable bidding closing time for sellers etc. An analyst or a user may issue

the following query to obtain top-k pairs of such products sold during last 7 days.

```
Q1:
Select a.id, b.id from auction a, auction b
where a.id < b.id
order by dist(a.spec,b.spec) - |a.bid - b.bid|
limit k
window [7 days]
```

Here $dist(a.spec, b.spec)$ computes the distance (or difference) between their specifications and $|a.bid - b.bid|$ denotes the absolute difference between the final bids they receive. Note that the query prefers the pairs of products that have small difference between their specifications but have large difference between their selling prices. The condition $a.id < b.id$ ensures that a pair (a, b) is not repeated as (b, a).

While the above example shows a simple scoring function, in real-world applications, the users may specify a more sophisticated scoring function. Our framework allows the users to define arbitrarily complex scoring functions. A query that retrieves top-k pairs among the most recent n data items (i.e., sliding window of size n) and uses the scoring function s is denoted as $Q_{(k,n,s)}$.

Our framework that handles top-k pairs queries has the following features.

Unified framework. To the best of our knowledge, we are the first to study top-k pairs queries over sliding windows. We present a unified framework that efficiently solves the top-k pairs queries involving *any* arbitrarily complex scoring function. In our framework, the server maintains N most recent objects where N indicates the size of the largest sliding window any query is allowed to use. Each object has D attributes and the users may define any scoring function that uses $d \leq D$ of these attributes to compute the scores. Our framework handles multiple top-k pairs

queries where each query is allowed to use a different scoring function, a different size of sliding window $n \leq N$ and a different value of k.

Intuitively, it may be possible to improve the performance if the scoring functions satisfy certain properties. We propose optimizations to significantly enhance the performance for a broad class of scoring functions called *global scoring functions* [CLW+11]. We remark that k-closest pairs queries, k-furthest pairs queries and their variants are among many of the popular queries that use the global scoring functions.

Low storage requirement. Our system uses $O(ND)$ space to maintain the most recent N objects. The system may receive different queries (issued by a single user or different users) and several queries having different values of k and n may share the same scoring function. For each unique scoring function, our system maintains a small subset of candidate pairs called *K-skyband* (to be formally introduced in Section 3.3). All the queries that use this scoring function are answered using only the pairs in the K-skyband. We show that the expected size of the K-skyband is $O(K \log(N/K))$ where K is the maximum value of k of the queries that use this scoring function and N is the size of the largest sliding window any query is allowed to use. Hence, in addition to $O(ND)$ memory space, our system uses $O(K \log(N/K))$ memory for each unique scoring function. Note that the total number of possible pairs is $O(N^2)$ and $O(K \log(N/K))$ is much smaller. Later, we show that $O(ND)$ is the lower bound storage requirement (see Theorem 4).

Efficient skyband maintenance. As the new objects arrive and the old objects expire, the skyband is needed to be maintained. Based on a novel concept of K-staircase, we present efficient techniques to maintain the K-skyband. We show that $O(N)$ is a lower bound cost for maintaining the K-skyband for arbitrarily complex scoring functions or when the system is unaware of the properties of the scoring

functions. For this case, the expected cost of our algorithm is $O(N(\log(\log N) + \log K))$ which is reasonably close to the lower bound cost. Note that, in practice, K is usually small (e.g., less than 1000) and $\log(\log N)$ is less than 2 even for a very large value of N (e.g., $N = 10^{99}$).

Efficient query answering. We propose efficient techniques to answer the top-k pairs queries using the K-skyband. Given a K-skyband, the complexity of our technique to answer a top-k pairs query is $O(\log |SKB| + k)$ in the worst case where $|SKB|$ is the size of the K-skyband. The expected cost of our technique is $O(\log(\log n) + \log K + k)$ where n is the size of the sliding window used by the query and K is the largest value of k any query may use. Note that the lower bound cost for query answering is $O(k)$ and the expected cost of our algorithm is reasonably close.

Support for top-k objects queries. We present techniques for answering top-k objects queries over sliding windows (Section 3.6.2). In contrast to the existing techniques, our framework allows arbitrarily complex scoring functions, supports out-of-order data streams and can answer top-k objects queries involving any value of k and n such that $k \leq K$ and $n \leq N$. The experimental results demonstrate the superiority of our algorithm over the state-of-the-art algorithm [MBP06] in terms of running time as well as memory consumption.

Handling out-of-order streams. In Section 3.6.1, we show that our proposed techniques for top-k pairs queries can be applied on out-of-order data streams. The experimental results demonstrate that the performance of our algorithm is better for the out-of-order streams as compared to that of in-order streams.

Batch query processing. We present a new batch processing algorithm that computes the results of multiple top-k queries in a batch (Section 3.6.3). The amortized cost of the algorithm meets the lower bound cost $O(k)$ when the number

of queries is larger than the number of elements in the K-skyband.

Support for chromatic queries. We show that our techniques can handle both *chromatic* and *non-chromatic* variants of top-k pairs queries [CLW+11]. For more details, see Section 3.6.4.

Extensive evaluation and analysis. As discussed above, we conduct a detailed complexity analysis to evaluate our algorithms and demonstrate that the cost of our proposed approach is reasonably close to the lower bound cost. To experimentally verify this, we design an algorithm called *supreme* algorithm that assumes the existence of an oracle that can conduct certain calculations without requiring any computation time. The usage of oracle allows the supreme algorithm to meet the lower bound. Our extensive experiments on real and synthetic data demonstrate that our algorithm performs reasonably well as compared to the supreme algorithm and is more than three orders of magnitude faster than a naïve algorithm.

The rest of this chapter is organized as follows. Section 2.1 presents the related work. We formally define the problem and present a solution overview in Section 3.3. Our system consists of three modules and these modules are presented in Section 3.4 (query answering module), Section 3.5 (skyband maintenance module). Section 3.7 presents our experiment results followed by the conclusion of this chapter in Section 3.8.

3.2 Preliminaries

Sliding windows. Consider a stream of objects. For a fixed number N, a *count-based* sliding window contains the most recent N objects of the data stream. Similarly, for a fixed value T, a *time-based* sliding window contains the objects that arrive within last T time units. Note that the objects defined here may refer to the

different instances of an object in different time in the real world application (e.g. pair-trading). For the ease of presentation, in the rest of the chapter, we consider only the count-based windows. However, our techniques can also be applied to answer the top-k pairs queries over the time-based sliding windows.

Age of a pair of objects. Let o be the i^{th} most recent object. We say that the age of the object o is i and we denote the age of an object as $o.age$. Note that a sliding window of size N consists of every object o for which $o.age \leq N$. We say that an object o has been expired if $o.age > N$.

A pair of objects (o_i, o_j) expires if at least one of the objects o_i and o_j expire. Note that the age of a pair (o_i, o_j) is $max(o_i.age, o_j.age)$. For the simplicity of the notations, we denote the age of a pair p as $p.age$. A sliding window of size N contains every pair p for which $p.age \leq N$.

Score of a pair. Given a scoring function $s(\cdot, \cdot)$, the score of a pair (o_i, o_j) is $s(o_i, o_j)$. For the simplicity of notations, the score of a pair p is denoted as $p.score$.

Top-k pairs query. A top-k pairs query $Q_{(k,n,s)}$ takes three parameters k, n and s and considers a set of pairs P that consists of every pair x for which $x.age \leq n$. The query $Q_{(k,n,s)}$ returns an answer set from P that consists of k pairs such that for every pair p in the answer set and for any other pair $p' \in P$, $p.score \leq p'.score$ (the scores are computed using the scoring function s).

Snapshot vs continuous queries. Note that the set of objects in the sliding window changes dynamically as the new objects arrive and the old objects expire from the sliding window. Hence, some users may be interested in continuous update of the results. In contrast, some users may only be interested in retrieving the top-k pairs from the current sliding window. The queries that require continuous updates of the results are called continuous queries and the queries that compute the results only once are called snapshot queries.

3.3 Solution Overview

Before we present our framework, we revisit the concept of K-skyband [PTFS05]. Then, we prove that K-skyband is the minimal set of pairs required to be maintained in order to answer top-k pairs queries.

K-Skyband. Let x and y be two points in d dimensional space. For any point x, $x[i]$ denotes the value of x in i^{th} dimension. A point x dominates a point y if for every dimension i, $x[i] \leq y[i]$ and for at least one dimension j, $x[j] < y[j]$. Given a set of points P, a K-skyband consists of every point $x \in P$ that is dominated by at most $(K-1)$ other points of P.

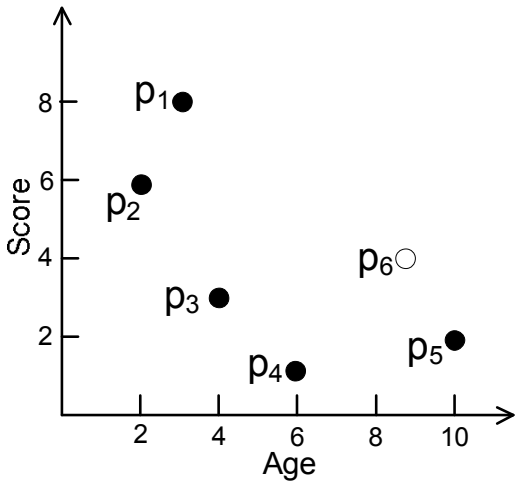


Figure 3.1: K-skyband (K=2)

Consider the example of Figure 3.1 that shows six points p_1 to p_6 in a two-dimensional space. The point p_6 is dominated by two points p_3 and p_4. Hence, the K-skyband (K=2) does not contain the point p_6. The 2-skyband consists of the points p_1, p_2, p_3, p_4 and p_5 because each of these points is dominated by at most one other point.

Given a pair of objects $p = (o_i, o_j)$ and a scoring function s, the pair can be mapped to a two dimensional age-score space where score is $p.score = s(o_i, o_j)$ and

age is $p.age = max(o_i.age, o_j.age)$. Figure 3.1 shows six pairs of objects shown in the age-score space.

THEOREM 1 : Let P be the set of all possible pairs of most recent N objects and each pair be mapped to the age-score space using a scoring function s_1. Let $SKB_{(K,s_1)}$ be the K-skyband of P in the age-score space. Every top-k pairs query $Q_{(k,n,s_1)}$ can be answered using the pairs in $SKB_{(K,s_1)}$ if $k \leq K$ and $n \leq N$.

PROOF. It is sufficient to show that a pair $p' \notin SKB_{(K,s_1)}$ cannot be among the top-k pairs of any query $Q_{(k,n,s_1)}$. Since p' is not in the K-skyband, it implies that there are at least K other pairs such that for each such pair p, $p.score \leq p'.score$ and $p.age \leq p'.age$. Hence, for any sliding window of size $n \leq N$ that contains p', there exist at least K other pairs that are in the sliding window and have scores at most equal to the score of p'. Hence, such top-k pairs query can be answered without considering p'. □

Consider the example of Figure 3.1. Any top-k pairs query $Q_{(k,n,s)}$ can be answered by considering only the pairs p_1 to p_5 where $k \leq (K = 2)$, $n \leq (N = 10)$ and s is the scoring function used to map the pairs to the age-score space. The next theorem shows that the K-skyband is a minimal set of pairs required to be maintained in order to guarantee the correctness.

THEOREM 2 : Let $SKB_{(K,s_1)}$ be the K-skyband as defined in Theorem 1. For any algorithm that does not maintain a pair $p \in SKB_{(K,s_1)}$, there exists a query $Q_{(k,n,s_1)}$ that cannot be answered correctly.

PROOF. Consider a query $Q_{(K,p.age,s_1)}$ (i.e., $k = K$ and $n = p.age$). Since p is a pair in the K-skyband, there exist at most $(K-1)$ other pairs with age at most equal to $p.age$ and scores smaller than $p.score$. In other words, there are at most

$(K-1)$ pairs in the sliding window of size $n = p.age$ that have scores smaller than $p.score$. Hence, p must be among the top-K pairs of the query[1]. □

3.3.1 Expected size of K-skyband

Existing analysis to estimate the expected size of K-skyband (e.g., [ZLZ+09]) assumes that i) the values of objects in one dimension are independent of their values in the other dimensions and ii) the values of the objects on each dimension are unique. Unfortunately, the existing analysis cannot be directly applied to our problem because the second assumption does not hold in our problem settings. This is because many pairs have the same value on the age dimension (i.e., have the same age). Nevertheless, we conduct an analysis and show that the expected size of the K-skyband we need to maintain is $O(K \log(N/K))$.

We assume that the scores of pairs are independent of their ages. This is a reasonable assumption for the scoring functions that do not use ages of the objects to determine the scores of pairs.

LEMMA 1 : Let p be a pair with age x. Assuming that the scores of pairs are independent of their ages, the probability that p is in K-skyband is $min(K/x^2, 1)$.

PROOF. Consider an object o_i and assume that $o_i.age = x$. Every pair (o_i, o_j) for which $o_j.age < o_i.age$ has age equal to $o_i.age$. Hence, the number of pairs with age equal to x is $(x-1)$. Also, for any pair p with $p.age = x$, the number of pairs that have age less than x is $1 + 2 + \cdots + (x-2) = O(x^2)$. Let p' be one of these $O(x^2)$ pairs. Note that the pair p is dominated by p' iff $p'.score \leq p.score$. Hence, the probability that a pair with age x is not dominated by any other pair in the

[1]Note that the proof assumes that there does not exist any other pair p' for which $p'.age = p.age$ and $p'.score = p.score$. We remark that even if such pairs exist, we can easily handle this case by assuming that the score of one pair is slightly larger (larger by an infinitely small value) than the other based on some criteria such as the IDs of the objects in the pairs.

sliding window is $1/x^2$ assuming that every pair is equally probable to have the smallest score. Similarly, the probability that a pair with age x is dominated by at most K other pairs is $min(K/x^2, 1)$. □

THEOREM 3 : Assuming that the scores of pairs are independent to the ages of the pairs, the expected size of the K-skyband is $O(K \log(N/K))$.

PROOF. From Lemma 1, the probability that a pair p with age x is dominated by at most K other pairs is $min(K/x^2, 1)$. As stated in the proof of Lemma 1, the number of pairs with age equal to x is $(x-1)$. Hence, the expected number of pairs that have age equal to x and are in K-skyband is $(x-1) \times min(K/x^2, 1)$. The expected total number of pairs that are in K-skyband is approximately $\sum_{x=2}^{N} x \cdot min(K/x^2, 1)$. Let $y = \lfloor\sqrt{K}\rfloor$. This expression can be simplified as follows.

$$\begin{aligned}\sum_{x=2}^{N} min(\frac{K}{x}, x) &\approx \sum_{x=2}^{y} x + \sum_{x=y+1}^{N} \frac{K}{x} \\ &\approx K + K \sum_{x=y+1}^{N} \frac{1}{x} \\ &\approx K + K(H_N - H_y)\end{aligned}$$

where $H_N = \sum_{x=1}^{N} 1/x$ and is called N^{th} harmonic number. For the case when $y = 1$ (i.e., $K < 4$) , the term $\sum_{x=2}^{y} x$ is considered zero and note that this does not affect our complexity analysis.

It is well known that H_N grows almost as fast as natural log of N. More precisely, H_N is known to be (e.g., see [Knu73]) approximately equal to $ln(N) + \gamma$ where $\gamma \approx 0.577$ is *Euler's constant*. Hence, H_N and H_y can be approximated to $ln(N)$ and $ln(y)$, respectively . So, the expected number of pairs in K-skyband is $O(K \cdot (ln(N) - ln(\sqrt{K}))$ or $O(K \log(N/K))$. □

3.3.2 Framework

In real world scenarios, different users have different requirements. Therefore, different users may choose different scoring functions each involving a different set of attributes. Similarly, different users (or even a single user) may issue the top-k pairs queries with different values of k and n. We present a framework that aims to handle all these different queries efficiently. Our framework consists of the following three modules:

1. Stream Manager. Assume that each object has D attributes and every query issued on the system can use $d \leq D$ of these attributes in its scoring function. Moreover, suppose that N is the maximum size of the sliding window any query is allowed to use. The stream manager maintains $(D + 1)$ lists each consisting of N elements. For every $0 < i \leq D$, the i-th list stores the objects sorted in ascending order of i-th attribute values of the objects. The $(D+1)$-th list is sorted in ascending order of the ages of the objects. Clearly, the storage requirement is $O(ND)$. The theorem below shows that this is the minimum amount of storage required to answer the top-k pairs queries.

THEOREM 4 : To answer a top-k pairs query over the sliding window of size N, the lower bound on storage requirement is $O(ND)$ where D is the number of attributes involved in the scoring function.

PROOF. Assume that an object o is deleted such that $o.age \leq N$. Since the values of the newly arrived objects are unknown, a new object o' may arrive in the stream such that $s(o, o')$ is minimum (i.e., the pair (o, o') is one of the top-k pairs). If the object o is deleted from the stream, this pair will not be considered and the system will miss the correct answer. Hence, the object o must not be deleted. Moreover, the system must store all D attribute values of each object because the

scoring function s may involve $d \leq D$ attributes. Hence, the lower bound on the storage requirement is $O(ND)$. □

2. Skyband Maintenance Module. Let $S = \{s_1, \cdots, s_m\}$ be the set of unique scoring functions used by different queries. For each scoring function s_i, the skyband maintenance module maintains a set of skyband pairs $SKB_{(K_i,s_i)}$ where K_i is the maximum value of k for any query that uses the scoring function s_i (see Figure 3.2).

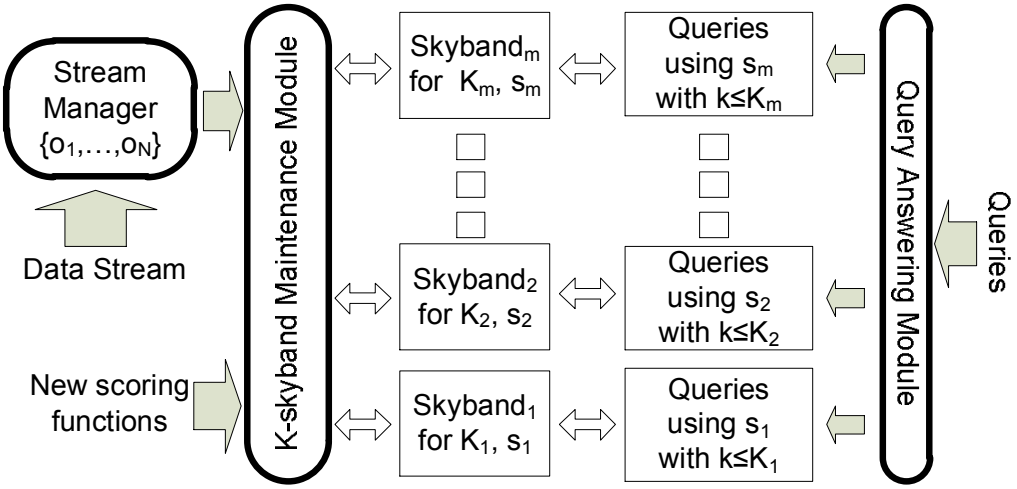


Figure 3.2: Framework

If a user issues a query $Q_{(k,n,s_i)}$ that uses a scoring function s_i not being used by any of the existing queries in the system, the skyband maintenance module creates a new skyband $SKB_{(K_i,s_i)}$ for this new scoring function. Upon receiving the object updates and new queries, the skyband maintenance module updates all the skybands in the system.

3. Query Answering Module. The query answering module is responsible for answering the snapshot or continuous top-k pairs queries. A query $Q_{(k,n,s_i)}$ is answered using the skyband $SKB_{(K_i,s_i)}$.

In Section 3.4, we present the details of the query answering module. The details of the skyband maintenance module is presented in Section 3.5. The techniques

used for stream manager are shown in Section 3.6.4.

3.4 Query Answering Module

In this section, we present our query answering technique. As discussed earlier, to answer a query $Q_{(k,n,s_i)}$, the query answering module uses the skyband $SKB_{(K_i,s_i)}$. For the ease of presentation, we denote K_i as K and $SKB_{(K_i,s_i)}$ as skyband in this section.

3.4.1 Snapshot Top-k Pairs Queries

A straight forward approach to answer a top-k query is to scan the list of skyband pairs in increasing order of their scores. Any pair p for which $p.age > n$ is ignored. The algorithm stops when k pairs with age at most equal to n are retrieved. These k pairs are reported. Note that the cost of this algorithm is $O(|SKB|)$ in the worst case where $|SKB|$ is the size of the K-skyband. Next, we present an approach that answers the top-k pairs query in $O(\log |SKB| + k)$ in the worst case.

To enable efficient computation of the queries, the skyband maintenance module indexes all the K-skyband pairs in a priority search tree (PST) [McC85]. Algorithm 1 shows the PST construction algorithm and Figure 3.4 shows a PST constructed using the pairs in 2-skyband of Figure 3.3. The pairs are labeled such that the age of a pair p_i is i. The number inside each node corresponds to its score. For each node, PST also stores the median value used to split the left and right subtrees (see line 3 of Algorithm 1). For example, the age of root node p_1 is 1, its score is 6 and the left and right subtrees are decided based on the median score 4 (shown under the dotted line).

Before we describe the properties of PST, we define a few terms. Ancestor of

Algorithm 1: PrioritySearchTree(P)

1: **if** P is empty **then** return NULL
2: Choose an element p with smallest age among P
3: $median$ = median of score values of elements in P
4: P_R = {elements in P with score greater than $median$}
5: $P_L = P - P_R - \{p\}$
6: p.right-subtree = PrioritySearchTree(P_R)
7: p.left-subtree = PrioritySearchTree(P_L)
8: return p

a node is its parent or (recursively) the parent of its ancestor. For example, in Figure 3.4, the nodes p_1 and p_2 are the ancestors of the node p_3. Two nodes are called cousins to each other if they have a common ancestor and they do not have a child-ancestor relationship with each other. For example, the nodes p_4 and p_6 are cousins to each other because they have a common ancestor p_1. A node x is called a left cousin of a node y if they share a common ancestor e and x is in the left subtree of e and y is in the right subtree of e. Right cousins are defined similarly. In Figure 3.4, the node p_6 is a left cousin of the node p_4 and the node p_4 is a right cousin of the node p_6.

The priority search tree has the following properties: 1) the age of a node cannot be smaller than the age of its ancestor (e.g., the age of p_3 is larger than the ages of its ancestors p_1 and p_2), 2) the score of a node is always greater than the scores of its left cousins and is always smaller than the scores of its right cousins (e.g., the score of p_6 is greater than the scores of its left cousins (p_7 and p_8) and is smaller than the scores of its right cousins (p_2, p_3 and p_4). Note that the score of a child may be smaller or larger than (or even equal to) the score of its ancestor.

We utilize the above mentioned properties to efficiently answer a top-k pairs

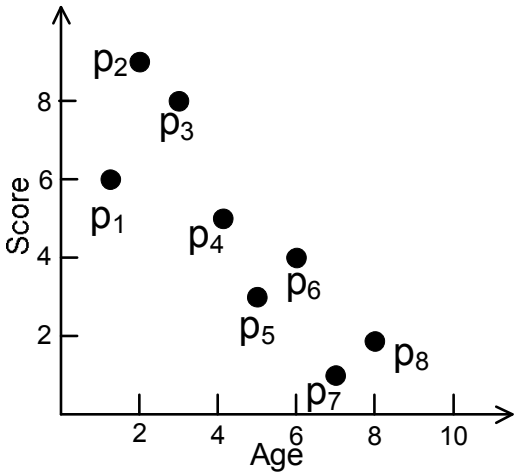


Figure 3.3: 2-skyband

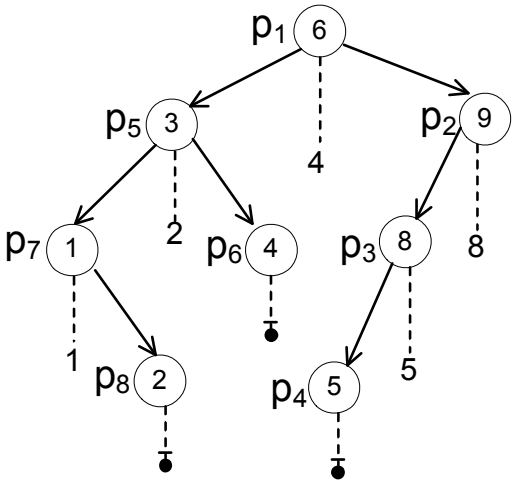


Figure 3.4: Priority Search Tree

query $Q_{(k,n,s)}$. Algorithm 2 shows our query processing algorithm that traverses the PST in an order very similar to the *post-order* traversal. In a post-order traversal, for any node e, its left subtree is visited before its right subtree and the node e is visited in the end. Our algorithm traverses the PST in the post-order except the following two differences: i) it only considers the nodes that lie in the sliding window (see lines 9 and 10) and ii) the algorithm terminates when k objects are visited in the post-order (line 3). It can be proved that the top-k pairs are among the pairs that are either visited or are among the marked nodes in the stack S (line 11). Finally, the set of candidates is scanned and k pairs with the smallest scores are obtained (line 12).

Example 1 : Consider the 2-skyband shown in Figure 3.3 and the PST shown in Figure 3.4. Consider a query that wants to retrieve top-2 pairs in the sliding window of size 7. The post-order traversal returns two nodes p_7 and p_6 and the stack contains the nodes p_1, p_5 and p_2. The nodes p_1 and p_5 are the marked nodes and p_2 is not a marked node. The top-2 pairs are p_7 and p_5 which are selected from the candidates (p_7, p_6, p_1 and p_5). Note that our algorithm does not consider the node p_8 because it does not lie in the sliding window. ■

Algorithm 2: TopPairs(PST,k, n)

```
1: visitedSet = φ
2: if root.age ≤ n then insert root in a stack S
3: while visitedSet.size < k AND S is not empty do
4:    e = top element of S
5:    if e is a leaf OR is marked then
6:       insert e in visitedSet and remove from S
7:    else
8:       mark e
9:       if e.rightChild.age≤ n then push e.rightChild in S
10:      if e.leftChild.age ≤ n then push e.leftChild in S
11: candidates = visitedSet ∪ marked nodes in stack S
12: visit candidates to obtain k pairs with smallest scores
```

Proof of correctness. The algorithm returns k nodes in post-order traversal. Let x be the node with the largest score among these k nodes. Any other node y that has score smaller than $x.score$ must satisfy one of the followings: 1) y is one of the left cousins of x; 2) y is a child of x or 3) y is an ancestor of x. Since our algorithm visits the nodes in post-order, any node that satisfies the condition 1 or 2 is either visited by our algorithm or is not visited because it does not lie in the sliding window (its age is greater than n). Hence, any node that lies in the sliding window and may possibly have score smaller than the score of x is one of its ancestors. Note that the stack contains the unvisited ancestors of all the visited nodes. Moreover, every ancestor of a visited node is a marked node in the stack (see line 8) and our algorithm considers all the marked nodes of the stack (see line 11 of Algorithm 2). Hence, our algorithm correctly determines the top-k pairs.

Complexity analysis. Priority search tree is always a balanced tree [McC85]

because the left subtree and right subtree of a node are determined based on the median score. Therefore, the height of the tree in the worst case is $O(\log |SKB|)$ where $|SKB|$ is the number of pairs stored in PST. Hence, the number of candidates at line 11 of Algorithm 2 is $O(\log |SKB| + k)$. This is because the number of elements in stack at any time is bounded by the height of the tree. To obtain the top-k pairs, we use the the median of medians selection algorithm [BFP$^+$73] to obtain the k pairs in time linear to the number of candidates. Hence the complexity of the algorithm is $O(\log |SKB| + k)$ in the worst case.

As shown earlier, the expected size of K-skyband for a sliding window of size N is $O(K \cdot \log (N/K))$ (Theorem 3). Note that our algorithm does not access a node e and its children if e does not lie in the sliding window of size n. This means that we essentially consider only the pairs in K-skyband that lie in the sliding window of size n. Hence, the expected cost is $O(\log |SKB_n| + k)$ where $|SKB_n|$ is the size of K-skyband for the sliding window of size n. Hence, the expected cost is $O(\log (K \cdot \log (n/K)) + k) = O(\log (\log n) + \log K + k)$. We remark that in the worst case the expected cost is $O(\log (\log N) + \log K + k)$ because the maximum size of the stack in the worst case may still be $O(\log |SKB|)$ even though we ignore the nodes with age greater than n. This is because the PST is a balanced tree with respect to the overall data set and may not necessarily be balanced for a subset of the data.

3.4.2 Continuous Top-k Pairs Queries

The initial results of a continuous top-k pairs query are computed using the algorithm presented earlier for computing the snapshot queries. The results of a query $Q_{(k,n,s)}$ may change if one of the top-k pairs expires or if a new pair has a score smaller than the score of one of the existing top-k pairs. We first handle the expired

pairs and then handle the new pairs.

Handling pairs expired from K-skyband. For each query $Q_{(k,n,s)}$, we maintain two lists of top-k pairs one sorted on their ages and the other sorted on their scores. We use the list of top-k pairs that is sorted on the ages to determine when a pair expires. Let p be an expired pair. We delete p from both of the sorted lists.

Handling new pairs in K-skyband. The skyband maintenance module provides a list of new pairs added to the K-skyband. The list is provided sorted in ascending order of the scores of the new pairs. We scan the list in ascending order and every pair p is added to the answer of the query if $p.score < score_k$ where $score_k$ is the largest score among the scores of the top-k pairs. Whenever such a pair p is added to the answer, the pair with the largest score in the top-k pairs is deleted and the $score_k$ is updated accordingly. The algorithm stops scanning the sorted list when $p.score \geq score_k$. This is because all the remaining pairs are guaranteed to have scores greater than $score_k$ and are not needed to be considered.

Note that after handling the expired pairs and the newly arrived pairs, the answer set of a query may contain less than k pairs (e.g., when the number of deleted pairs is greater than the number of pairs added in the answer set). In such cases, we call Algorithm 2 to compute the top-k pairs from scratch in $O(\log |SKB| + k)$.

Complexity analysis. In the worst case, the complexity of updating the results is $O(\log |SKB| + k)$ because we call Algorithm 2 when the number of deleted pairs is greater than the number of inserted pairs. This worst case may happen only when one or more pairs are deleted from the top-k pairs. We analyse the probability of this case to happen.

For any object o_i, the number of pairs containing o_i in the sliding window of size n is $O(n)$. The total number of possible pairs in sliding window is $O(n^2)$. The probability that any of the pairs related to an object o_i has the smallest score among

all possible pairs is $n/n^2 = 1/n$. The probability that any of the pairs related to the object o_i is one of the top-k pairs is k/n. Hence, the probability that any of the expired pairs is among the top-k pairs is k/n. Therefore, the probability of the worst case to happen is k/n and the expected amortized complexity of updating the results is $O(k/n(\log |SKB| + k))$ per update.

3.5 Skyband Maintenance Module

3.5.1 Handling Arbitrary Scoring Functions

In this section, we present the details of skyband maintenance module (SMM) for arbitrarily complex scoring functions. The K-skyband needs to be updated when an object expires or when a new object arrives. Below, we describe how to handle both of the cases.

Handling when an object expires. Handling an expired object is easy because we only need to delete the relevant pairs from the K-skyband. Note that the age of an expired object o_i is the largest among all the objects in the sliding window. Moreover, every pair that is to be deleted has age equal to $o_i.age$. We keep a list of K-skyband pairs sorted on their ages and for each pair in the list we store a pointer to the relevant node in the PST. We use this list to delete every pair p for which $p.age = o_i.age$.

Handling when an object arrives. When a new object o_i arrives, we may need to update the K-skyband. For arbitrarily complex scoring functions, we need to consider all valid pairs of o_i with the existing objects in the sliding window. The number of pairs to be considered in this case is $O(N)$. Note that $O(N)$ is the lower bound cost for handling a new object because, for arbitrarily complex scoring functions, if we do not consider a pair (o_i, o_j) then we may miss the correct result

because (o_i, o_j) may be one of the top-k pairs.

Algorithm 3: Handling new object (o)

1: Let S be the pairs in K-skyband sorted on scores
2: **for** each new pair p of the object o **do**
3: compute the score and age of p
4: **if** p is not dominated by K-skyband **then**
5: insert p in S in sorted order
6: UpdateSkybandAndStaircase(S)/* `Algorithm 4` */

Algorithm 3 shows the details of handling a newly arrived object o. We say that a pair p is dominated by a K-skyband if there are at least K pairs in the K-skyband that dominate p. For each new pair p, we first need to check whether it is dominated by the existing K-skyband or not (line 4). The pairs that are not dominated by the K-skyband are added to the existing K-skyband which is kept sorted in ascending order of the scores of pairs (line 5). After all the pairs are considered, the algorithm updates the K-skyband (line 6).

As mentioned earlier, for each new pair p, we need to check whether it is dominated by the existing K-skyband or not (line 4). A naïve approach to do so is to consider all the pairs in the existing K-skyband and count the number of pairs that dominate p. If the number of dominating pairs is less than K then the pair p is not dominated by the K-skyband. Note that the complexity of this approach is linear to the size of the K-skyband, i.e., $O(|SKB|)$. Next, we present an approach that checks whether a pair p is dominated by the K-skyband or not in $O(\log |SKB|$. First we introduce the concept of *K-staircase*.

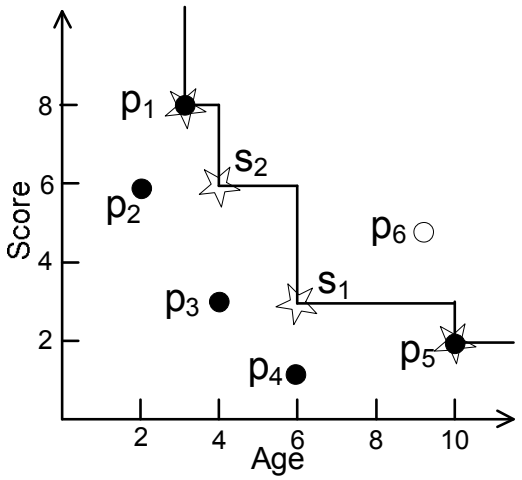


Figure 3.5: 2-staircase

K-staircase

Given a set of points P, the K-staircase is a set of points $SCase$ such that if a point p is dominated by *any* point $x \in SCase$ then there are *at least* K points in P that dominate p. Moreover, for any point p', if there does not exist *any* point $x \in SCase$ that dominates p' then there are *at most* $K-1$ points in P that dominate p. Note that the points in the K-staircase can be used to check whether a point is dominated by the K-skyband or not. More specifically, a point p is dominated by the K-skyband if and only if it is dominated by at least one point of the K-staircase.

Figure 3.5 shows a set of points $P = \{p_1, \cdots, p_6\}$. The K-staircase ($K = 2$) is also shown which consists of the points p_1, p_5, s_1 and s_2 (shown as stars). Note that the points in the staircase are not necessarily the points in the set P (see s_1 and s_2). Before we show our algorithm to compute the K-staircase, we present the intuition.

Consider a point p_3 that is in K-skyband ($K = 2$) as shown in Figure 3.5. Among the points that have scores at most equal to $p_3.score$, we identify a point that has K^{th} smallest age. In Figure 3.5, the points that have scores at most equal

to $p_3.score$ are p_3, p_4 and p_5 and the point p_4 has the K^{th} ($K = 2$) smallest age among these points. Based on p_3 and p_4, we determine a K-staircase point s_1 such that $s_1.score = p_3.score$ and $s_1.age = p_4.age$. Please note that such a point s_1 is dominated by at least K points of P. Hence, any point that is dominated by s_1 is dominated by at least K points of P. Moreover, any point that dominates s_1 is dominated by at most $K - 1$ points of P. To construct K-staircase, we repeat the above procedure for every point of the K-skyband and determine a relevant K-staircase point. Below, we present the details.

Updating K-skyband and K-staircase

Recall that in Algorithm 3, we need to update the K-skyband and K-staircase after all the new pairs are added to the existing K-skyband (see line 6). In this section, we present our technique to efficiently update the K-skyband and K-staircase. In [TPK+03], the authors presented an algorithm to construct the K-skyband from a set of two-dimensional points P. Since our algorithm to construct the K-staircase has a similar structure, we embed the two algorithms to construct both the K-skyband and K-staircase in parallel. If the points in the dataset P are sorted in the ascending order of their scores, the algorithm constructs the K-skyband and K-staircase in $O(|P| \cdot \log K)$ where $|P|$ is the number of points in P.

Algorithm 4 presents the details. The points in P are accessed in ascending order of their scores (if two points have the same score, the point with the smaller age is accessed first). An accessed point p cannot be the K-skyband point if the algorithm has accessed at least K other points with age at most equal to $p.age$ (line 10). This is because all of these K points have scores at most equal to $p.score$ (recall that the points are being accessed in ascending order of scores).

If a point p is in K-skyband then we identify a K-staircase point x such that

Algorithm 4: UpdateSkybandAndStaircase(P)

```
1: Initialize a max-heap H with key set to age of elements
2: Let P be sorted in ascending order of scores
3: for each pair p in P do
4:    if |H| < K then
5:       add p to SKB_K
6:       insert p in H
7:       if |H| = K then
8:          insert (p.score, H.top().age) into K-staircase
9:    else
10:      if p.age ≥ H.top.age then
11:         discard p
12:      else
13:         add p to SKB_K
14:         insert p in H
15:         H.pop()/* delete top element of H */
16:         insert (p.score, H.top().age) into K-staircase
17: output SKB_K and K-staircase.
```

$x.score = p.score$ and $x.age = H.top().age$ where $H.top().age$ is the maximum age of a pair in the heap (line 16). Note that the heap stores K smallest ages and $H.top().age$ corresponds to the K^{th} smallest age among the points that have been accessed (i.e., have scores smaller than $p.score$).

Checking dominance using K-staircase. We say that a point p is dominated by the K-staircase $SCase$ if and only if there exists a point $x \in SCase$ that dominates the point p. As stated earlier, a point p is dominated by K-skyband if and only if p is dominated by the K-staircase. Next, we show that checking whether a point p is dominated by the K-staircase can be done in $O(\log |SKB|)$.

Note that the points of the K-staircase returned by Algorithm 4 are sorted on their scores. To check whether a point p is dominated by the K-staircase or not, we do a binary search on the points in the K-staircase and retrieve a point x that has score smaller than $p.score$ and the point next to x in the K-staircase has score greater than $p.score$. It can be proved that if p is not dominated by x then the point is not dominated by any point in the K-staircase. This is because all the points of the K-staircase that have scores smaller than x have age greater than $x.age$ (see the K-staircase of Figure 3.5). Since the size of K-staircase is bounded by the size of K-skyband, checking whether a point is dominated by K-staircase takes $O(\log |SKB|)$.

Complexity analysis

The following lemma is important in analysing the complexity.

LEMMA 2 : When a new object arrives, the expected number of new pairs that are not dominated by the existing K-skyband is $O(K)$.

PROOF. For a newly arrived object o_{new}, there are $O(N)$ new pairs in the sliding window. Let p_x be a new pair with age equal to x. The set of new pairs is $\{p_2, p_3, \cdots, p_N\}$. From Lemma 1, a pair with age x has probability $min(K/x^2, 1)$ not to be dominated by K-skyband. Hence, $\sum_{x=2}^{N} min(K/x^2, 1)$ gives the number of new pairs that are not dominated by the K-skyband. The summation can be approximated to $\sqrt{K} + K \cdot \sum_{x=\sqrt{K}+1}^{N} 1/x^2$. This is reduced to $\sqrt{K} + K \cdot C$ where C is a constant smaller than $\pi^2/6$ (see Basel's problem[2]). Hence, the number of such pairs is $O(K)$. □

Cost of handling a new object. We analyse the complexity of Algorithm 3.

[2]http://en.wikipedia.org/wiki/Basel_problem

lines 2 to 4: For a newly arrived object, Algorithm 3 considers $O(N)$ new pairs (line 2). For each of these pairs, the algorithm checks whether it is dominated by the K-staircase or not. Hence, the total cost of these lines is $O(N \cdot \log |SKB|)$.

line 5: According to Lemma 2, the number of pairs that are not dominated by the K-skyband is $O(K)$. These $O(K)$ pairs are inserted in the K-skyband set S. The cost of each such operation is logarithmic to the size of S. Hence, the cost of line 5 is $O(K \cdot \log (|SKB| + K))$ where $O(|SKB| + K)$ is the expected size of S after all K pairs are added.

line 6: At line 6, Algorithm 4 is called. The cost of Algorithm 4 to compute the K-skyband and the K-staircase for a sorted dataset of size $|S|$ is $O(|S| \cdot \log K)$ [TPK+03]. Since the size of S is $O(|SKB| + K)$, the cost of computing the K-skyband and the K-staircase (line 6 of Algorithm 3) is $O((|SKB| + K) \cdot \log K)$. After the K-skyband is updated, the new pairs inserted in the K-skyband are inserted in the priority search tree (PST) and the pairs that are not among the K-skyband pairs anymore are deleted from the PST. Since the size of the K-skyband is expected to remain the same before and after the update, the number of new pairs is equal to the number of pairs deleted from the PST, i.e., $O(K)$ according to Lemma 2. The cost of inserting and deleting these pairs from the PST is $O(K \cdot \log |SKB|)$.

Overall cost of Algorithm 3: The above analysis demonstrates that the overall complexity of Algorithm 3 is $O(N \cdot \log |SKB| + K \cdot \log (|SKB| + K) + (|SKB| + K) \cdot \log K)$. Since $|SKB|$ is larger than K and N is larger than $|SKB|$ if $K \ll N$ (which is usually the case), the overall complexity of Algorithm 3 is $O(N \cdot \log (|SKB|))$.

Cost of handling an expired object. When an object o_i expires, the number of pairs that are to be deleted from the K-skyband is at most K. This is because the K-skyband contains at most K pairs that have equal age (the K pairs

with the smallest scores). Recall that each deletion and insertion on PST takes $O(\log |SKB|)$. In the worst case, K pairs are to be deleted and the worst case cost is $O(K \cdot \log |SKB|)$.

Overall cost. Note that the cost of handling a new object dominates the cost of handling an expired object. Hence, the overall cost is $O(N \cdot \log |SKB|)$. Since the expected size of $|SKB|$ is $O(K \cdot \log (N/K))$, the overall expected complexity is $O(N \cdot (\log (\log N) + \log K))$.

3.5.2 Optimization for Certain Scoring Functions

In the previous subsection, we showed that the skyband can be maintained by considering $O(N)$ new pairs when a new object arrives in the data stream. In this section, we show that for a broad class of scoring functions we can reduce the number of considered pairs. We call these scoring functions the *global scoring functions*. The global scoring functions are based on monotonic and loose monotonic functions as defined in [CLW+11] and can be used to model several important queries such as k-closest pairs queries, k-furthest pairs queries and their variants.

we give formal definitions of these functions.

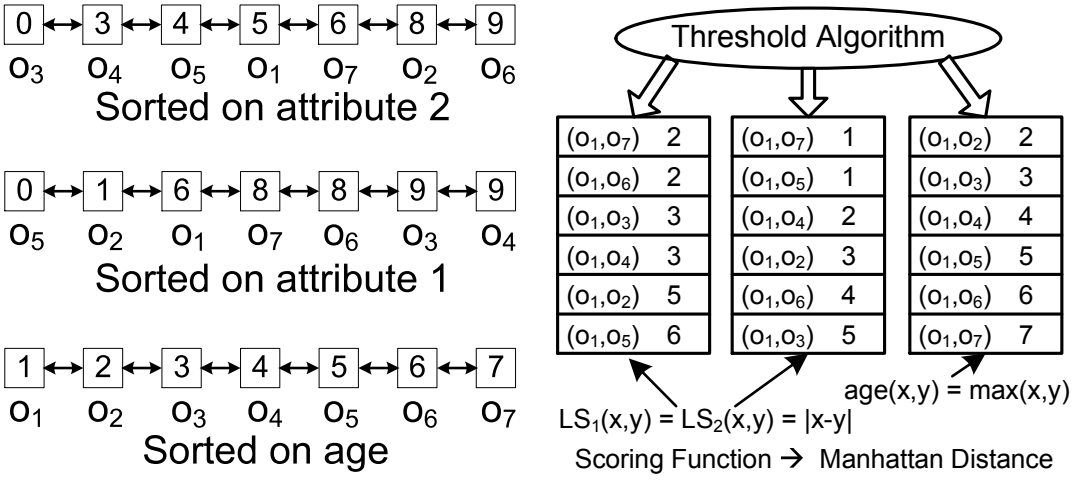
Monotonic function. A function f is called a monotonic function if it satisfies $f(x_1, \cdots, x_n) \leq f(y_1, \cdots, y_n)$ whenever $x_i \leq y_i$ for every $1 \leq i \leq n$.

Loose monotonic scoring function. Let $ls(.,.)$ be a scoring function that takes two values as parameter and returns a score. A function $ls(.,.)$ is a loose monotonic function if for every value x_i both of the following are true: i) for a fixed x_i and every $x_j > x_i$, $ls(x_i, x_j)$ either monotonically increases or monotonically decreases as x_j increases, and ii) for a fixed x_i and every $x_k < x_i$, $s(x_i, x_k)$ either monotonically increases or monotonically decreases as x_k decreases.

Note that the loose monotonic scoring functions are more general than the

monotonic scoring functions, i.e., every monotonic function is a loose monotonic function but the converse may not be true. The absolute difference of two values (e.g., $|x_i - x_j|$) is a loose monotonic function but not a monotonic function. The average of two values is a loose monotonic function as well as a monotonic function.

Global scoring function. Let d be the number of attributes used by the scoring function. For each attribute i, the user specifies a loose monotonic scoring function $ls_i(.,.)$ that computes the score of a pair on the attribute i. Such scoring function is called a local scoring function and the score $ls_i(a,b)$ of a pair (a,b) is called its local score. The users are allowed to define a different local scoring function for each attribute. The user defines a global scoring function gsf that takes d local scores as parameter and returns the final score of a pair (a,b) as $gsf(ls_1(a,b), \cdots, ls_d(a,b))$. We require that such global scoring function must be a monotonic function. Note that the global scoring functions are more general than the monotonic scoring functions used by many real world applications [CLW+11]. For instance, k-closest pairs queries, k-furthest pairs queries and their variants can be answered by using global scoring functions (see [CLW+11] for details).



(a) Sorted Lists (b) Applying TA

Figure 3.6: Optimization for global scoring functions

Technique

Let D be the total number of attributes of the objects. As described in Section 3.2 and shown in Figure 3.6(a), the stream manager maintains $(D + 1)$ sorted lists (D lists each sorted on one of the attributes and one list sorted on the ages). The global score (i.e., final score) of a pair is computed by combining $d \leq D$ local scores where the i-th local score corresponds to the score of a pair on the i-th attribute.

For a newly arrived object o and for an attribute i, we can incrementally retrieve the pairs of objects related to the object o in ascending order of their i-th local scores (see [CLW+11] for details). Figure 3.6(b) shows an example where, for a newly arrived object o_1, the lists can be used to incrementally retrieve the pairs of o_1 in sorted order of the scores. We iteratively retrieve these pairs in ascending order of scores for each attribute i and then apply an algorithm similar to the threshold algorithm (TA) [FLN03] to terminate the algorithm before visiting all $O(N)$ new pairs of the newly arrived object.

Algorithm 5 presents the details. The algorithm accesses the pairs in round-robin fashion from the $d+1$ attributes where the $(d+1)^{th}$ attribute corresponds to the age of a pair (line 4). Each accessed pair p is mapped to age-score space and is inserted in S if it is not dominated by the K-staircase (line 6).

Let $\underline{ls_i}$ be the local score of the last retrieved pair for the i^{th} attribute and $\underline{age}$ be the age of the last pair retrieved for the age attribute. Note that $\underline{ls_i}$ corresponds to the smallest possible local score of any unseen pair for the i^{th} attribute. Hence, $gsf(\underline{ls_1}, \cdots, \underline{ls_d})$ is the smallest possible final score of any unseen pair where $gsf()$ denotes the global scoring function. Similarly, $\underline{age}$ is the smallest possible age of any unseen pair. Hence, we map a dummy point (see line 10) to the age-score space with the smallest possible age and the smallest possible score. If this dummy point is dominated by the K-staircase then any unseen pair will also be dominated

Algorithm 5: handling new object o)

1: S = points in K-skyband sorted on scores
2: dummy point = $(0, 0)$
3: **while** dummy point not dominated by K-staircase **do**
4: **for** $i = 1$ to $i = d + 1$ **do**
5: access next best pair p of o in ascending order of i^{th} local score
6: **if** p is not dominated by K-staircase **then**
7: insert p in S in sorted order of scores
8: Let $\underline{ls_i}$ be the score of last pair seen for i^{th} attribute
9: Let $\underline{age}$ be the age of last pair seen from the age list
10: dummy point = $(\underline{age}, gsf(\underline{ls_1}, \cdots, \underline{ls_d}))$
11: UpdateSkybandAndStaircase(S)

by the K-staircase. For this reason, we do not need to consider remaining unseen pairs (see line 3) if the dummy pair is dominated by the K-staircase.

Complexity analysis

Note that the main difference between Algorithm 3 and Algorithm 5 is that Algorithm 3 considers $O(N)$ new pairs when a new object arrives whereas Algorithm 5 considers fewer pairs by using the threshold algorithm (TA). Let M be the number of the pairs considered by Algorithm 5. We estimate the value of M and obtaining the overall complexity is similar to that of the Algorithm 3.

We access the pairs in round robin fashion for the $d+1$ attributes. Note that the algorithm may terminate if at least K pairs have been seen for each of these $d + 1$ attributes. This is because for any unseen pair there would be at least K pairs that have both the score and age less than it. Fagin showed that the number of elements accessed from the $d+1$ lists in such case is $M = (d+1) \cdot N^{d/(d+1)} \cdot K^{1/(d+1)}$ [FLN03].

3.6 Extensions

3.6.1 Handling Out-of-order Streams

In many real-world applications, the objects do not arrive in correct order due to various reasons such as network delay and data sent from different sources [CKT08, LLG+09]. Such streams are called out-of-order data streams. In out-of-order streams, the age of an object does not denote the time since it has been in the sliding window (i.e., the time since it was received) but it denotes the time since it was sent to the server. Hence, the age of a newly received object may be larger than the age of objects received earlier.

As mentioned in [LLG+09], various stream processing technologies experience significant challenges when faced with out-of-order data streams. Our proposed techniques do not rely on the assumption that the age of a newly received object is the smallest among the existing objects. Hence, all of our proposed techniques can be directly applied on out-of-order data streams. In fact, one optimization is possible in the skyband maintenance module (Algorithm 3). For out-of-order data streams, we update K-skyband and K-staircase (line 6 of Algorithm 3) only if the set S is changed due to insertion of any pair at line 5. Note that for in-order data streams, when an object arrives, there is at least one new pair that has the smallest age among all existing pairs in the sliding window. Hence, S is always updated due to insertion of this new pair and K-skyband and K-staircase is needed to be updated.

Our theoretical analysis and experimental evaluation demonstrate that our proposed techniques perform better for out-of-order streams. This is mainly due to the following reason. As shown in Lemma 3, if an object arrives late (i.e., out-of-order), the new pairs have lesser chance to be in the K-skyband and this results in low

maintenance cost of the K-skyband.

LEMMA 3 : Assume that the age of a newly received object o_{new} is y. The expected number of new pairs that are not dominated by the existing K-skyband is inversely proportional to the value of y.

PROOF. If $y > N$, then every new pairs has age greater than N and can be ignored. Otherwise, there are $O(N)$ new pairs in the sliding window. There are $(y-1)$ pairs with age equal to y (the pairs of o_{new} with every object o that has age smaller than y). The remaining pairs can be denoted as the following set $\{p_{y+1}, p_{y+2}, \cdots, p_N\}$ where p_x denotes that the age of pair p_x is x. From Lemma 1, a pair with age x has probability $min(K/x^2, 1)$ not to be dominated by K-skyband. Hence, $(y-1) \times min(K/y^2, 1) + \sum_{x=y+1}^{N} min(K/x^2, 1)$ is the expected number of new pairs that are not dominated by the K-skyband. Clearly, this value is inversely proportional to the value of y, i.e., the expected number of new pairs that are not dominated by the K-skyband is larger for smaller values of y and vice versa. □

3.6.2 Top-k Objects Queries

Given a scoring function that computes the score of an object, a top-k objects query retrieves k objects with the smallest scores. A top-k objects query over sliding windows considers the objects in the current sliding window (e.g., the most recent n objects) and returns k objects with the smallest scores. Such queries have many applications and have received significant research attention [MP07b, MBP06, BOPY07].

In this section, we present techniques to efficiently handle top-k objects queries that outperform state-of-the-art algorithm [MBP06] in terms of both running time and memory consumption. We remark that our proposed algorithm has the following novel features not considered/supported by the existing algorithms: 1) It

supports arbitrarily complex scoring functions whereas the existing algorithms only support either monotonic functions [MBP06] or kNN queries [MP07b, BOPY07]; 2) It can answer top-k objects queries having any window size $n \leq N$ in contrast to the previous techniques that focus only on $n = N$; 3) Our proposed techniques can handle out-of-order data streams.

Our framework is similar to the framework we presented to answer top-k pairs queries. Specifically, each object is mapped to a score-age space and a K-skyband is maintained by the skyband maintenance module. The query answering module uses the K-skyband to answer snapshot and continuous top-k objects queries for any $k \leq K$ and any $n \leq N$. The query answering module (and its complexity analysis) is exactly the same as presented in Section 3.4. The skyband maintenance module is different and is presented below.

Skyband maintenance for top-k objects

Note that if the data stream is in-order then the newly arrived object has the smallest age and cannot be dominated by any existing object in the sliding window. Hence, the newly arrived object must always be inserted in K-skyband. This observation was exploited in the existing work [MBP06]. In contrast, for out-of-order data streams, a newly arrived object may or may not be dominated by the K-skyband. We present the techniques for out-of-order streams which can be directly applied for in-order streams.

A straightforward approach to maintain K-skyband is to insert the newly arrived object o in the K-skyband (if it is not dominated by K-skyband) and remove every object o' that is dominated by o and $(K-1)$ existing objects. To efficiently check whether the newly arrived object is dominated by the K-skyband, we can use K-staircase. This straightforward approach may be expensive because

it requires updating K-skyband and K-staircase every time a new object arrives. Therefore, we adopt a lazy-update approach shown in Algorithm 6 that updates the K-skyband and K-staircase only if the size of K-skyband increases by a parameter x.

Algorithm 6 computes the score and age of the newly arrived object o and checks whether it is dominated by K-staircase or not (line 3). If o is not dominated, it is inserted in S and the priority search tree (line 4). Let x be a parameter and $|SKB|$ denote the size of K-skyband after the last update of K-skyband and K-staircase (line 7). If after the insertion of o in S, the size of S becomes larger than $x+|SKB|$ then Algorithm 4 is called to update the K-skyband and K-staircase.

Algorithm 6: Handling new object(o)

1: Let S be the objects in K-skyband sorted on scores
2: compute the score and age of o
3: **if** o is not dominated by K-Staircase **then**
4: insert o into S and PST
5: **if** $|S| > x + |SKB|$ **then**
6: UpdateSkybandAndStairCase(S)/* `Algorithm 4` */
7: $|SKB|$ = size of K-skyband

Complexity analysis. Assume that the size of K-skyband is reasonably stable. The cost of line 3 is $O(\log |SKB|)$. The insertion cost of each object into S and PST is at most $O(\log (x + |SKB|))$ (line 4). The cost of line 6 is $O((x+|SKB|) \log K)$ where $x+|SKB|$ is the number of elements in S at the time K-skyband and K-staircase is updated (see the cost of Algorithm 4 presented in Section 3.5.1). Since line 6 is called after at least x new arrivals, the amortized cost of line 6 is $\frac{(x+|SKB|) \log K}{x}$. Hence, the amortized cost of the whole algorithm is $O(\log (x + |SKB|) + \frac{1}{x}(x + |SKB|) \log K)$. Assuming that $x = |SKB|$, the

amortized cost of the algorithm is $O(\log |SKB| + \log K) = O(\log |SKB|)$.

Optimizing the value of x. Intuitively, if we choose larger x, the cost of line 4 increases whereas the amortized cost of line 6 is reduced (and vice versa). Next, we show how to choose an optimal value of x. Let $Cost(x)$ denote the amortized cost of the algorithm for value x.

$$Cost(x) = \log (x + |SKB|) + \frac{(x + |SKB|) \log K}{x} \tag{3.1}$$

To minimize the cost, we take the derivative of $Cost(x)$ and set it equal to 0.

$$\frac{1}{x + |SKB|} - \frac{|SKB| \log K}{x^2} = 0$$

The optimal value of x is then obtained by solving the above equation.

$$x = \frac{|SKB| \times (\log K + \sqrt{\log K(4 + \log K)})}{2}$$

Note that the above analysis is valid for $K > 1$. For a more accurate analysis that is also applicable for $K = 1$, $O(\log K)$ in Eq. (3.1) is to be replaced by $O(C + \log K)$ where C is a constant.

3.6.3 Batch Processing for Multiple Queries

In this section, we present a query processing algorithm that answers multiple snapshot queries in a batch. Suppose $\mathcal{Q}$ is a set of queries that share the same scoring function. For a query $Q_i \in \mathcal{Q}$, k_i and n_i denote the values of k and n used for Q_i, respectively. Algorithm 7 presents the details of the technique.

Recall that the skyband maintenance module stores the pairs in K-skyband in sorted order of scores. The algorithm accesses the pairs in ascending order of scores. For each accessed pair p, it accesses the queries in $\mathcal{Q}$ in descending order of the window lengths. Note that if the age of p is greater than the window length n_i

Algorithm 7: Batch(SKB, Q)

```
1: for each pair p in SKB in ascending order of scores do
2:   for each Q_i in descending order of window lengths n_i do
3:     if p.age > n_i then
4:       break;
5:     insert p into Q_i.topK
6:     if |Q_i.topK| = k_i then
7:       report Q_i.topK and remove Q_i from 𝒬
```

of an accessed query then p cannot be an answer to any of the remaining queries (because the age of p would be larger than the window lengths of such queries). This observation is exploited at line 4 of Algorithm 7. If $p.age \leq n_i$ then p is added to a linked list $Q_i.topK$ that stores the answer of Q_i (line 5). If $Q_i.topK$ contains k_i elements then Q_i is removed from $\mathcal{Q}$ and the results of Q_i are reported (line 7). We remark that the algorithm for top-k objects queries is exactly the same except that pair is replaced with object in the pseudocode.

Complexity Analysis. For the sake of simplicity, assume that each query has same value of k. For each pair p accessed at line 1, the condition of line 3 is satisfied at most once. For each query Q_i accessed before this condition is satisfied, p is inserted as an answer in Q_i. Hence, the overall cost of the algorithm is $O(|SKB| + k|\mathcal{Q}|)$ where $|\mathcal{Q}|$ is the total number of queries. Note that when $|\mathcal{Q}|$ is larger than $|SKB|$ the amortized cost for each query is $O(k)$ which meets the lower bound query processing cost.

3.6.4 Handling Chromatic Top-k Pairs Queries

The top-k pairs queries can be classified into chromatic and non-chromatic top-k pairs queries [CLW+11]. Chromatic queries are further classified into *homochro-*

matic and *heterochromatic* queries. Assume that each object in the stream is assigned a color. A homochromatic top-k pairs query returns the top-k pairs among the pairs that contain two objects having the same color. In contrast, a heterochromatic top-k pairs query considers only the pairs that contain two objects with different colors. The top-k pairs queries that consider all the pairs (i.e., the colors are not taken into consideration) are called the non-chromatic queries.

Consider the query Q1 shown in Section 1.2 and assume that a user wants to consider only the pairs of products that were auctioned by different sellers. The user may issue a heterochromatic query by adding a condition `a.seller` $\neq$ `b.seller`. Similarly, a user who wants to consider the pairs of products sold by the same seller may issue a homochromatic query by adding a condition `a.seller=b.seller`. In this section, we propose extension to handle chromatic top-k pairs queries.

Recall that stream manager receives the data stream and maintains the most recent N objects. As showed in Section 3.5.2, the skyband maintenance module can efficiently maintain K-skyband for a broad class of scoring functions if the objects are sorted on their attribute values as well as on their ages. Next, we show that the stream manager can store the objects in sorted order in a way that enables the system to efficiently handle both the non-chromatic and chromatic top-k pairs queries.

For each of D attributes and the age, the stream manager stores three doubly linked lists as shown in Fig. 3.7. Fig. 3.7 shows an example where three lists are shown for the objects sorted on their ages. Some objects are assigned grey color (o_2, o_4 and o_5) and the others are assigned white color (o_1, o_3, o_6). Below, we describe the structure of each of the three lists.

Non-chromatic list. Since non-chromatic queries do not impose any restriction on the colors of the objects in the pair, the non-chromatic list links the adjacent

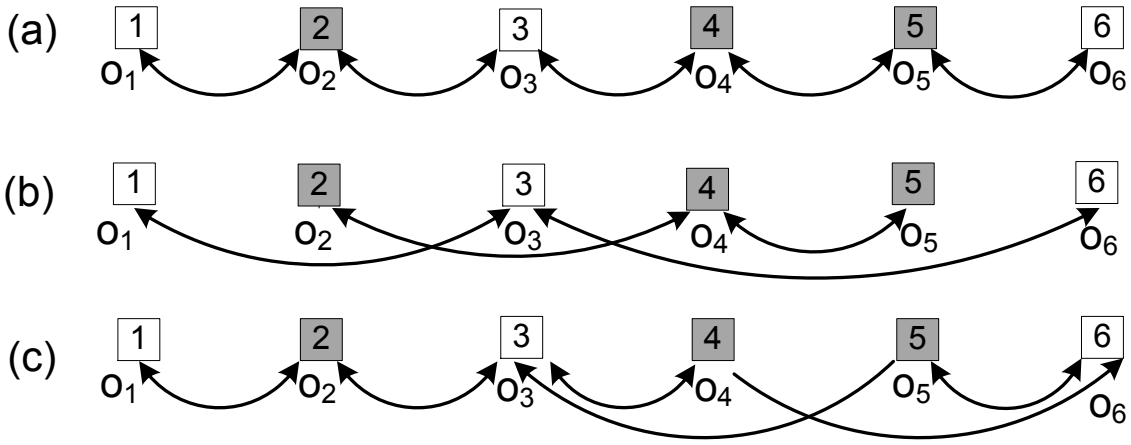


Figure 3.7: Sorted lists (a) non-chromatic (b) heterochromatic (c) homochromatic

objects to each other (see Fig. 3.7 (a)).

Homochromatic list. For any new object o_i, we only need to consider its pairs with the objects that have the same color as that of o_i. Hence, for every object o_i, the homochromatic list provides the links to its adjacent objects (in sorted order) of the same color. For example, in Fig. 3.7 (b), the right adjacent object of o_3 is o_6 and its left adjacent object is o_1.

Heterochromatic list. For every object o_i, the heterochromatic list provides links to its adjacent objects (in sorted order) having different colors. For example, in Fig. 3.7 (c), the right adjacent object of o_4 is o_6 and the left adjacent object of o_4 is o_3. These links are used to access the heterochromatic pairs related to o_i in constant time [CLW$^+$11].

As the new object arrives or the old object expires, the lists and affected links to the adjacent objects are updated. This can be done in $O(\log N)$ for each object update. To answer the chromatic queries, the skyband maintenance module only uses the relevant lists to maintain the K-skyband. For example, to maintain a K-skyband for the homochromatic top-k pairs queries, the skyband maintenance module only uses the homochromatic lists. The query answering module does not

require any change.

3.7 Experiments

First, we present the results for our top-k pairs algorithms and then we provide the results for other techniques.

3.7.1 Top-k Pairs Queries

Experimental settings

Real data. We use a publicly available data set[3] collected from 54 sensor nodes deployed in the Intel research lab in Berkeley between February 28th and April 5th, 2004. Each node measures environment readings such as temperature, humidity and light. The data set consists of 2.3 million records collected from these sensors. We use the following scoring function.

$s(o_x, o_y) = \frac{|o_x.time - o_y.time|}{|o_x.temp - o_y.temp||o_x.humidity - o_y.humidity|}$

The scoring function prefers the pairs of sensor readings that are taken within small duration of time and report quite different temperature and humidity. We remark that we tried several other inherently different scoring functions and the experimental results demonstrated similar trends.

Synthetic data. We generate synthetic data following uniform, correlated and anti-correlated [BKS01] distributions and each data set consists of 2 million objects. Let $o[i]$ be the value of the object o in i^{th} dimension. For a scoring function that uses d dimensions, we use the following four different scoring functions.

$s_1(o_x, o_y) = \sum_{i=1}^{d} |o_x[i] - o_y[i]|$

$s_2(o_x, o_y) = -\sum_{i=1}^{d} |o_x[i] - o_y[i]|$

[3] http://db.csail.mit.edu/labdata/labdata.html

$s_3(o_x, o_y) = \prod_{i=1}^{d} |o_x[i] - o_y[i]|$

$s_4(o_x, o_y) = -\prod_{i=1}^{d} |o_x[i] - o_y[i]|$

Note that the scoring function s_1 retrieves the k-closest pairs and s_2 retrieves the k-furthest pairs according to the Manhattan distance between the pairs. Analogously, s_3 and s_4 retrieve top-k similar pairs and top-k dissimilar pairs, respectively, according to the product of the differences of the attributes. We conducted experiments for several other scoring functions and obtained results similar to the ones reported in this chapter.

Parameter	**Range**
Data distribution	real, **uniform**, correlated, anticorrelated
# of attributes (d)	2, **3**, 4, 5, 6
N (in thousands)	**10**, 50, 100, 500, 1000
K	1, 5, 10, **20**, 50, 100

Table 3.1: Experiment Parameters for Top-k Pairs Queries

Unless mentioned otherwise, for a fixed value of k and n, we issue four queries $Q_{(k,n,s_i)}$, one for each of the four scoring functions, and report the average query cost per object update. The table 3.1 shows the different parameters used in our experiments and the bold values are the default values used in the experiments unless mentioned otherwise.

Evaluating overall cost

To the best of our knowledge, we are the first to study the problem of top-k pairs over data stream. This problem is inherently different from other related problems such as k-closest pairs queries on moving objects [ZZS+05, UMY07], static top-k pairs queries [CLW+11] and incremental distance join [HS98] etc. Although at first it may seem easy to extend previous techniques, a careful analysis demonstrates that the extension of these techniques to answer k-closest (or top-k) pairs queries

over sliding windows is either non-trivial or inefficient.

We evaluate our algorithm (Algorithm 3) that answers the queries involving arbitrarily complex scoring function. Since it uses a K-staircase to maintain the K-skyband, our algorithm is called *SCase*. For an extensive evaluation of our algorithm, we carefully design two competitors called *Naïve* and *Supreme*. Below, we present the details.

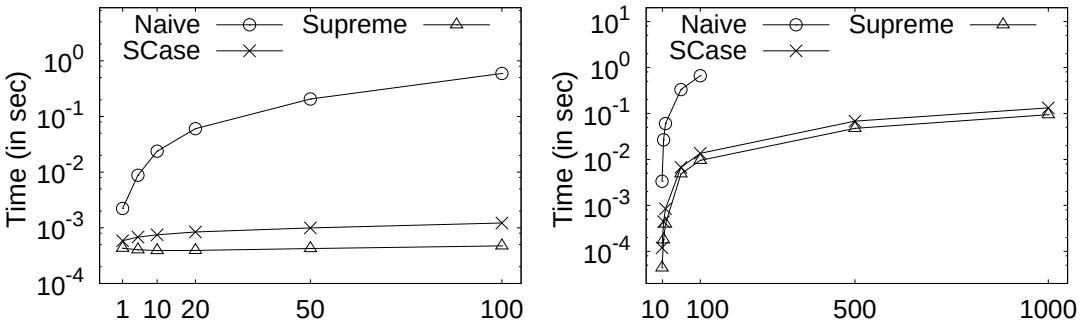
Naïve Algorithm. A naïve approach to answer continuous top-k pairs query is to maintain all $O(N^2)$ pairs in sorted order of their scores. However, this approach appeared to be too slow. Another serious drawback is that the space complexity is quadratic and is prohibitive for large sliding windows. Therefore, we devised a better naïve approach that uses $O(KN)$ space. For each newly arrived object, K pairs related to it with the smallest scores are computed. All $O(KN)$ pairs are kept sorted on their scores. When an object o_i expires, all the pairs related to it are deleted. Note that the object o_i may be among the top-K pairs of an unexpired object o_j. After we delete the pairs related to o_i, we need to update the top-k pairs of every such object o_j.

Supreme algorithm. We assume that there exists an oracle that answers questions without requiring any computation time. We use this oracle such that the supreme algorithm meets the lower bound cost[4]. More specifically, for query answering, we assume that the supreme algorithm requests oracle to return, in sorted order of scores, only the pairs of K-skyband that lie in the sliding window. The supreme algorithm returns first k pairs and requests oracle to stop. Clearly, the query answering cost of the supreme algorithm is $O(k)$ that meets the lower bound.

As implied by Theorem 2, every algorithm must maintain the pairs in K-

[4]Note that the performance of an algorithm also depends on the way it is implemented. However, we remark that the supreme algorithm is a reasonable benchmark to evaluate the scalability of our approach. Having said this, for a fair evaluation, the supreme algorithm is implemented by using the code that is a subset of the code used by our algorithm.

skyband for exact answering of top-k pairs queries. To maintain K-skyband, the supreme algorithm uses Algorithm 3 and computes only line 2 and line 3. The remaining steps are answered by the oracle in no time. Note that the skyband maintenance of the supreme algorithm meets the lower bound of $O(N)$.

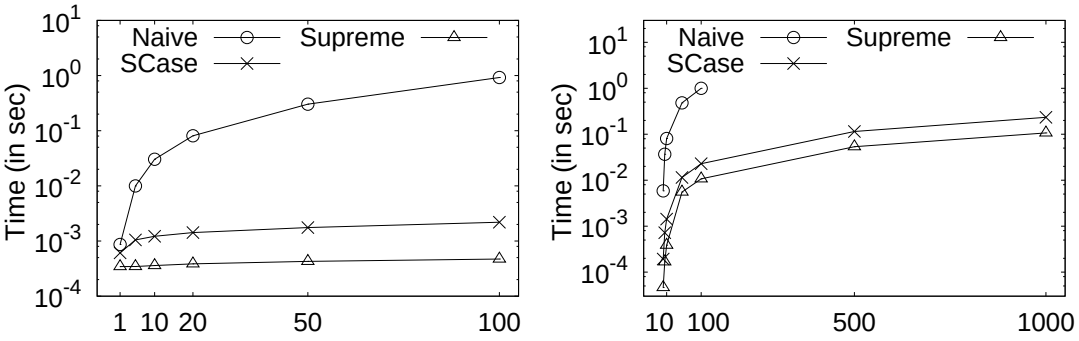


(a) Varying K (b) Varying N (in thousands)

Figure 3.8: Overall cost evaluation on the real data

In Figure 3.8, we compare our algorithm with other algorithms using the real sensor data set. We issue 100 top-k pairs queries $Q_{(k,n,s)}$ where $k \leq K$ and $n \leq N$ are randomly chosen for each query. Our algorithm demonstrates two to three orders of magnitude improvement over the naïve algorithm and performs reasonably well as compared to the supreme algorithm. For $N \geq 500,000$, the naïve algorithm did not complete its execution in 7 days and the estimated completion time was around 2 months. Therefore, we do not show results for the naïve algorithm for the larger values of N.

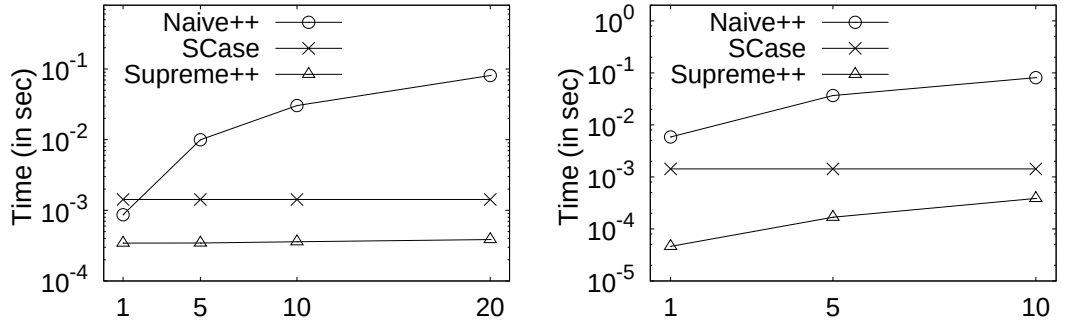
In Figure 3.9 and Figure 3.10, we perform experiments on synthetic data sets to conduct a more detailed evaluation. Since we also want to observe the performance of the algorithms for varying n and varying k, we decide not to randomly generate n and k. Instead, in each experiment, we run four queries each using a fixed value of n and k and using one of the four scoring functions (s_1, s_2, s_3 and s_4) presented in Section 3.7.1. In Figure 3.9(a) and Figure 3.9(b), we study the effect of K and N on both algorithms. For each query, we set $n = N$ (the largest sliding window)



(a) Varying K (b) Varying N (in thousands)

Figure 3.9: Effect of K and N on synthetic data

and $k = K$ (the largest possible value of k). The results are similar to the results obtained using the real data set.



(a) Varying k (b) Varying n (in thousands)

Figure 3.10: Effect of k and n on synthetic data

In Figure 3.10, we study the effect of k and n. As stated earlier, our algorithm does not know the values of n and k in advance hence maintains a K-skyband for most recent N objects. In contrast, for a more strict evaluation of our algorithm, we assume that both the naïve and the supreme algorithms know the values of n and k in advance. In effect, the supreme algorithm maintains k-skyband (note that $k \leq K$) for most recent n objects only. The naïve algorithm uses only $O(kn)$ memory instead of $O(KN)$ memory. We call these variations of the supreme and naïve algorithms as **supreme++** and **naïve ++**, respectively.

The results are reported in Figure 3.10(a) and Figure 3.10(b). In Figure 3.10(a),

the naïve ++ algorithm performs better for $k = 1$ because it needs to maintain only $O(n)$ pairs in total whereas we need to maintain 20-skyband ($K = 20$) for most recent $N = 10,000$ objects.

Figure 3.10(b) shows that our algorithm outperforms naïve ++ algorithm even for $n = 1000$ although it incurs maintenance cost to maintain a K-skyband for a window size N of $10,000$. Note that the complexity of supreme++ is $O(n)$ and the complexity of our algorithm is $O(N \cdot (\log(\log N) + \log K)$. Hence, the cost of supreme++ increases with increase in n whereas the cost of our algorithm remains unaffected.

Evaluating query answering module

In this section, we evaluate the performance of our query answering module.

Snapshot Query Answering. We compare our query answering algorithm with the supreme query answering algorithm as well as another algorithm called *linear* algorithm. The linear algorithm is the approach we discussed in the first paragraph of Section 3.4.1 and it takes time linear to the size of K-skyband in the worst case. Our query answering algorithm (Algorithm 2) is called *snapshot*. We study the effect of each of the parameters K, N, k and n, separately.

In Figure 3.11(a) and Figure 3.11(b), we study the effect of varying K and N, respectively. The default value of n is 1000 and the default value of k is 20. As expected, the cost of supreme algorithm is negligible. This is because, in all the experiments, the supreme algorithm needs to iterate over a link list of size k. The snapshot algorithm outperforms the linear algorithm and scales better with the increase in the values of K or N. The cost of linear algorithm increases because the size of K-skyband increases with the increase in K or N.

In Figure 3.11(c) and Figure 3.11(d), we fix the values of K and N and study

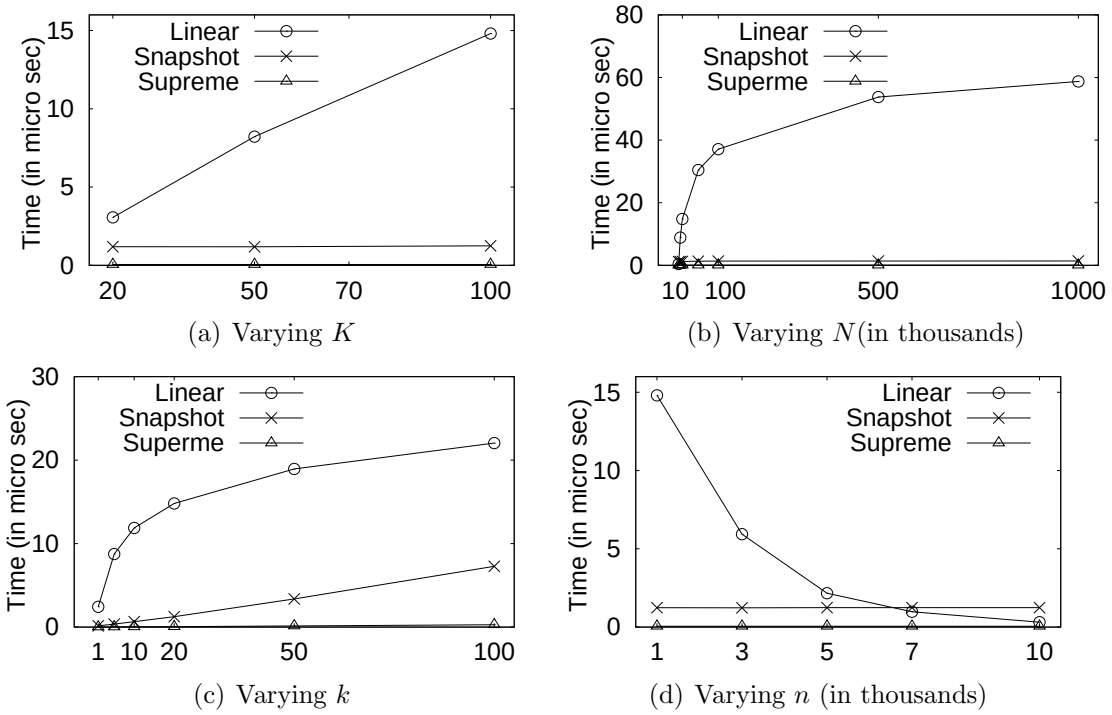


Figure 3.11: Linear vs Snapshot Algorithm

the effect of k and n on both of the algorithms. The default value of K is chosen to be 100 so that we can answer the queries with any $k \leq 100$. The snapshot algorithm performs better than the linear algorithm for varying k.

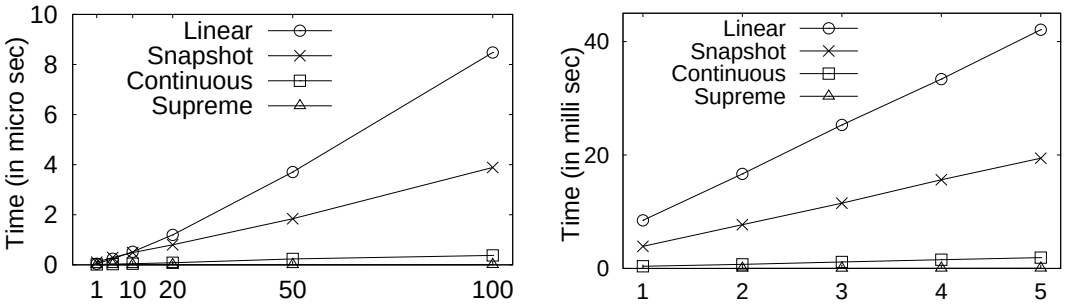
Figure 3.11(d) shows that the linear algorithm performs slightly better than the snapshot algorithm when the value of n is close to N. This is because the linear algorithm accesses the pairs in K-skyband in ascending order of scores and terminates when k pairs are found with age at most equal to n. The algorithm is expected to terminate earlier when n is large. Note that when $n = N$ the cost of linear algorithm is $O(k)$ which is impossible to be outperformed.

Recall that our complexity analysis shows that the cost of snapshot algorithm is $O(\log(\log n) + \log K + k)$. As anticipated by our complexity analysis, the cost of our snapshot algorithm increases with increase in k (see Figure 3.11(c)) but is not significantly affected by a moderate increase in K or n (see Figure 3.11(a) and

Figure 3.11(d)).

Continuous Query Answering. Next, we evaluate the performance of our continuous query algorithm which is denoted as *continuous* in the figures. The supreme algorithm for continuous query answering assumes that the oracle notifies it whenever a pair is deleted or added to the existing answer and the supreme algorithm updates the results accordingly. We also choose the linear algorithm and the snapshot algorithm as competitors such that these algorithms compute the results from scratch whenever the results are to be updated.

In Figure 3.12(a), we show the effect of K on the continuous query algorithm for 1000 queries that randomly choose the values of n and k. Figure 3.12(a) shows the average cost per query per object update. Clearly, our continuous query algorithm outperforms the linear and snapshot algorithms and scales better.



(a) Varying K (b) Varying $|\mathcal{Q}|$ (in thousands)

Figure 3.12: Evaluation of continuous queries algorithm

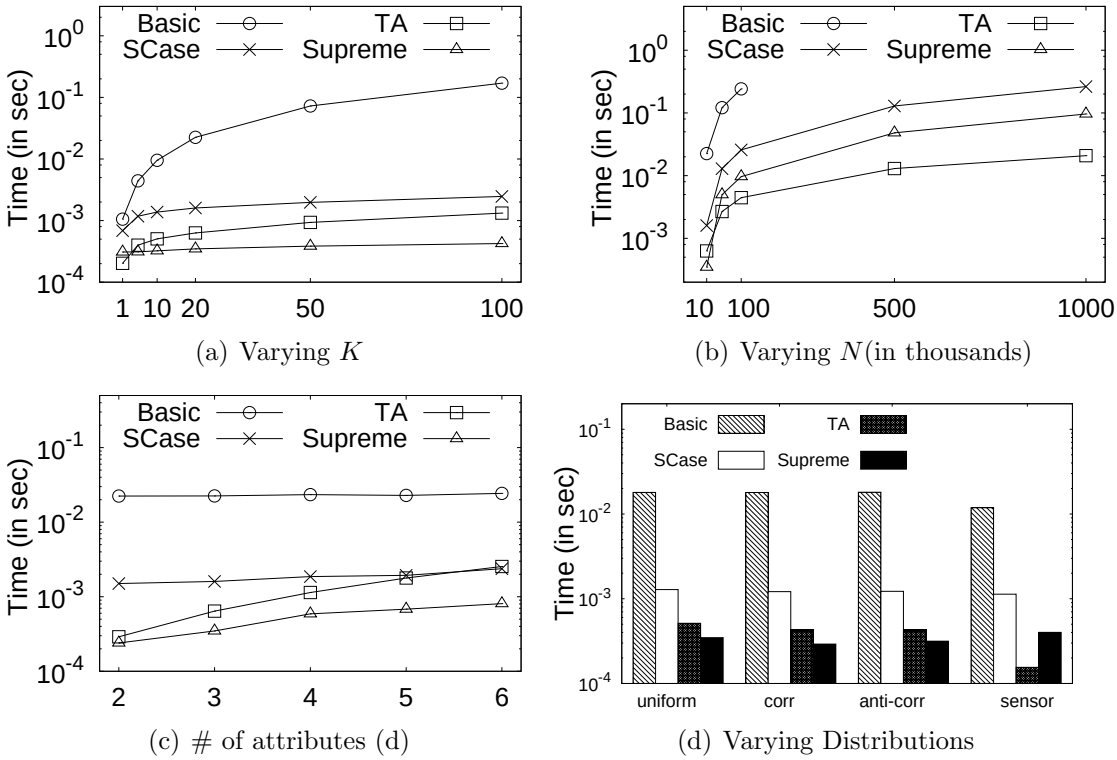
Figure 3.12(b) shows the performance of the algorithms for the increasing number of queries. Each query $Q_{(k,n,s)}$ uses a randomly chosen value of k and n. Figure 3.12(b) shows the total cost for all the queries per object update. Our continuous query algorithm outperforms the linear and snapshot approaches.

Evaluating skyband maintenance module

In this section, we evaluate our skyband maintenance module. We compare four algorithms. The *SCase* algorithm is the Algorithm 3 which uses K-staircase and can be applied on any arbitrarily complex scoring function. The *basic* algorithm is the same as Algorithm 3 but does not use K-staircase. As stated in Section 2.1, previous algorithms to maintain K-skyband [MBP06, BOPY07] cannot be directly applied. Nevertheless, we embedded all applicable optimizations (e.g., dominance counter) of their techniques in the *basic* algorithm. The *TA* algorithm is Algorithm 5 which is applicable only on the queries using global scoring functions. The supreme algorithm maintains the skyband as discussed in Section 3.7.1. Note that TA has an advantage over all other algorithms (including the supreme algorithm) that it knows that the scoring function is a global scoring function and uses its properties.

In Figure 3.13(a) and Figure 3.13(b), we study the affect of K and N, respectively. As expected, the TA algorithm always outperforms the basic and SCase algorithms. This shows the effectiveness of using optimizations for global scoring functions. Also, note that SCase algorithm outperforms the basic algorithm which shows the effectiveness of using the K-staircase. TA outperforms even the supreme algorithm when window size N is large. This is because TA utilizes the properties of the global scoring function and does not compute the score of all $O(N)$ objects when a new object arrives.

In Figure 3.13(c), we vary the number of attributes d used by the scoring functions and study the effect on the algorithms. The performance of TA degrades as the number of attributes increases. This verifies our complexity analysis given in Section 3.5.2. The cost of supreme algorithm increases mainly because the cost of computing the score of a pair increases as the number of attributes increases. The basic and SCase algorithms are not affected by the number of attributes because



(a) Varying K (b) Varying N(in thousands)

(c) # of attributes (d) (d) Varying Distributions

Figure 3.13: Skyband maintenance techniques

the main cost in these two algorithms is not the cost of computing the scores of the pairs.

In Figure 3.13(d), we show the effect of data distribution on the algorithms. TA consistently performs better than SCase and the basic algorithm on each different data set. Also, SCase algorithm performs significantly better than the basic algorithm.

Evaluating memory and theoretical analysis

Table 3.2 and 3.3 evaluate the memory cost of our algorithm and our theoretical analysis for varying K and varying N, respectively (other settings are default). The tables compare our memory usage with the lower bound memory required (as per Theorem 4). Note that the memory used by our algorithm is

quite close to the lower bound memory required. The tables also compare the theoretical value of K-skyband size with the experimental value of K-skyband size (average size of K-skyband for all queries in the system). Note that in our theoretical analysis (Theorem 3), we state that the expected size of K-skyband is $O(K(ln\ N - ln\sqrt{K})) = O(K \log (N/K))$. Our experiments show that the actual size of K-skyband is about $2(K(ln\ N - ln\sqrt{K}))$; this confirms the correctness of our theoretical analysis.

Below is the explanation of the legend used in the tables.

LB: Lower bound memory usage (in MB)

OUR: The memory used by our algorithm (in MB).

|SKB|: Experimental value of average K-skyband size (in number of pairs)

T $= K(ln\ N - ln\sqrt{K})$.

K	**LB (in MB)**	**OUR (in MB)**	**|SKB|**	**2T**
1	0.46	0.461	17.4	18.42
5	0.46	0.467	82.3	84.05
10	0.46	0.473	159.9	161.18
20	0.46	0.486	308.5	308.50
50	0.46	0.522	730.6	725.43
100	0.46	0.58	1398.0	1381.55

Table 3.2: Memory Usage on Varying K ($N = 10,000$)

N	**LB(in MB)**	**OUR(in MB)**	**|SKB|**	**2T**
1000	0.04	0.058	216.5	216.40
5000	0.23	0.254	280.8	280.77
10000	0.46	0.486	308.5	308.50
50000	2.28	2.312	372.9	373.88
100000	4.58	4.614	399.8	400.60
500000	22.88	22.920	465.7	465.98
1000000	45.78	45.822	496.8	492.71

Table 3.3: Memory Usage on Varying N ($K = 20$)

3.7.2 Top-k Objects Queries

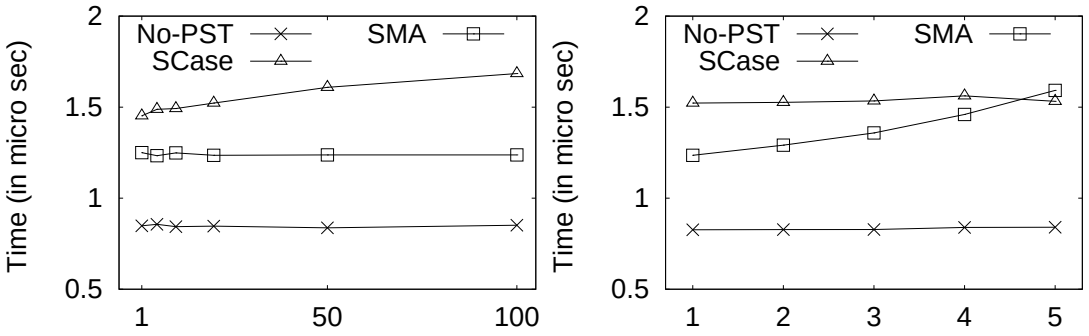
We compare our proposed algorithm with the state-of-the art algorithm SMA (Skyband Monitoring Algorithm) [MBP06] that answers top-k object queries for monotonic scoring functions. We obtain the source code of SMA from the authors and perform the experiments using the settings similar to those used in [MBP06]. We use the synthetic data set that follows anti-correlated distribution and consists of 10 millions objects. We also conducted experiments for other data distributions and observed similar trends. Table 3.4 shows the parameters used in experiments and the default values are shown in bold. The scoring function we used in the experiment is $s(o) = \sum_{i=1}^{d} o[i]$.

Parameter	**Range**
# of attributes (d)	2, 3, **4**, 5, 6
N (in millions)	**1**, 2, 3, 4, 5
K	1, 5, 10, **20**, 50, 100

Table 3.4: Parameters for Top-k Objects Queries

Running time

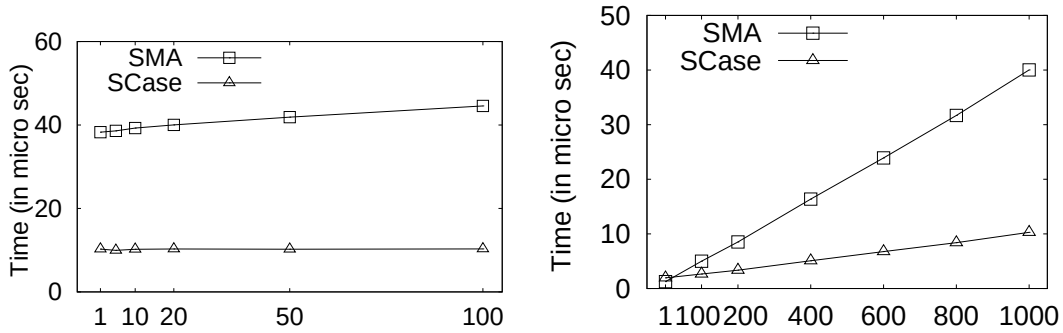
Figure 3.14 compares our algorithm with SMA for a top-k object query where the window size n is equal to N. Note that our algorithm maintains PST to support any window size $n \leq N$. If $n = N$, our algorithm is not required to store the PST and the query can be answered by returning first k objects from the K-skyband which is kept sorted on scores by the skyband maintenance module. We use this optimization to answer the query where $n = N$ and call this algorithm `No-PST`. The algorithm that uses PST is called `SCase`. `No-PST` outperforms the other two algorithms. The cost of `SCase` is higher than the cost of other two algorithms because it needs to maintain the PST to support any $n \leq N$.



(a) Varying K (b) Varying N (in millions)

Figure 3.14: Top-k objects queries for $n = N$

Next, we compare our algorithm with SMA for the queries with window sizes $n \leq N$. We extend SMA so that it can also support any $n \leq N$. Specifically, to support any query with $n < N$, SMA ignores every object that has age greater than n during the computation of top-k queries. In Figure 3.15, we randomly generate 1000 queries with each query using randomly generated values k and n ($k \leq K$ and $n \leq N$). Although our proposed algorithm is more general and can support arbitrarily complex scoring functions and out-of-order streams, the results demonstrate that our proposed algorithm outperforms SMA and scales better.

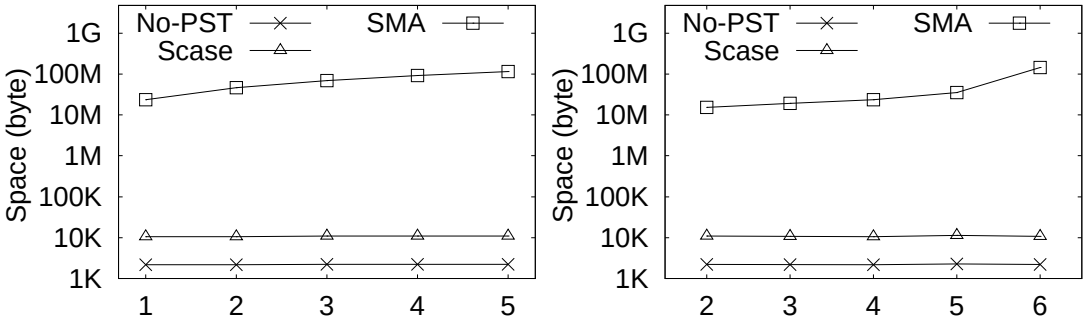


(a) Varying K (b) Varying the number of queries

Figure 3.15: Top-k queries for randomly generated k and n

Memory usage

In this section, we show that the memory consumed by our algorithm is much lower than the memory used by SMA. This is because our algorithm maintains only the K-skyband whereas SMA indexes all N objects in a grid data structure. Figure 3.16(a) and Figure 3.16(b) show the memory used by the algorithms for varying N and varying d (number of attributes used in the scoring function), respectively. The memory used by SMA is significantly higher (please note that log-scale is used). The memory used by `No-PST` is smaller than `SCase` because the former does not need to store the priority search tree. The memory consumption of SMA increases with the increase in d because d-dimensional grid is required which consumes higher memory. In contrast, our algorithms are not affected by d.



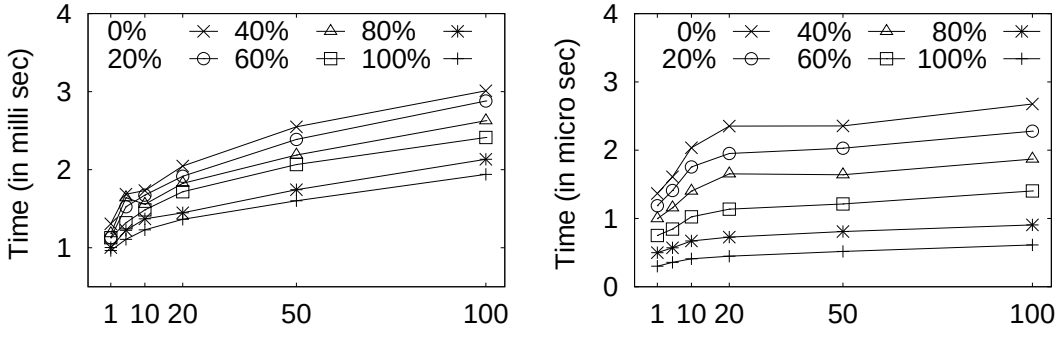
(a) Varying N (in millions) (b) Varying number of attributes d

Figure 3.16: Evaluating the memory usage

3.7.3 Miscellaneous

Results for out-of-order streams. We present the results for out-of-order data streams where objects may arrive late. For each object that arrives late, we randomly generate a value y between 1 to N and delay it by a value y (e.g., its age when it arrives is y). Figure 3.17(a) and Figure 3.17(b) show the results for top-k pairs queries and top-k objects queries, respectively. x% denotes that x percentage

of the objects arrive late. Note that 0% corresponds to the in-order data streams. In each experiment, we run 100 queries and report the overall running time. As anticipated by our theoretical analysis, the performance of the algorithms is better for the cases when more objects arrive late.

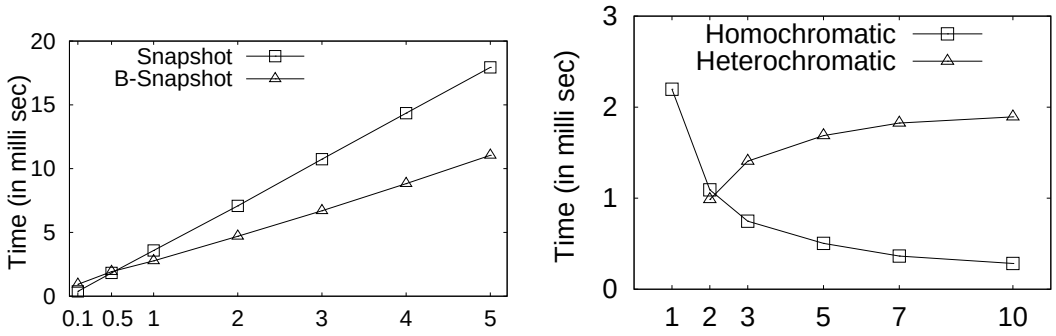


(a) Top-k pairs queries (b) Top-k objects queries

Figure 3.17: Out-of-order data streams

Batch query processing. We next evaluate our technique for the batch query processing algorithm proposed in Section 3.6.3. The algorithm that uses the batch query processing is denoted as B-snapshot. Note that the complexity of the snapshot algorithm for $|\mathcal{Q}|$ queries is $O((\log|SKB|+k)|\mathcal{Q}|)$ whereas the complexity of B-snapshot is $O(|SKB|+k|\mathcal{Q}|)$. According to this analysis, B-snapshot performs better when $|\mathcal{Q}|$ is large enough such that $|\mathcal{Q}|\log|SKB| > |SKB|$. Figure 3.18(a) compares the cost of snapshot and B-snapshot algorithms and verifies the complexity analysis that B-snapshot performs better when the number of queries is large.

Results for chromatic queries. In Figure 3.18(b), we vary the number of colors (each object is randomly assigned one color) and study the performance of our algorithms for heterochromatic and homochromatic queries. Note that the homochromatic query is the same as a non-chromatic query when only one color is used. The cost of both homochromatic and heterochromatic queries is lower than



(a) Batch processing for varying $|\mathcal{Q}|$(in thousands)

(b) Chromatic queries for varying number of colors

Figure 3.18: Batch processing and chromatic queries

the cost of non-chromatic queries. The cost of homochromatic queries decreases with the increase in number of colors because the number of valid pairs decreases. In contrast, the cost of heterochromatic queries increases because the number of valid pairs increases when the number of colors is larger.

3.8 Conclusion

We present efficient techniques to answer a broad class of top-k pairs and top-k objects queries over sliding windows. The efficiency of the proposed techniques is evaluated by a detailed complexity analysis and an extensive experimental study. The proposed framework can handle arbitrary scoring functions, supports queries with any window size and works for out-of-order data streams.

Chapter 4

Continuous Monitoring of Top-k Loyalty Queries

Chapter 4 presented our technique to answer continuous loyalty queries and threshold queries over sliding windows.

4.1 Overview

A traditional query Q returns every object that satisfies the query criteria at the time t query was issued. The traditional queries do not consider the history of the objects' values, i.e., the values of objects in the recent past. Hence, the traditional queries fail to capture how persistently an object satisfies the query criteria. Consider the example of a stock broker who issues a query at time t to retrieve the profitable stocks. He may define a set of criterions to denote the profitability. A traditional query returns every stock s that satisfies the criterions at time t. Although a returned stock s meets the criteria at time t, the history of the stock s may indicate that it usually does not satisfy the criteria and is not a good choice for investment. Hence, a query that does not take into account the history of stock

items is not suitable.

To address the above mentioned problem, in this chapter, we propose a new query operator called *loyalty queries*. A loyalty query considers how persistently the objects satisfy the query criteria. Consider a *traditional query* Q that defines a set of criterions. Let $Q(o, t)$ denote whether an object o satisfies the criteria of query Q at time t or not. More specifically, $Q(o, t)$ is true if and only if the object o satisfies the query criteria at time t. Let T be a user defined parameter. The loyalty of an object o is the total time duration for which $Q(o, t)$ is true within last T time units. The measure is called "loyalty" because it signifies how persistently the object o meets the criteria in the recent past. In this chapter, we study continuous *top-k* loyalty queries that continuously report k objects with the highest loyalties. We also show that the proposed approach can be easily used to answer threshold loyalty queries that return every object with loyalty greater than a given threshold.

Loyalty queries have many interesting applications in different areas such as location based services, wireless sensor network, stock market, traffic monitoring, and internet applications, etc. For instance, in the example of the stocks, the stock broker may retrieve top-k loyal objects to retrieve better options for investment. Consider another example of a paid parking system that notifies the nearby cars of its availability, i.e., the cars that are in its *influence zone* [CLZZ11] or the cars that are within 1 Km of the parking space [CBL$^+$10]. At a given time t, the system may send SMS to some cars that satisfy the criteria (e.g. a car that lies within 1Km at time t). However, most of such cars may just be passing through that area and may not be interested in parking. On the other hand, a car that satisfies the criteria for majority of the time in recent past may actually be looking for the parking. Hence, the system may use top-k loyalty queries to send notifications to such cars.

Consider another example of a wireless sensor network. An environmental scientist may be interested in monitoring the most rainy sites. A traditional query selects every sensor that reports rain at a given time t. Clearly, the query may miss a site s that is usually the rainiest but it is not raining there at time t. Moreover, the results are also affected by an erroneous reading by a sensor at time t. For these reasons, a top-k loyalty query is a more feasible tool to retrieve the rainiest sites.

We next summarize our contributions in this chapter.

- **Novel query operator**. To the best of our knowledge, we are the first to study continuous loyalty queries. In this chapter, we formalize the definition of loyalty queries and present a framework that efficiently solves the loyalty queries.

- **Continuous updates.** We study the problem in a continuous time domain where the updated results are reported as soon as the results change as opposed to the *time-stamp* model where the results are updated after every u time units. Note that the time-stamp model suffers from either high computational cost or low accuracy. More specifically, if u is small, the computation cost increases because the results are to be updated more often. On the other hand, if u is large, the accuracy is reduced because the results may have become invalid between two successive time-stamps. The continuous updates provided by our algorithm do not have these limitations.

- **Optimal computation cost**. An object issues an update if it starts satisfying the query criteria or if it stops satisfying the query criteria. Note that the top-k loyal objects may change whenever an object issues an update. Let N be the total number of object updates issued in the last T time units. Upon

receiving an object update, our algorithm updates the top-k loyal objects in $O(\log N)$. We prove that this is the lower bound update cost for top-k loyalty queries, hence our algorithm is optimal.

- **Low communication cost**. In distributed environment, the updates of an object are generated locally and sent to a centric server for query processing. We obverse that some updates do not contribute to computing the final results of the loyalty queries. These updates are so called *trivial* updates. We further develop an efficient pruning technique on the trivial updates to further reduce the communication cost and the overall computation cost as well.

- **Extensive evaluation and analysis**. We theoretically analyse the complexity of our algorithm and prove that it meets the lower bound cost. We also conduct experiments to show the effectiveness and the efficiency of our proposed approach. We compare our algorithm with the Bentley-Ottmann sweep line algorithm [BO79]. For N object updates, the total cost of the Bentley-Ottmann algorithm is $O(N^2 \log N)$ in the worst case. In contrast, the total worst case cost of our algorithm is $O(N \log N)$. Extensive experiments conducted on both real and synthetic data sets demonstrate that our proposed approach is an order of magnitude faster than the Bentley-Ottmann algorithm.

The remainder of the chapter is organized as follows. In Section 4.2, we give an overview of the related work and formalized definition of loyalty queries. We introduce our framework in Section 4.3, while in Section 4.4 we present our solution to the top-k loyalty queries. The techniques of the threshold queries are presented in Section 4.5. The experimental results are reported in Section 4.6. Section 4.7 concludes the chapter.

4.2 Preliminaries

In this section, we first present an overview of the related work. Then, we formally define the problem studied in this chapter.

Traditional queries. A traditional query Q defines a set of criteria. Given an object o and a timestamp t, we use $Q(o,t)$ to denote whether o satisfies the query criteria of Q at t. For ease of presentation we define $Q(o,t)$ using a step function.

$$Q(o,t) = \begin{cases} 1 & \text{if } o \text{ satisfies the query critera of } Q \text{ at } t; \\ 0 & \text{if } o \text{ does not satisfy the query critera of } Q \text{ at } t. \end{cases}$$

Consider an application for monitoring the cars around the parking space and the parking system notifies the cars with high loyalties in Figure 4.1. Given two moving objects(cars) o_1 and o_2, o_1 enters the space at time 5 and leaves at time 8. o_2 enters at time 10 and leaves at time 18. Therefore, $Q(o_1,6) = 1$ and $Q(o_2,6) = 0$.

Sliding windows. Usually users are not interested in the entire past history of the data stream but rather the recent data over sliding windows. In this chapter, we consider a data stream model in the continuous time domain. For a fixed length of time period T, a sliding window contains all the objects and the corresponding attributes within last T time units. We argue that the stream model in the continuous time domain is more general than the model in the discrete time domain. In the rest of the chapter we only consider our problem in the continuous time domain. However, our techniques can also be applied to answer the loyalty queries in the discrete time domain. Note that different types of models can be adopted instead of using a sliding window model. For example, users consider a tilted history for loyalty queries. That is, the more recent instants carry heavier weights. This problem is more challenging and could be the further work of our research.

Loyalty of an object. Given a traditional query Q and a sliding window size T, we define $loyalty(o,t)$ (the loyalty of an object o at time t) as follows.

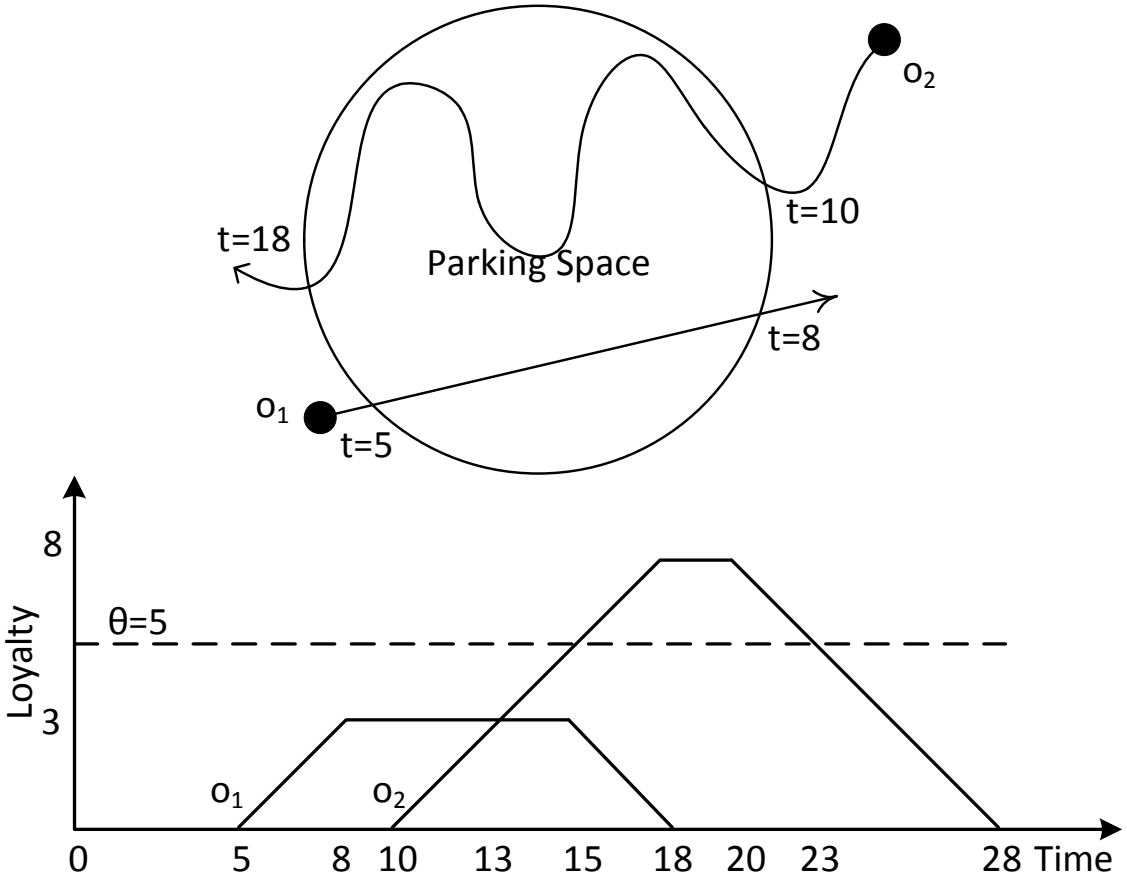


Figure 4.1: Example of Loyalty Queries

$$loyalty(o,t) = \int_{t-T}^{t} Q(o,x)dx$$

The loyalty of an object o shows how long o is a query result of Q during the last T time units. Without loss of generality, in this chapter we prefer the objects with higher loyalties.

Consider a sliding window of size 10 ($T = 10$) in Figure 4.1. Then, $loyalty(o_1, 8)$ (the loyalty of o_1 at time 8) is 3 because o has been around the parking space for 3 time units. Note that the coordinates in Figure 4.1 present the loyalties of o_1 and o_2 as the time t changes. Similarly, we can see that $loyalty(o_2, 13) = 3$.

Top-k loyalty queries. Consider a set of objects O, a traditional query Q, a

sliding window size T and a parameter k. The top-k loyalty query at time t returns an answer set from O that consists of k objects such that for every object o in the answer set and for any other $o' \in O$, $loyalty(o,t) \geq loyalty(o',t)$.

Consider the example in Figure 4.1. If we monitor the top-1 loyal object and the window size T is 10, o_1 is the result of the top-1 loyalty query from 5 to 13 and o_2 is the result from 13 to 28.

Threshold loyalty queries. Consider a set O of objects, a traditional query Q, a sliding window size T and a threshold θ, the threshold loyalty query at time t returns an answer set from O that consists of any object o such that $loyalty(o,t) \geq \theta$.

In Figure 4.1, given the threshold $\theta = 5$, o_2 is a result of the threshold loyalty query from time 15 to 23.

Continuous queries. In this chapter, we study the continuous loyalty queries, namely, we issue the query once and it monitors the query results continuously. Since we solve the queries in the continuous time domain, it is impossible to compute the results for an infinite number of time snapshots. In this chapter we shows that although the loyalty of an object is changing over time, we do not need to update the loyalty and the query results for every time snapshot.

Note that the top-k loyalty queries are more challenging to solve, since we need to consider the relationships among the objects. In this chapter we mainly focus on solving the top-k loyalty queries.

4.3 Framework

In this section we introduce our framework for solving loyalty queries for any given traditional query Q. In real world scenarios, given a set of objects of observation O, the objects may be distributed and users may want to know the global results

of a loyalty query. Thus, we present a general framework that aims to handle the loyalty queries in both centralized and distributed environments.

Our framework consists of two main components: the traditional query module and the loyalty query module.

4.3.1 Traditional Query Module

Given a traditional query Q (e.g., a range query), each object issues an *update* when it starts satisfying the query criteria or when it stops satisfying the criteria. More specifically, traditional query module is responsible to report to loyalty query module whenever the value of $Q(o, t)$ is changed for any object o.

In a centralized system, the system detects the updates of the objects and processes the updates internally. In distributed environments such as client-server architectures, an object (client) sends a message to the loyalty query module (server) to report an update.

Object updates. Given a query Q and an object o, we say there is an update u of o at time t if the derivative of Q at t is infinity, i.e., $\frac{d}{dt}Q(o, t) = \infty$. In other words, $Q(o, t)$ changes at time t.

Consider the example in Figure 4.1, a moving object issues the update only when it enters or exits the monitoring space. Therefore, o_1 reports two updates at time 5 and 8, and o_2 reports two updates at time 10 and 18.

Basically we adopt existing techniques for continuously monitoring the results of the traditional queries. A straightforward way is to continuously monitor the traditional query result and report once the update occurs. However, since most state-of-art techniques for continuous monitoring queries compute and output their results incrementally, it is seamless to report updates based on these online algorithms. For instance, the techniques in the papers [CBL+10, CLZZ11] can work as

a traditional query module to find the loyal objects within the query range or the influence zone for a majority of the recent time. As this part of work has already been done and our aim is to support a variety of traditional queries in our loyalty query framework, in this chapter we focus on efficiently processing of the loyalty queries.

4.3.2 Loyalty Query Module

If we assume the attributes of an object is varying continuously such as a moving object, the number of updates during a time period is finite. For a specified loyalty query, it receives updates from the traditional query module in the form of an update stream $U = \{u_1, u_2, u_3, ..., u_n\}$. The updates arrive in the time order. We process the updates continuously in the loyalty query module and output the results to users.

Our query algorithm is triggered only when the update arrives or a possible result change of the loyalty query happens. Therefore, we can output updated results of the loyalty queries when the result changes. In other words, we report which object is newly added in the answer set or which object is removed from the set. Then, the cost of each output is $O(\Delta)$ where Δ is the number of result changes. Figure 4.2 shows the general framework for processing loyalty queries in a distributed system.

4.4 Top-k Loyalty Queries

Before we describe the algorithm of processing threshold loyalty queries, we present the details of answering top-k loyalty queries. This is because it is more challenging to solve the top-k loyalty queries and similar techniques can be applied to the

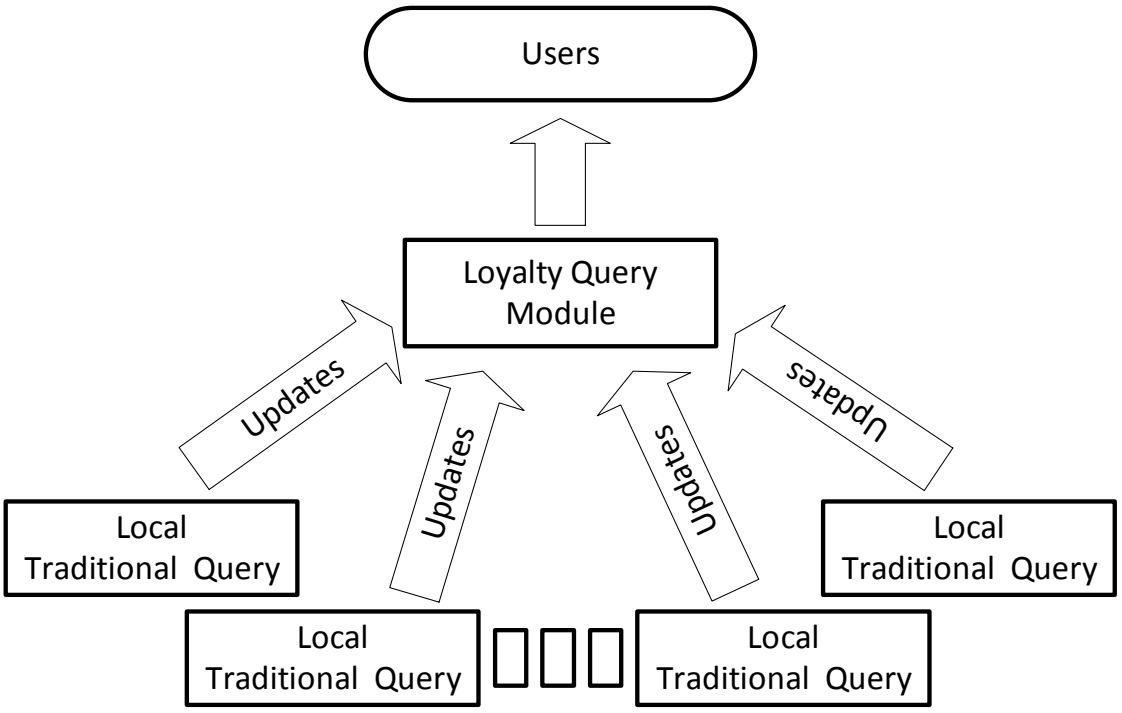


Figure 4.2: Framework of Loyalty Queries

threshold queries.

Initially, we present the base algorithm of solving top-k loyalty queries. We then extensively analyze the time and space complexity of the proposed approach. Finally, we present an efficient pruning technique to further accelerate the base algorithm.

4.4.1 Algorithms

Consider a top-1 loyalty query. If we draw the loyalty changes in a loyalty-time plane (see Figure 4.1), intuitively this problem is similar to finding the upper envelop in this plane. Similarly the top-k query is to retrieve k upper envelops. This problem can be solved by the line sweep algorithm in computational geometry. Given N line segments in the plane, the Bentley-Ottmann sweep algorithm [BO79, PS85] maintains the exact vertical ordering of the intersections of

the line segments, when the vertical line sweeps the plane from left to right. The total cost is $O((N+M)\log N)$ where M is the number of intersections of the line segments. In the worst case, the number of intersections M can be $O(N^2)$. A simple example is that the half of lines are horizontal and the other half are increasing. In this case the number of intersection is $N^2/4$. Therefore, the overall complexity can be $O(N^2\log(N))$. In our problem, we use N to denote the number of updates issued in the last T time units. Then, the amortized cost of the Bentley-Ottmann algorithm is $O(N\log N)$ for each update. In this chapter we present an algorithm to answer the top-k loyalty queries in $O(\log N)$ time for each update. The space requirement of our algorithm is $O(N)$.

Before we describe the algorithm, we show some obversion for handling updates in the sliding windows to enable the efficient computation.

States of objects. The state of an object o denotes whether the loyalty of o is increasing, stationary or decreasing. The state of o can be derived by computing the derivative of $loyalty(o,t)$.

$$\begin{aligned} state(o,t) &= \frac{d}{dt} loyalty(o,t) \\ &= \frac{d}{dt}\int_{t-T}^{t} Q(o,x)dx \\ &= Q(o,t) - Q(o,t-T) \end{aligned}$$

As shown above, $state(o,t)$ depends on the traditional query result at the current time $Q(o,t)$ and the result T time before the current time $Q(o,t-T)$. Moreover, there are only three types of states: increasing, stationary and decreasing.

- *Increasing.* The loyalty of o is increasing if $state(o,t) = 1$.
- *Stationary.* The loyalty of o is stationary if $state(o,t) = 0$.
- *Decreasing.* The loyalty of o is decreasing if $state(o,t) = -1$.

In Figure 4.1, the loyalty of object o_1 is increasing from 5 to 8. Then, the loyalty of o_1 is stationary from 8 to 15 and finally becomes decreasing from 15 to 18.

Echo updates. As soon as an *original* update arrives from the traditional query module, we know the traditional query result at the current time $Q(o,t)$ changes. Moreover, these updates will expire from the sliding window after T time, which will affect $Q(o,t-T)$. Therefore, we clone a series of *original* updates and make them take effect after T time. These updates are annotated as *echo* updates. For example, in Figure 4.1 we retrieve an original update at time 5 that o_1 becomes a result of the range query. Given $T = 10$, the echo update is created at time 15. The updates stand for both original and echo updates in the following of this chapter, unless mentioned otherwise.

Determining states. When we receive an original update u from a traditional query, we update the current query result $Q(o,t)$. Then we create an echo update u'. The timestamp of u' is $t+T$ and we also attach the new query result. Therefore, $state(o,t)$ can be computed by maintaining $Q(o,t)$ and $Q(o,t-T)$ correctly. In our algorithm we only handle the update if the state of an object changes.

Data structures. In order to efficiently maintain the top-k loyal objects over sliding windows and the sequence of future updates and events, our algorithm maintains the following data structures:

- **Update queue** U (a FIFO data structure) is utilized to maintain a sequence of echo updates. Each update is associated with the timestamp t_4 when it will be issued, the object o and the updated traditional query result $Q(o,t)$. The echo updates are created in the sequence of the original updates. Therefore, we can simply use a FIFO to organize the echo updates.

- **Border object** BO is denoted as the $(k+1)$th loyal object at time t. We

set BO empty if the number of objects is less than $k+1$. We define the *border line* indicating the $(k+1)$th line segment which divides the top-k lines and the remaining lines in the loyalty-time plane. In our algorithm only the line intersections related to the border line are processed. Consider a more complicated example of the top-2 loyalty query in Figure 4.3. The object o_3 is the 3rd loyal object from t_1 to t_3. Hence, $BO = o_3$ from t_1 to t_3. We mark the border line with a bold polyline in Figure 4.3.

- **Event queue** E (a priority queue) is utilized to maintain a sequence of potential future events. *Events* denote the potential future result changes of the loyalty queries. The result changes occur only when the border object swap its order of the loyalty with another object. In the loyalty-time plane, the event is created when one line will potentially intersect the border line in the future. If an event is created at time t, each event is associated with the *signatures* of the border object BO and another o at time t. A signature is the identification of the last update of an object. The signature of o will be changed if any update or event related to o is processed. An event is invalid and will not be processed if the signature of BO or o of the event is not up-to-date. The event is inserted into the event queue with the timestamp t' where t' is the potential intersecting time. Consider the example in Figure 4.3. We can predict that the line segment of o_1 will potentially intersect the border line at t_3. Therefore, an event is created to handle the intersection.

- **Top-k sets** $A = A_+ \cup A_= \cup A_-$ maintain the objects geometrically *above* the border object, namely the top-k loyal objects. A is divided into three subsets according to the states of the objects. A_+, $A_=$ and A_- are the subsets of the top-k objects with increasing, stationary and decreasing states respectively. Each subsets is organized in a binary search tree and the elements in the

subset are sorted in the decreasing order of their loyalties. Consider the example of Figure 4.3. A contains two objects o_1 and o_2 at t_1. $A_+ = \{o_2\}$ and $A_= = \{o_1\}$.

- **Bottom sets** $B = B_+ \cup B_= \cup B_-$ maintain the remaining objects *below* the border object. B is also divided into B_+, $B_=$ and B_- according to the states. Note that unlike other subsets, B_- can be organized just in a list without sorting their loyalties. B_+ and $B_=$ are represented explicitly in the binary search trees with the decreasing order of loyalties. Consider the example of Figure 4.3. B contains one object o_4 at t_4 and $B_+ = \{o_4\}$.

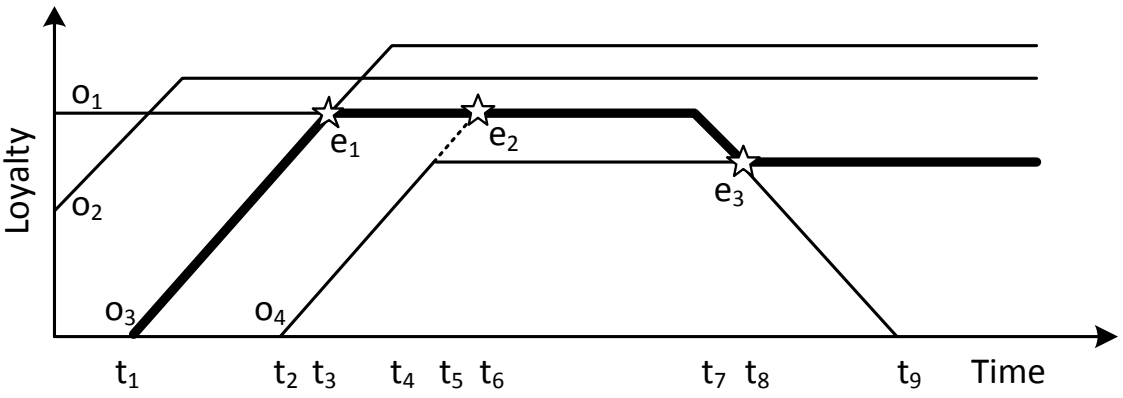


Figure 4.3: Example of Top-2 Loyalty Queries

Solution overview. Before we present the details of our algorithm for processing top-k loyalty queries, we show the main idea of our algorithm. The algorithm uses a sweep line approach to process updates and create events for handling the possible result changes. The algorithm is triggered when 1) an original update arrives from traditional query module, or 2) an echo update arrives from the update queue, or 3) an event arrives from the event queue. We make sure that our algorithm correctly maintains the border object and the objects in top-k set. An event is created if a possible result change of the loyalty query will occur in the future.

Algorithm 8: ProcessUpdate(u)

1: Determine $state(o, t)$ and update $loyalty(o, t)$.
2: **if** $o \in A$ **then** /* o `is in the top-`k `set` */
3: Remove o from the subset A_i
4: Add o into the corresponding subset A_j
5: **else if** $o \in B$ **then** /* o `is in the bottom set` */
6: Remove o from the subset B_i
7: Add o into the corresponding subset B_j
8: **else if** $o \notin BO$ **then** /* o `is a new comer` */
9: **if** $|A| < k$ **then** /* `# of objects less than` k */
10: Add o into A_+
11: **else if** $BO = \emptyset$ **then** /* `# of objects is` k */
12: $BO = o$
13: **else**
14: Add o into B_+
15: CheckSetVariation(BO, A, B) /* `Call Algorithm 9` */
16: Update the signature of o.

Processing updates. When a new update arrives from the update queue, we first recompute the state and loyalty of the corresponding object. Then, the object is moved to the correct subset. As the position of the object in a subset may be changed, we check the subsets and the border object and create the possible events.

Algorithm 8 shows our algorithm for processing a newly arriving update. First we determine the state of the object o based on the query results on both slides of the sliding window (see line 1). If the update is original, we create an echo update by cloning the original update and insert it into the update queue P. Since the state of the object o changes, we move o into the corresponding subset based on its

state and current position(lines 2–8). As the orders of elements in the subsets may change, we call Algorithm 9 to check the variation related to the border line and create new events to handle the future intersection(line 15). Finally we update the signature of o (line 16) as the state of o changes.

Algorithm 9: CheckSetVariation(BO, A, B)

```
1:  if A_=.last is varied and state(BO, t) is increasing then
2:      AddEvent(A_=.last, BO)
3:  else if A_−.last is varied and state(BO, t) is not decreasing then
4:      AddEvent(A_−.last, BO)
5:  else if B_+.first is varied and state(BO, t) is not increasing then
6:      AddEvent(B_+.first, BO)
7:  else if B_=.first is varied and state(BO, t) is decreasing then
8:      AddEvent(B_=.first, BO)
9:  if BO is varied then
10:     if state(BO, t) is increasing then
11:         AddEvent(A_=.last, BO)
12:         AddEvent(A_−.last, BO)
13:     else if state(BO, t) is stationary then
14:         AddEvent(A_−.last, BO)
15:         AddEvent(B_+.first, BO)
16:     else if state(BO, t) is decreasing then
17:         AddEvent(B_+.first, BO)
18:         AddEvent(B_=.first, BO)
```

Handling set variations. Algorithm 9 shows the procedure of handling the variation of the subsets and creating events for the possible result changes. We observe that any intersection related to the border line is associated with the line segment

immediately above or below the border line in each subset. Another important observation is that the two line segments with the same state(in the same subset) will not intersect each other. Therefore, we only check the last elements(objects with the minimal loyalty) in $A_=$ and A_-, and the first elements(objects with the maximal loyalty) in B_+ and $B_=$, which are the only potential line segments (objects) to first intersect the border line without considering the new updates in the future (see lines 1–8). In Figure 4.3, $A_=$ contains two objects o_2 and o_3 at t_6, and the last element in $A_=$ is o_2. Then, we consider the state change of the border object BO. Based on the state of the border line, two events are created to handle the possible intersections(lines 9–18).

Algorithm 10: AddEvent(o, BO)

1: Compute the intersecting time t' of o and BO.

2: Create an event e associated with o, BO and their signatures.

3: Insert e into event queue E with timestamp t'.

Creating events. Algorithm 10 shows how we create a new event. We first compute the intersecting time t' of the two lines segments(line 1). Then, the event e is created and inserted into the event queue E with t'(lines 2 and 3). Note that it is important for us to store the signature information of o and BO with event e. The change of the signature of o indicates that the state or position of the object has been updated before the event occurs. Therefore, the event is invalid and will not be processed.

Processing events. The details for processing an event is shown in Algorithm 11. When an event e arrives from the event queue E, we first check the validity of the line intersection by verifying the signatures (line 1). If it is valid, we swap the positions of BO and o (lines 2 and 3), and call Algorithm 9 again since the subsets and BO are changed(see line 4). We update the signatures of the objects as well

Algorithm 11: ProcessEvent(e)

1: **if** the signatures of o and BO are not varied **then**
2: Add BO into the subset of o and remove o from the subset.
3: $BO = o$.
4: CheckSetVariation(BO, A, B)
5: Update the signatures of o and BO.

(see line 5), since the positions of the objects are changed.

Handling objects with zero or maximum loyalty. Note that in the above algorithms we do not especially handle the objects with zero loyalties or maximum loyalties(the loyalty is T). Here, we show that these objects can be processed more efficiently. For an object o with $loyalty(o,t) = 0$ and $state(o,t) = 0$, we simply remove o from the subsets. For the objects with $loyalty(o,t) = T$ and $state(o,t) = 0$, we maintain a list F to store the objects instead of placing them in $A_=$. We can save the cost because maintaining the list is constant in time.

Example 1: Consider the top-2 loyalty query shown in Figure 4.3. Initially, there are two objects o_1 and o_2 with non-zero loyalties. An update of o_3 arrives at t_1. o_3 becomes the border line object(line 11 in Algorithm 8). We mark the border line with a bold line in Figure 4.3. Then, we check the set variation(Algorithm 9). Since BO has been changed, we create an event e_1 with the last element in $A_=$ (o_1) for possible order swapping at t_3(line 11 in Algorithm 9). We mark the created event with a star. Note that we only create and process the events (intersections) related to BO. An update of o_4 arrives at t_2. We check subsets variation and no event is created. Event e_1 is processed at t_3. o_3 is moved into A_+ and o_1 becomes the border object(line 2 and 3 in Algorithm 11). We check the set variation (line 4 in Algorithm 11) and create an event e_2 with o_4 at t_6 (line 15 in Algorithm 9). After that o_4 issues another update at t_5 and the signature of o_4 is changed. Therefore,

e_2 is invalid and is not processed at t_6. At t_7, o_4 issues an update and the state of BO is changed. After checking the variation, e_3 is created similarly.

4.4.2 Analysis

Proof of Correctness

In the proposed algorithm, we make the border object BO present the $(k+1)$th loyal object correctly. All the potential events (intersections) related to BO are created and processed. Therefore, we always make the following inequalities hold.

$$\begin{cases} \min_{o \in A} \{loyalty(o,t)\} \geq loyalty(BO,t) \\ loyalty(BO,t) \geq \max_{o \in B} \{loyalty(o,t)\} \\ |A| \leq k \end{cases}$$

In our algorithm the objects in top-k set cannot be changed unless BO is changed. As a consequence, our algorithm correctly determines the top-k loyal objects.

Performance Analysis

We first analyze the time complexity of our algorithm. As we use binary search trees to maintain the subsets, the cost of inserting or removing an object in a subset of A is $O(\log k)$ and the corresponding cost in a subset of B is $O(\log L)$ where L is the number of objects which have updates in the last T time unit. The cost of insertion in the update queue is $O(1)$ because the update queue is a FIFO. Let M be the number of events processed in the last T time units and M' be the number of events created in the last T time units. Note that some created events may become invalid and will not be processed in the future. As the event queue is organized by a priority queue, the cost of insertion in the event queue is $O(\log M')$, where M' is also the

size of the event queue. Let N be the number of updates issued in the last T time units. For each processed update and event, the algorithm creates constant number of events. Therefore, $M' = O(N+M)$. Then, the total cost in the last T time units is $O((N+M)(\log M' + \log k + \log L) = O((N+M)(\log (N+M) + \log k + \log L)$. Note that $k \leq L$ and L is usually much smaller than the total number of objects n. Therefore, the total cost in the last T time units is $O((N+M)(\log (N+M)+\log L)$.

In Theorem 5 we prove that the number of processed events is at most twice of the number of updates, i.e., $M \leq 2N$. For each processed update (see Algorithm 8 and Algorithm 9), we create at most two events. Note that actually at most one event will be processed among the created two events. This is because after one event is processed, the signature of the object is changed and the other event becomes invalid. However, when we process an event (see Algorithm 11), another two events will be created. Hence, the theorem is non-trivial. We show that the theorem can be proved by the geometry property of the border line.

THEOREM 5 : Given N updates, our algorithm processes at most $2N$ events. $M \leq 2N$.

PROOF. Consider the loyalty-time plane and assume that each line segment presents an update in the plane (see Figure 4.3). The border line is actually one of the connected line segments that go through the plane from left to right. For an increasing line or decreasing line, it appears in the border line at most once, while a horizontal line may appear in the border line multiple times. However, the horizontal lines are only connected with the increasing and decreasing lines in the plane. Assume that the border line has at least two line segments. Therefore, one horizontal line on the border line must connect with one increasing line or decreasing line. Let P be the number of line segments on the border line and Q be the number of increasing and decreasing lines. In the worst case, every horizontal line

segment is associated with one increasing or decreasing line. Therefore, $P \leq 2Q$. Each connected vertex on the border line presents a processed event. Consequently, we prove that $M \leq 2N$. □

Theorem 5 indicates that the number of processed events is at most twice of the number of updates. We can derive that $M = O(N)$. Moreover, $L \leq N$ because the number of objects which have updates will not larger than the number of updates. Therefore, the total cost of our algorithm in the last T time units is $O(N(\log N))$. The cost for each update is $O(\log N)$.

Proof of Optimality

THEOREM 6 : In the worst case, the lower bound cost of updating the results of a top-k loyalty query is $O(\log N)$ for each update where N is the number of updates issued in the last T time units.

PROOF. We show that it is necessary to maintain a priority queue to process the future events. An event actually means a possible result change of the loyalty query. Consider that we have n objects with the stationary state and different loyalties, and we are monitoring a top-1 loyalty query. Let o_i be the ith loyal object. The border object is o_2. Then, the object with lowest loyalty o_n has an update and the loyalty of the object becomes increasing. This creates an event because o_n is possible to become a border object in the future. After that o_{n-1} issues an update and becomes increasing and so forth. Assume loyalty of o_2 is much higher than the objects below. Therefore, we have N updates and may create N events where $N = n - 2$. Firstly, we argue that we must store all the these possible events to correctly report a future result change, otherwise we may miss a possible result change. This is because any object is possible to become a border line object if all the objects above it issues an update and become stationary state. Secondly, we

must keep the event in order so that we can efficiently know the first event in future. In other words, we employ the priority queue to maintain all the possible events. The minimum cost of maintaining an event in such data structure is $O(\log N)$. Therefore, in the worst case it takes $O(\log N)$ time to process an update. □

In the worst case, our algorithm meets the lower bound cost of the problem, thus is optimal in the worst case.

Space Analysis

Next, we investigate the space requirement of our algorithm. The space of the update queue is $O(N)$ where N is the number of updates issued in the last T time units. The size of the event queue is $O(M')$. According to the above analysis, $M' = O(N)$. The size of each subset is $O(L)$. If we do not consider the objects with maximum loyalties, then $L \leq N$. Therefore, for each top-k loyalty query, our algorithm uses $O(N)$ space.

4.4.3 Pruning

Although the algorithm is already optimal for solving the top-k loyalty queries in terms of time complexity, in this subsection we show that we can further prune some of the updates from the computation of the final results. The pruning rule can reduce both the overall computation cost and the communication cost in terms of the number of messages exchanged over distributed data streams. We first present an observation that can reduce the number of considered updates, and show how the pruning rule works over centralized data streams.

THEOREM 7 : Let o_k be the object with the minimal loyalty in A and o be any object in O. o will not be a result of top-k loyalty query in the next $(loyalty(o_k, t) - loyalty(o, t)/2$ time, where t is the current timestamp.

PROOF. Consider that o_k becomes decreasing and o becomes increasing at t. Let $d = (loyalty(o_k, t) - loyalty(o, t))/2$. o will be always below o_k in the time period $[t, t + d)$. Thus, $loyalty(o_k, t + \Delta t) > loyalty(o, t + \Delta t)$ where $0 \leq \Delta t < d$. Consequently, we prove Theorem 7. □

Based on the theorem, we may ignore some computation of $o \in O$ in time period $[t, t + d)$. We call d is the *safe time* of object o at t. To achieve this we maintain a list of echo updates for each object. In our algorithm we avoid the redundant computation for the trivial updates in the safe time. Next, we define the trivial updates.

Trivial updates. Let $U_o = u_1, u_2, ..., u_n$ be a series of echo updates of object o in the update queue U at time t. The trivial updates are a subset $U_t \subseteq U_o$ such that for each u_i in U_o, $u_i.time < t + d$.

Since the object will never be a top-k loyal object during the safe time, the trivial updates in this period will not affect the top-k results. Thus, it is not necessary to wait and process the trivial updates one by one. Instead, for all the trivial updates in U_o we only update the data structure once.

Algorithm 12 shows how we process the trivial updates. We process the updates from the list U_o. If the first update is non-trivial (line 3), we just call Algorithm 8 and process the update normally (line 7). If the update from the list is trivial, the algorithm continue to find the next update from U_o and update the loyalty of the objects according to the update u_i until the next update is non-trivial or there is no update left in the list.(see lines 3–5). Then, we process the echo update u_{i-1} with the modified loyalty of o (line 9).

The algorithm is triggered only when an echo update is processed. The cost of finding the trivial updates and updating the loyalty takes $O(|U_t|)$ time and processing of the update using Algorithm 8 takes $O(\log L)$. Therefore, $(|U_t| - 1)$

Algorithm 12: ProcessUpdateWithPruning(U_o)

```
1: Let u_i be the ith update in U_o.
2: i = 1
3: while u_i exists and u_i is trivial do
4:     Recompute the loyalty of o based on u_i.
5:     i = i + 1
6: if i = 1 then
7:     ProcessUpdate(u_1). /* Call Algorithm 8 without pruning */
8: else
9:     ProcessUpdate(u_{i-1}). /* Call Algorithm 8 with pruning */
```

trivial updates scheduled to be processed in the future are processed in $O(1)$ time for each. Thus, our pruning technique reduces the total cost of the computation.

Optimizing communication cost. In the context of many applications within distributed networked systems such as sensor networks, the communication overhead is also an important issue. Since the communication is the principal energy drain for a sensor node, reduction on the number of communication times can maximize the running time of a sensor node. A lot of research has gone into design of algorithms that are optimal with respect to the number of messages exchanged[KPKK09, CMY$^+$12, KCRR06, DKR06]. Here we consider the network messages are based on a two-way communication protocol which is commonly utilized in distributed data stream processing[CMY$^+$12, KCRR06], and we show that the communication cost can be reduced by pushing the pruning rule into the local nodes.

For each local node, we dynamically maintain the loyalties of a subset of objects O_i based on the updates and current objects' states. When a local node sending an update to the loyalty query module, the loyalty query module immediately returns

the current loyalty of the kth object $loyalty(o_k, t)$. Thus, whenever the node detects a new update, we can determine the update is trivial or not based on Theorem 7. If the update is trivial, we do not send a message to report the update and just update the loyalty of the object locally. Note that the pruning rule can still be applied to prune the trivial updates for the echo updates on the server side. We evaluate the pruning rule in the experiments and show that the it can reduce about 45% messages exchanged in the network with a large sliding window.

Discussion. One issue is that the answers to a top-k loyalty query may be sensitive to the size of a sliding window. If we choose the sliding window size too small, the answers of top-k loyalty queries are meaningless because most results may have maximum loyalties. In contrast, if the window size is too large, the query answers could not represent the timeliness of objects. Therefore, a user may determine the size of sliding windows empirically and the methodology of determining the window size could be the further work of this research.

4.5 Threshold Loyalty Queries

Different from the top-k queries, the threshold loyalty queries report the object whose loyalty is above a threshold θ. This problem is simpler because we do not need to consider the ordering the objects and each object can be considered individually. In the algorithm of threshold queries, we consider the border object BO as a dummy object with constant loyalty which is the threshold. We also maintain two sets: the top-k set A and the bottom set B, but not divide them by the different states. We also retain the event queue and update queue in the algorithm for threshold queries. An event-based algorithm is proposed in the similar way to the algorithm of top-k queries. Here, we show the differences. 1)A and B

do not need to be sorted. 2) When we process an update, the event is created if the object will potentially cross the border line. Therefore, for each update we create at most one event. 3) We do not check the set variation because each object is considered individually. 4) When we process an event of an object o, o is either moved from B to A or moved from A to B. In other words, o either becomes a query result or is removed from the result set. The details of the algorithm is straightforward and omitted.

Analysis. For the threshold queries, we do not make the set A and B sorted. Therefore, each insertion and deletion in A and B takes constant time. Let N be the number of updates issued in the last T time units, M be the size of event queue. The cost of maintaining the event queue is $O(\log(M))$. For each update we create at most one event and for each object we maintain at most one event, namely $M \leq N$. Therefore, the cost of the algorithm is $O(\log N)$ for each update. As we handle the events for multiple objects, the part $O(\log N)$ is necessary for our algorithm to maintain the priority queue. Similarly, our algorithm uses $O(N)$ space.

Pruning. The similar pruning technique proposed for top-k queries can be used for answering the threshold loyalty queries. The definition of the trivial updates is slightly different. Since we know the border line is horizontal, an increasing object in B will not cross the border line in the next $\theta - loyalty(o,t))$ time. Let safe time $d = \theta - loyalty(o,t))$. Therefore, any update in the time period $[t, t+d]$ is considered as a trivial update. Also, the technique for reducing the communication cost is still applicable for the threshold queries. We omit the details here.

4.6 Experiments

All algorithms are implemented in C++ and complied by GNU GCC. The experiments are performed on a PC with Intel Core i5 3.10GHz CPU and 8G memory under Debian Linux. We conducted extensive experiments on both real and synthetic data sets.

In the experiments, we focus on evaluating the performance of the proposed algorithm for answering top-k loyalty queries. Therefore, we do not count the cost of computing traditional query results and assume that all the inputs are in the form of object updates.

Real data. We use the global surface summary data (GSOD)[1] produced by the National Climatic Data Center (NCDC). We collect the climatic data from GSOD between 1930 to 1980. The record in the data set includes timestamp, station id, a variety of sensor data, and indictors for occurrence of fog, rain, snow, hail, thunder and tornado. We preprocess the data set to output the updates of the occurrences of rain. Therefore, we can find the rainiest stations over sliding windows by using a top-k loyalty quires. The data set consists of 7.6 million records collected from 12237 stations.

Synthetic data. In our experiment we simulate continuous time domain in discrete timestamps. Synthetic data is generated by a two state Markov chain model, which has many applications as statistical models of real-world processes. For each object o_i,

[1] ftp://ftp.ncdc.noaa.gov/pub/data/gsod/

$$Pr(Q(o_i, t+1) = 1 | Q(o_i, t) = 0) = p_i$$
$$Pr(Q(o_i, t+1) = 0 | Q(o_i, t) = 0) = 1 - p_i$$
$$Pr(Q(o_i, t+1) = 0 | Q(o_i, t) = 1) = p'_i$$
$$Pr(Q(o_i, t+1) = 1 | Q(o_i, t) = 1) = 1 - p'_i$$

p_i and p'_i are uniformly chosen from $[0, m]$ for each object. The data set consists of 10 million random updates with n objects.

Table 4.1: Experiment Parameters for Loyalty Queries

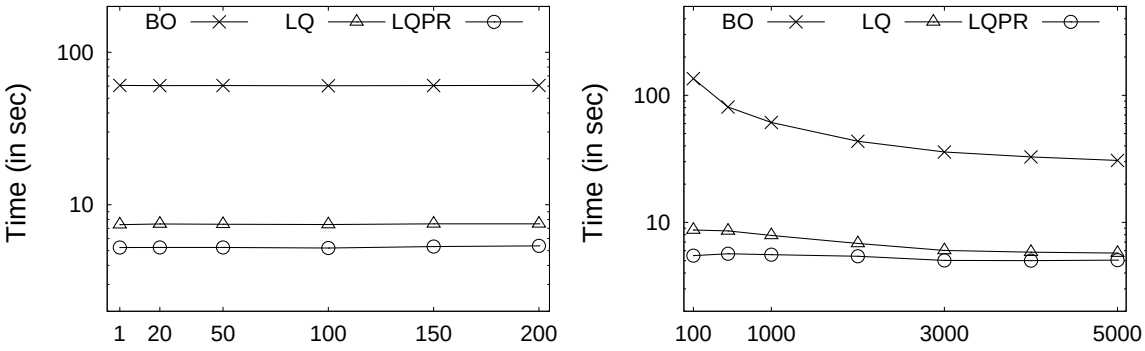
Parameter	**Range**
Sliding window size T ($\times$ 1000)	**10**, 25, 50, 75, 100
# of objects n ($\times$ 1000)	**1**, 5, 10, 15, 20
# of results k	1, 10, 20, 50, **100**, 150, 200
Probability parameter m	0.0001, **0.001**, 0.01, 0.1, 1

The table 4.1 shows the different parameters used in our experiments and the bold values are the default values used in the experiments unless mentioned otherwise.

To the best of our knowledge, we are the first to study the problem of top-k loyalty queries. We use the Bentley-Ottmann algorithm as our competitor called *BO* below. Our base loyalty query processing algorithm is called *LQ*. The loyalty query processing algorithm optimized by using the pruning rule is called *LQPR*. Note that all the figures are in the logarithmic scale except the figures for evaluating our pruning technique.

In Figure 4.4, we compare our algorithm with the Bentley-Ottmann algorithm using the real climatic data set. We process the whole data set and evaluate the running time of the algorithms. Our algorithm is extremely efficient (processing 7 million updates in seconds) and demonstrates one order magnitude improvement over the Bentley-Ottman algorithm. The algorithm with pruning rule outperforms the base algorithm in all the settings. In Figure 4.4(a) and Figure 4.4(b), we study

the effect of k and T on the algorithms. The default window size is 1000. As expected, the cost of these algorithm is not significantly effected by the variation of k and T. In Figure 4.5(b) we vary the sliding window size T from 100 to 5000. An interesting observation is that the performance of the algorithms is even better when the window size T is large. This is because the range of loyalties is large when the sliding window size is large. This makes the objects less possible to swap their orders. We observe this significantly on BO since it need to process every order change among the objects.



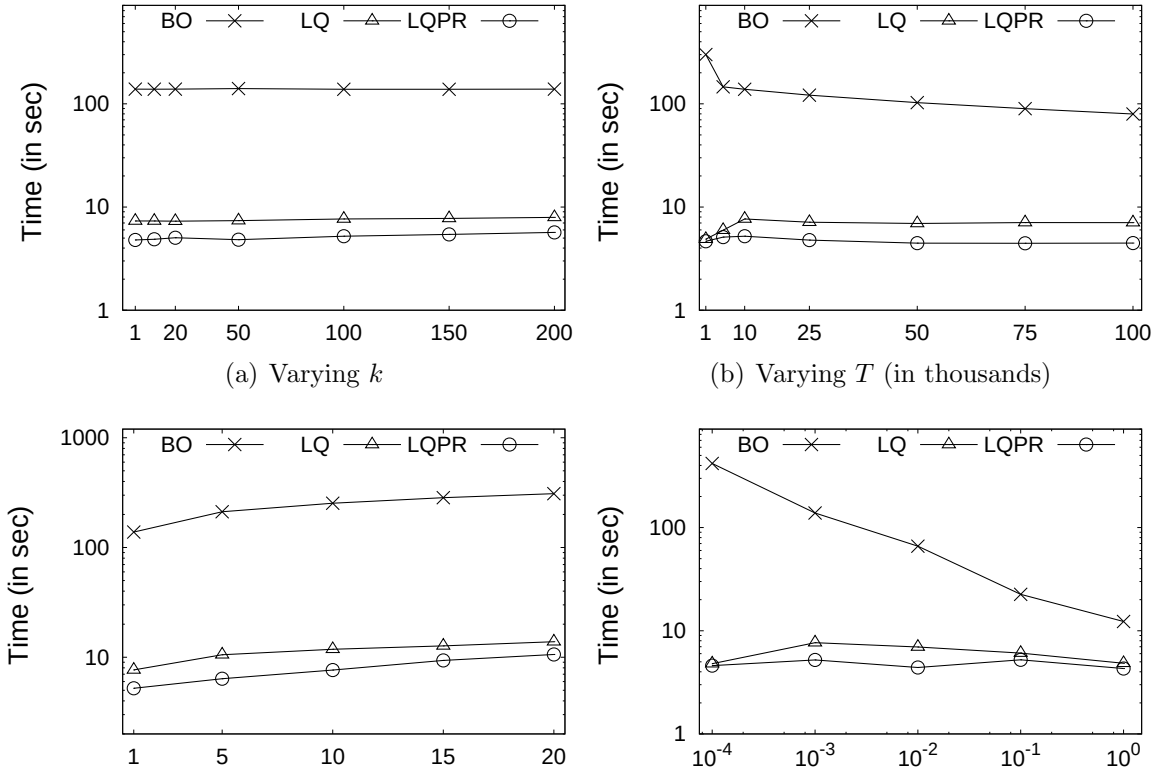
(a) Varying k (b) Varying T(in thousands)

Figure 4.4: Performance evaluation on the climatic data

In Figure 4.5, we perform experiments on syntectic data sets to conduct a more detailed evaluation. We study the effect of varying k and T in Figure 4.5(a) and Figure 4.5(b). The similar tendency can be observed on the synthetic data set. Figure 4.5(a) shows that the pruning rule does not work well when the sliding window size T is small. The reason is that the number of updates generated with certain probability in a small sliding window is small. Therefore, not many updates can be pruned according to the pruning rule.

In Figure 4.5(c) and Figure 4.5(d), we vary the number of objects n and the probability m used in generated synthetic data and study the effect on the algorithms. Figure 4.5(c) shows that the processing time of our algorithms increases

with increase in n. This is because the number of objects which have updates in the sliding window L increases with larger n. Figure 4.5(d) shows that the performance of our algorithms remains unaffected with increase in the frequency of updates, although we vary m in a very large scale. LQPR does not show a good pruning power when $m = 0.0001$ because the number of updates in the sliding window is too small so that few updates can be pruned.



(a) Varying k

(b) Varying T (in thousands)

(c) # of objects n (in thousands)

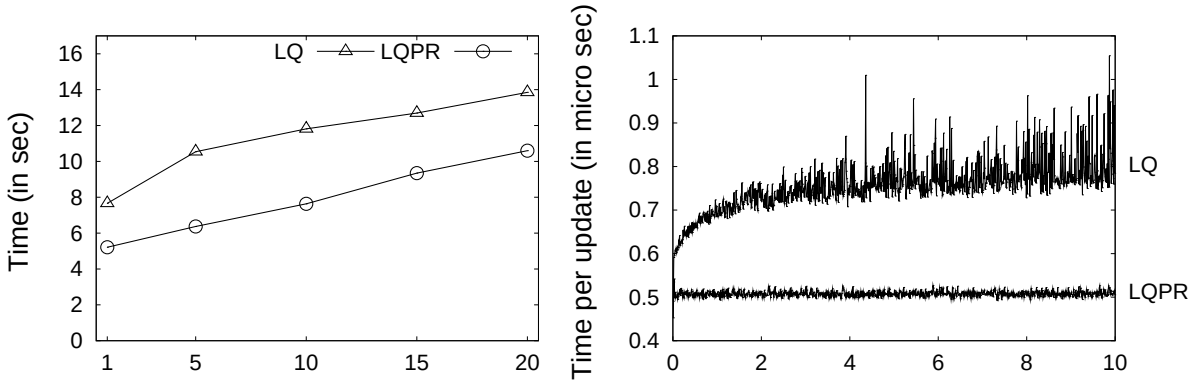
(d) Probability parameter m

Figure 4.5: Performance evaluation on the synthetic data

Next, we evaluate the efficiency and effectiveness of the pruning rule on the synthetic data set. We find that BO is about one order of magnitude slower than our algorithms. Thus, we exclude BO in the following evaluation and show the processing time in linear scale. We evaluate the total running time of both our algorithms for a centralized computation in Figure 4.6. Then, we assume that the

updates of each object are reported on an independent local client. We simulate a distributed data stream environment and conduct the experiments on evaluating the communication cost in terms of the total number of messages exchanged in the network in Figure 4.7.

Figure 4.6(a) evaluates the total processing varying the number of objects n. We find that the processing time of both algorithms increases with the increase of the number of objects. This is because the number of updates N over the sliding windows increases when n increases for the synthetic data sets. Due to the effectiveness of our pruning technique, LQPR outperforms LQ in all the cases. Figure 4.6(b) studies the average processing per update. Since the processing time of one update is too short to capture precisely, we record the average time for each batch of 10000 updates to estimate the delay per update. It shows that both of our algorithms are very efficient. LQPR can process more than 1.8 million updates per second even in the worst case on the synthetic data set. Moreover, the processing time per update of LQPR varies in a very small range, therefore has better stability than LQ. The algorithms performs slightly better at the beginning of the data sets, because we start our algorithm from scratch.

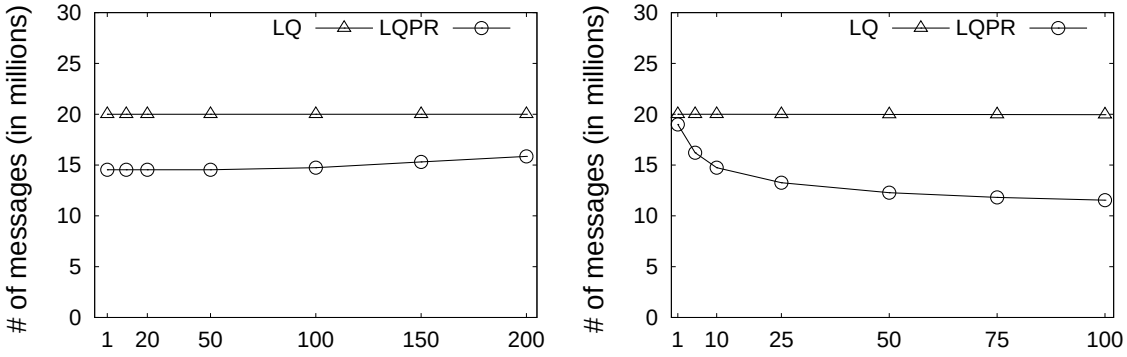


(a) # of objects (in thousands) (b) # of updates (in millions)

Figure 4.6: Efficiency evaluation for the pruning rule

In Figure 4.7(a) and Figure 4.7(b), we vary k and the window size T and evaluate

the number of messages exchanged. Figure 4.7(a) presents that we can reduce the communication cost by about 25% under the default setting. We observe that the pruning power is sightly better for a small k value, because of the higher loyalty of kth object for the small k. Figure 4.7(b) illustrates that the pruning rule works well for a larger window size. The number of messages decreases as T increases. This is due to the loyalties have large scales on large sliding windows and thus leads to a longer safe time to silence a local client. We obverse that about 45% updates be can be ignored with a sliding window of size 100 thousand.



(a) Varying k (b) Varying T(in thousands)

Figure 4.7: Evaluating communication cost

4.7 Conclusion

We introduce the loyalty queries for a variety of applications. We present efficient algorithms to answer the top-k and threshold loyalty queries. We prove the lower bound cost of the problem and present a detailed complexity analysis to show that our algorithm is optimal. We verify this by an experimental evaluation and demonstrate the efficiency of our approach.

Chapter 5

Depth-Related Problems for Top-k Queries

In this chapter, we introduce a series of depth-related problems. Our target is to provide I/O efficient algorithms for solving these problems, and we show that our techniques can be utilized to answering various top-k queries using only linear scoring functions over very large databases.

5.1 Overview

It is well known that we cannot directly compare two points in multi-dimensional space and there is no natural way to rank such data [Bar76]. Therefore, significant research attention has been put to somehow generalize the standard one dimensional ranking. As a result, the statisticians have developed the concept of *data depth* that is an attractive alternative to classical statistics [HRSS06]. The depth of a point p signifies how "deep" the point is w.r.t. a set of objects. For instance, for a one-dimensional data set, the objects with minimum and maximum values have a depth one whereas the median is the object with maximal depth. For multi-

dimensional data sets, the location depth gives a ranking mechanism where the objects are ranked based on their location depths [LPS99] (e.g., the objects on the convex hull have the depth one).

The data depth also provides the ability to quantify, analyze and visualize the multi-dimensional data sets without making prior assumptions about the probability distribution [LPS99]. It has a variety of applications such as statistical quality control, aviation safety, data analysis, gene clustering, regression analysis and query processing, to name a few. While the statisticians introduced the concept of data depth and laid its theoretical foundation, the need of developing efficient and implementable computer algorithms inspired computer scientists to study the depth-related problems. As a result, researchers from statistics community [Bar76, LPS99, Liu90, KZ10, Tuk77, RR96, RR98, RRT99] and computational geometry community [CSY84, MRR+01, KMV02, EW86, AS98] have put significant attention on solving and analysing these problems.

All of the existing algorithms assume that the data can fit into main-memory. However, this assumption does not always hold because massive data sets have become quite common in almost all disciplines ranging from financial markets to human biology to sociology. Unfortunately, the concept of data depth has not received sufficient attention from the database community. The absence of disk-based algorithms for the depth-related problems opens up a new area that needs to be explored by the database community.

Motivated by this, we propose efficient and I/O optimal disk-based algorithms for solving some important depth-related problems over large data sets. We chose these problems based on the following criteria: i) the problems should be similar to each other in nature; and ii) the problems must cover a variety of applications (e.g., the problems we study in this chapter have applications in data analysis,

outlier detection, top-k queries, clustering, Voronoi diagrams etc.). We hope that this research will motivate other database researchers to explore this area because many interesting problems remain open (see Section 5.6). Next, we introduce the problems we study in this chapter.

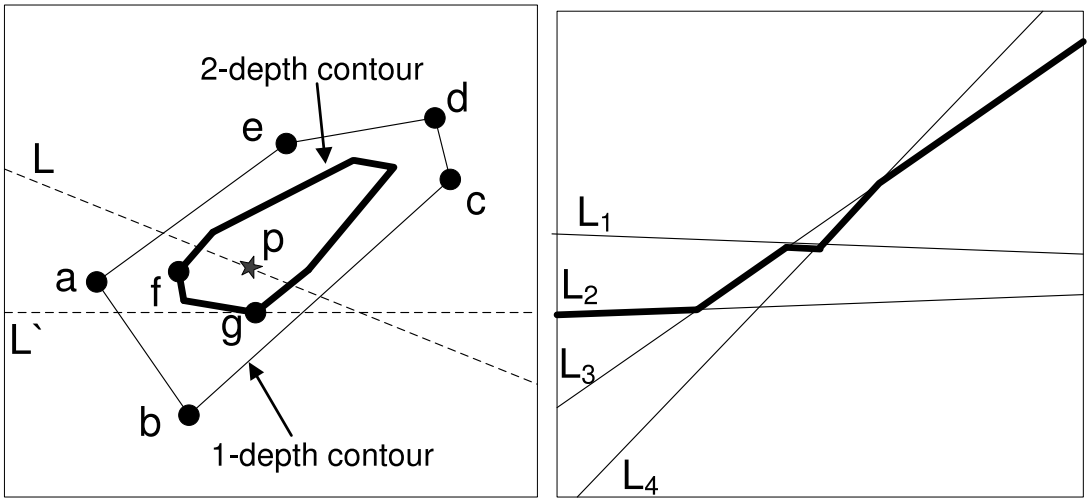
5.1.1 Problem Statements

In the past few decades, various notions of data depth have been introduced such as half-plane depth [Tuk74], convex peeling depth [Bar76], simplicial depth [Liu90] and regression depth [RH99] etc. In this chapter, we focus on the problems related to half-plane depth which is arguably the most promising depth measure [KZ10].

Definition 1 : **Half-plane depth**. Let H be a closed half-plane bounded by a line L. Half-plane depth of H is the number of data objects lying in H.

Definition 2 : **Location depth** (also called Tukey depth). Location depth of a point p (not necessarily a data object) is the minimum depth of any closed half-plane that is bounded by a line passing through p.

Fig. 5.1(a) shows 7 data objects a to g (the circles) and an arbitrary point p (the star). For a line L, a half-plane is called upper (resp. lower) half-plane of L if it is bounded from below (resp. above) by L. In Fig. 5.1(a), the upper half-plane of L contains 3 objects (c, d and e). Note that this is the minimum depth of any half-plane bounded by any line passing through p. Hence, the location depth of p is 3. Similarly, the location depth of g is 2 because the lower half-plane of L' has depth equal to 2 (it contains two objects g and b) and this is the minimum depth of any half-plane bounded by a line passing through g. Note that all the points that lie on the convex hull of the data set have depths equal to 1.



(a) k-depth contour and k-snippet (b) k-upper envelope

Figure 5.1: Illustration of k-depth contour, k-snippet and k-upper envelope

PROBLEM 1 : **k-depth contour**. Given a set of objects O and a positive integer k, find k-depth contour where k-depth contour is the set of points (not necessarily the objects) with location depth $\geq k$.

k-depth contour (also known as k-hull in computational geometry literature) is always a convex polygon. In Fig. 5.1(a), 1-depth contour and 2-depth contour are the outer and inner convex polygons, respectively. Note that the vertices of a depth contour are not necessarily the data objects. Furthermore, k-depth contour always contains $(k+1)$-depth contour.

Applications: k-depth contour has various applications such as in outlier detection [JKN98, HRSS06, KMV02, MRR$^+$01], regression analysis [KZ10], clustering [RR96] and data visualization [Tuk74]. k-depth contour is a robust tool for data picturization [Tuk74] and can be used for a visual representation of location, spread, correlation, skewness and tails of data [RRT99]. For instance, depth contours in Fig. 5.1(a) show that there is a positive correlation between x and y attributes of the objects. k-depth contour has also been used for outlier detection [JKN98] and has a nice feature that it does not rely on the probability distribution of the

underlying data set. Ruts *et al.* [RR96] demonstrated the applications of k-depth contours in clustering and discriminant analysis.

PROBLEM 2 : **k-snippet**. Given a set of objects O, find k-snippet which is a set consisting of every object $o \in O$ that either lies on the boundary of k-depth contour or outside it.

In Fig. 5.1(a), 1-snippet contains the objects a to e and 2-snippet contains every object.

Applications: k-snippet has applications in top-k queries involving linear scoring functions (e.g., weighted average where weights are allowed to be negative). Given a linear scoring function f, a top-k query returns k objects having the smallest scores where score of each object is computed using f. For a positive integer k, we say that an object o is *valuable* if it is among the top-k objects for least one linear scoring function. It is easy to show that k-snippet corresponds to the set of all valuable objects. This set is important in top-k queries because every top-m ($m \leq k$) query (involving linear scoring function) can be answered using k-snippet instead of accessing the whole database (e.g., see [XCH06]). Furthermore, in various applications, users may not be willing/able to define suitable scoring functions due to various reasons such as lack of knowledge about the data domain or due to incompatible attributes on each dimension (e.g., dollars vs inches) [FKS03]. In such cases, k-snippet returns a small subset of shortlisted objects that guarantees that no matter what linear scoring function is chosen, the top-k objects are found in k-snippet. Hence, k-snippet serves as a data summarization tool that returns only the objects that may be important for the user.

PROBLEM 3 : **k-upper envelope**. Given a set of lines $\mathcal{L}$, the upper score of a point p is the number of lines that lie strictly above it. k-upper envelope is the closure of the set of points that lie on the lines and have upper scores equal to $k-1$.

In Fig. 5.1(b), 2-upper envelope is shown using bold lines.

Applications: We demonstrate the relationship between k-depth contour and k-upper envelope in Section 5.2. k-upper envelope is also known as k-level arrangement [AdBMS98]. k-upper envelope has many important applications in domains such as Robotics and Computer Graphics [DGKS07]. It is also useful in solving various other important problems such as computing higher-order Voronoi diagrams [AdBMS98, Cha00, CE87], processing ranking queries [YAY12, SCL12a], designing data structures for halfspace range searching [CP86, Cla87], and solving hyper-plane partitioning problems such as sandwich cuts [Bor94] and weak line-separators [ERvK96].

Contributions

We make the following contributions in this chapter.

- To the best of our knowledge, we are the first to propose disk-based algorithms for a few important depth-related problems that have a wide range of applications in various domains.
- We present two efficient disk-based algorithms named SkyRider and KnightRider. We show that KnightRider algorithm is I/O optimal for k-upper envelope and k-snippet problems. It is also I/O optimal for k-depth contour when k is smaller than the minimum number of objects in any leaf node of the data structure (e.g., R-tree). Although KnightRider is not I/O optimal for k-depth contour problem when k is large, our experimental results (see table 5.2 in Section 5.5) demonstrate that its I/O cost is almost the same as the lower bound cost even when k is very large.
- We extensively evaluate our algorithms on both synthetic and real data sets.

The experimental results demonstrate that our algorithms do not only have low I/O cost but are also quite efficient. More specifically, we compare the CPU time of our algorithms with the CPU time of the best known main-memory algorithms [JKN98, EW86] assuming that their algorithms have sufficient main-memory to store the whole data set. The experimental results demonstrate that our algorithms are more than an order of magnitude faster.

The rest of this chapter is organized as follows. Section 5.2 presents the preliminaries of techniques involving in solving the problem. In Section 5.3 we first present a basic algorithm for solving k-upper envelope, called Rider algorithm, while we present an I/O optimal algorithm for this problem in Section 5.4.4. The proof of optimality is shown in Section 5.4.4. The extensive experimental results are presented in Section 5.5. Then, a conclusion is given in Section 5.6.

5.2 Preliminaries

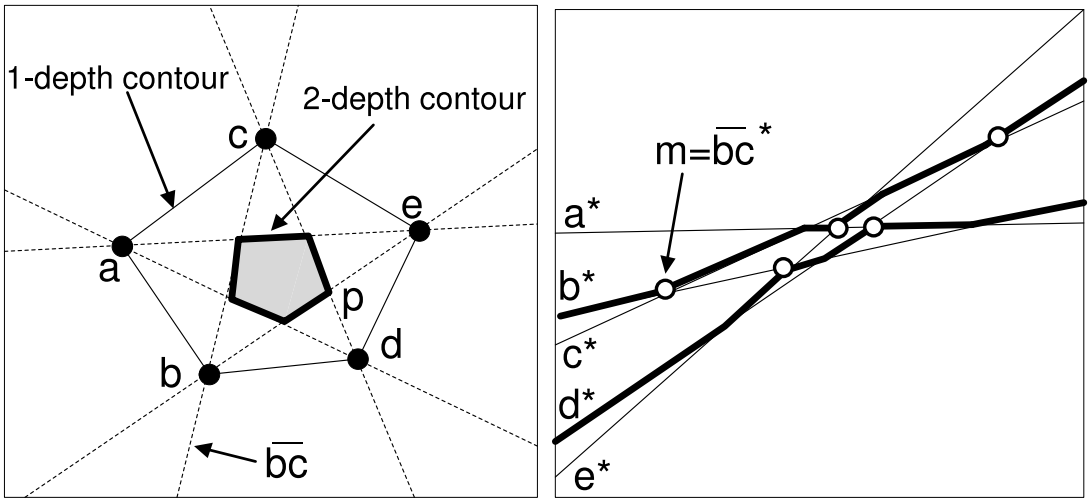
5.2.1 Computing k-depth Contour in Dual Space

In this section, we show that k-depth contour can be computed by mapping the objects to a dual space. Throughout this chapter, O denotes the set of data objects and n denotes the number of objects in O. First, we define a few concepts.

Definition 3 : **k-divider**. k-divider is a line that passes through at least 2 objects and contains exactly $k-1$ objects on one side and at most $n-k-1$ objects on the other side.

Definition 4 : **k-half-plane**. A half-plane that is bounded by a k-divider and contains at most $n-k+1$ objects is called a k-half-plane.

The k-depth contour can be obtained by taking the intersection of all k-half-planes [MRR+01]. We use $\overline{pq}$ to denote a line that passes through points p and q. $\overrightarrow{pq}$ denotes the upper half-plane defined by $\overline{pq}$ and $\underrightarrow{pq}$ denotes its lower half-plane. Consider the example of Fig. 5.2(a) that shows 5 data objects a to e. The line $\overline{bc}$ is a 2-divider and $\underrightarrow{bc}$ is a 2-half-plane. Note that all the broken lines shown in Fig. 5.2(a) are 2-dividers. In Fig. 5.2(a), 2-depth contour (the shaded polygon) is obtained by taking the intersection of the 2-half-planes $\underrightarrow{cd}$, $\underrightarrow{ae}$, $\underrightarrow{bc}$, $\overrightarrow{ad}$ and $\overrightarrow{be}$.



(a) Primal Space (b) Dual Space

Figure 5.2: Illustration of k-depth contour and related concepts

Definition 5 : **Lower (upper) hull**. Let Z be the convex hull of a set of points P. The lower (resp. upper) hull of P is the set of edges of Z that lie on or below (resp. on or above) every point $p \in P$.

In Fig. 5.2(a), the convex hull of the set of objects a to e is the outer polygon. The upper hull consists of the edges ac and ce. Similarly, the lower hull is the set of edges ab, bd and de.

Dual Mapping. A point $p = (u, v)$ in primal space is mapped to a line $p^* : y = ux + v$ in the dual space and a line $L : y = -ux + v$ in the primal space is mapped

to a point $L^* = (u, v)$ in the dual space (e.g., see [Mat02]). The transformation from/to primal to/from dual is denoted by using a superscript *, e.g., a point p in primal is denoted as p^* in dual and a line L in primal is denoted as L^* in dual. It is easy to see that $p^{**} = p$.

Fig. 5.2(b) shows the points and lines of Fig. 5.2(a) mapped to a dual space. For example, the objects a to e are mapped to the lines a^* to e^* and the line $\overline{bc}$ is mapped to a point m.

This dual mapping has a number of interesting properties: 1) A point z lies below (resp. above) a line L if and only if the line z^* lies below (resp. above) the point L^* (e.g., in Fig. 5.2(a), e lies below $\overline{bc}$ and, in Fig. 5.2(b), e^* lies below $\overline{bc}^*$; 2) two lines L_1 and L_2 intersect at a point z if and only if the line z^* passes through the points L_1^* and L_2^* (e.g., in Fig. 5.2(b), the lines b^* and c^* intersect each other at a point $\overline{bc}^*$ and, in Fig. 5.2(a), the line $\overline{bc}$ passes through the points b and c.

Next, we show that k-depth contour of a set of objects O can be computed by mapping all objects in dual space and then computing k-upper and *k-lower* envelopes.

Definition 6 : **k-upper (lower) envelope**. Consider a set of lines $\mathcal{L}$. Upper (resp. lower) score of a point p is the number of lines above (resp. below) p. k-upper (resp. lower) envelope is the closure of the set of points that have upper (resp. lower) score equal to $k - 1$.

In Fig. 5.2(b), the upper score of point m is 1 and its lower score is 2. 2-upper envelope and 2-lower envelope are shown using bold line segments. Note that k-upper envelope is the same as $(n - k + 1)$-lower envelope.

Definition 7 : **Convex vertices**. Let LH be the lower hull of the points on k-upper envelope and UH be the upper hull of the points on k-lower envelope. The vertices of LH and UH are called the convex vertices.

Fig. 5.2(b) shows all the convex vertices as hollow circles. Note that convex vertices correspond to lower (resp. upper) hull of *all points* (not only the vertices) of k-upper (resp. lower) envelopes (e.g., see the convex vertices of 2-lower envelope in Fig. 5.2(b)). For clarity of presentation, the lower hull of k-upper envelope and the upper hull of k-lower envelope are not shown.

Assume that the set of objects O is mapped to a dual space. Note that each convex vertex v in dual space corresponds to a k-divider in the primal space. For example, the vertex $m = \overline{bc}^*$ in Fig. 5.2(b) corresponds to the 2-divider $\overline{bc}$ in Fig. 5.2(a). All convex vertices shown in Fig. 5.2(b) correspond to 2-dividers in Fig. 5.2(a) (the broken lines). Since k-depth contour can be computed once all k-dividers are known, it is easy to compute k-depth contour once all convex vertices are determined. Below, is a step-by-step description of computing k-depth contour by using dual mapping.

1. Map all objects in O to lines in a dual space.

2. Compute k-upper envelope and k-lower envelope on these lines and determine the convex vertices (by computing lower and upper hulls).

3. Map the convex vertices to lines in primal space. Use these k-dividers to obtain the k-depth contour.

5.2.2 Problem Settings and Assumptions

We assume that all the data objects are indexed by a branch-and-bound data structure such as R-tree and Quad-tree etc. Although our algorithms can be applied on any branch-and-bound index, in this chapter, we use R-tree due to its simplicity and popularity.

Like most of the existing techniques (e.g., see [EW86]), we assume that no two lines are parallel and no three lines are concurrent when the objects are mapped

to the dual space. We remark that this assumption is made only for the ease of presentation. Later in Section 5.4.5, we show that such situations can be handled easily.

As discussed earlier, k-depth contour can be computed by obtaining k-upper and k-lower envelopes in the dual space. We also show (Lemma 13 and 14) that the lines in dual space that overlap with k-upper and k-lower envelope correspond to the objects in k-snippet. Therefore, computing k-upper (lower) envelope is the key component of solving these problems. Hence, we focus on presenting the techniques for k-upper envelope computation (the techniques to compute k-lower envelope are similar). In Section 5.4.4, we present techniques that are specific to compute k-depth contour and k-snippet.

5.3 The SkyRider Algorithm

5.3.1 The Rider: An Elementary Algorithm

In this section, we present a basic disk-based algorithm for computing k-upper envelope. This is called Rider algorithm and is also used as a subroutine in our main algorithms, SkyRider and KnightRider.

Intuitive description. Assume that all objects in O have been mapped to lines in a dual space. Let origin line L_o be the line with the k-th smallest slope. Let destination line L_d be the line with the k-th largest slope. Assume that all lines are roads and a bike rider starts traveling from the left most point on the origin line (i.e., at $x = -\infty$). The rider always travels towards its right (i.e., towards increasing value of x). Whenever it reaches at an intersection of two lines, it makes a turn. The rider keeps traveling until it reaches the right most point of the destination line (i.e., at $x = \infty$). It is easy to verify that the path that the rider

travels on corresponds to the k-upper envelope. The proof is straightforward and intuitive and is omitted.

In Fig. 5.2(b), assuming $k = 2$, the origin line is b^* and the destination line is d^*. The rider starts from the left most point of b^* and travels towards right. When he reaches at the intersection m, he makes a turn and continues traveling on c^*. The algorithm continues until the rider reaches the right most point of d^*. The path (shown in bold) is the path traveled by the rider and corresponds to k-upper envelope.

Algorithm 13 presents a more formal description.

Algorithm 13: Rider Algorithm

1 Find the origin line and call it L_c. Set the current location loc as the point on L_c with $x = -\infty$;

2 Among the lines that intersect L_c on right of loc, find the line L' that is the first line to intersect L_c. Let z be the intersection of L' and L_c. Add z to k-upper envelope;

3 Terminate the algorithm if there does not exist any such L' at line 2. Otherwise, set the current location as z (i.e., $loc \leftarrow z$) and current line as L' (i.e., $L_c \leftarrow L'$) and go to line 2;

Implementing rider algorithm on disk-resident data. In this section, we show how to implement the rider algorithm when the data objects (i.e., the lines in dual space) cannot fit into main-memory and are indexed by a branch-and-bound data structure (i.e., R-tree). More specifically, we describe how to implement the first two lines of the algorithm (the third line is self describing).

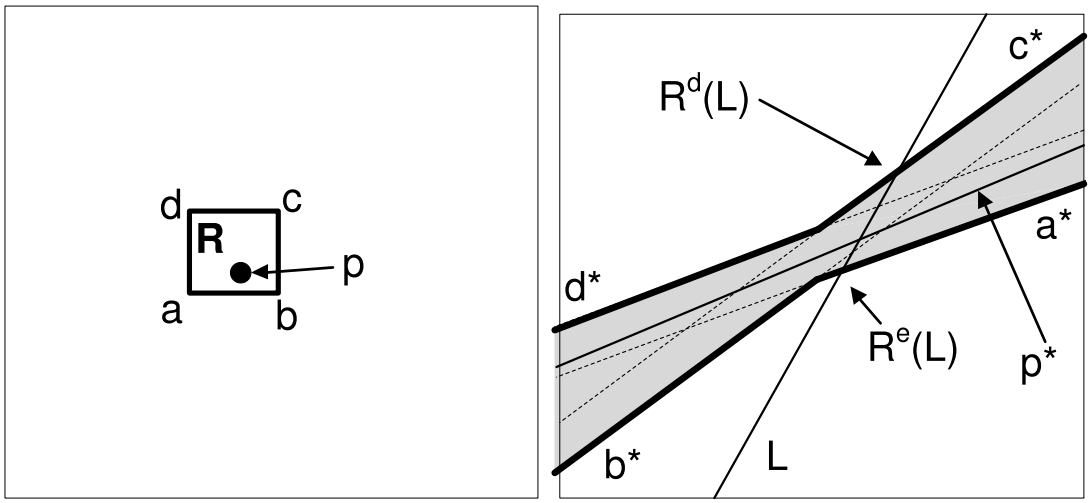
Line 1

Note that the origin line is the line with k-th smallest slope and it corresponds to the object in primal space that has the k-th smallest x-coordinate value. Such object can be easily found using a best-first search algorithm on R-tree.

Line 2

Before we present the details, we discuss how a rectangle in primal space is mapped to dual space and define a few terms and notations.

Spectrum of a rectangle. Consider the rectangle R shown in Fig. 5.3(a). In Fig. 5.3(b), we map the four corners of the rectangle (a to d) to four lines in dual space (a^* to d^*). The 1-upper envelope and 1-lower envelope of these four lines are shown using bold lines. The space between the 1−upper envelope and 1-lower envelope is called the *spectrum* of rectangle R and is denoted as R^*. In Fig. 5.3(b), the spectrum of R is shown shaded. It is easy to verify that, for any point $p \in R$, its corresponding line p^* in dual space lies entirely in R^*. In Fig. 5.3, the point p lies in the rectangle R and p^* lies in R^*.



(a) Primal Space (b) Dual Space

Figure 5.3: Mapping a rectangle to dual space

Entering (departing) junctions. Given a line L, assume that a rider starts traveling on L from its left most point towards its right most point. The point when the rider enters the spectrum of a rectangle R for the first time is called the entering junction of L w.r.t. R and is denoted as $R^e(L)$. Similarly, the point when the rider leaves the spectrum of R for the last time is called the departing junction of L w.r.t. R. Note that the entering and departing junctions of a line L w.r.t. R can be easily computed using the intersections of L with the dual lines corresponding to the corners of R. In Fig. 5.3(b), the entering junction of L w.r.t. R is the intersection of L and a^* and the departing junction is the intersection of L and c^*.

LEMMA 4 : For each point $p \in R$, the intersection of p^* with a line L (in dual space) is always between the entering junction $R^e(L)$ and departing junction $R^d(L)$.

The proof is straightforward and is omitted. In Fig. 5.3, $p \in R$ and its dual line p^* intersects with L between the entering and departing junctions of L w.r.t. R. Hereafter, whenever clear by context, we just use the terms entering/departing junction of L or entering/departing junction of R to denote entering/departing junction of L w.r.t. R.

Based on Lemma 4, we present a branch-and-bound algorithm that efficiently implements line 2 of Algorithm 13. Recall that line 2 requires finding the first line L' that intersects L_c on the right of the current location *loc*. The following two pruning rules prune the intermediate entries of the R-tree that cannot contain such line L'.

PRUNING RULE 1 : An intermediate node R can be pruned if departing junction $R^d(L_c)$ does not lie on the right of *loc*.

PROOF. Since $R^d(L)$ does not lie on the right of *loc*, $R^e(L_c)$ cannot lie on the right of *loc*. Therefore, for every point $p \in R$, p^* cannot intersect L_c on the right

of *loc*. Hence, R can be pruned. □

PRUNING RULE 2 : Given the current line L_c and two rectangles R_1 and R_2, the rectangle R_1 can be pruned if i) entering junction of R_2 lies on the right of *loc* and ii) entering junction of R_1 lies on the right of departing junction of R_2.

PROOF. Since the entering junction of R_2 lies on right of *loc*, it guarantees that there exists at least one object $o \in R_2$ such that o^* intersects L_c on the right of *loc*. Furthermore, the entering junction of R_1 lies on the right of departing junction of R_2. This guarantees that, for every point $p \in R_1$, there exists an object $o \in R_2$ such that the intersection of o^* with L_c lies on the left of the intersection of p^* with L_c. Hence, R_1 can be pruned. □

Algorithm 14 shows the detailed implementation of line 2 of the rider algorithm. $iBest$ stores the current best intersection and is initialized to a point on L_c that lies on $x = \infty$. A min-heap H is initialized with root of the R-tree. The key of each element e in heap is the entering junction of e (line 13). The heap is implemented such that its top-element always corresponds to the element with the left most entering junction (i.e., its entering junction lies on the left of the entering junction of every other element). If the de-heaped entry e is a data object then e^* is the first line to intersect L_c on right of *loc*. This is because i) at line 10, every line that does not intersect L_c on right of *loc* is pruned and ii) the heap guarantees that all other entries in the heap have entering junctions on the right of entering junction of e. Hence, the intersection of e^* and L_c is computed and returned (line 7).

If de-heaped entry e is an intermediate or leaf node then, for each child c of e, c is pruned (see line 10) if the departing junction of c does not lie on the right of *loc* (pruning rule 1). The child c can also be pruned if the entering junction of c lies on the right of $iBest$ (line 12). This is an indirect application of pruning rule 2.

Algorithm 14: GetIntersection(L_c, loc)

Input : L_c: the current line; loc: the current location

Output : z: the next intersection; L': first line intersecting L_c

```
1  iBest ← the point on L_c with x = ∞;
2  Initialize a min-heap H with root of the R-tree;
3  while H is not empty do
4      de-heap an entry e;
5      if e is a data object then
6          z ← intersection of e* with L_c;
7          Return z and e*;
8      for each child c of e do
9          if departing junction of c does not lie on right of loc then
10             continue;                    /* Pruning Rule 1 */;
11         if entering junction of c lies on right of iBest then
12             continue;                    /* Pruning Rule 2 */;
13         enheap c in H with key set to entering junction of c;
14         if entering junction of c lies on the right of loc then
15             if departing junction of c lies on left of iBest then
16                 iBest = departing junction of c;
17 Return NULL;
```

If the child c cannot be pruned using the above two conditions, we insert it in the heap (line 13). Then, we check whether $iBest$ should be updated or not. $iBest$ needs to be updated if both of the following are true: i) entering junction of c lies on the right of loc; and ii) departing junction of c lies on the left of $iBest$. In this case, $iBest$ is updated to the departing junction of c (line 16). Note that if the first condition does not hold then we cannot guarantee that there exists an object $o \in c$ such that o^* intersect on the right of loc.

The while loop terminates when the heap becomes empty. This implies that all lines intersect L_c on left of loc. Hence, the algorithm returns NULL.

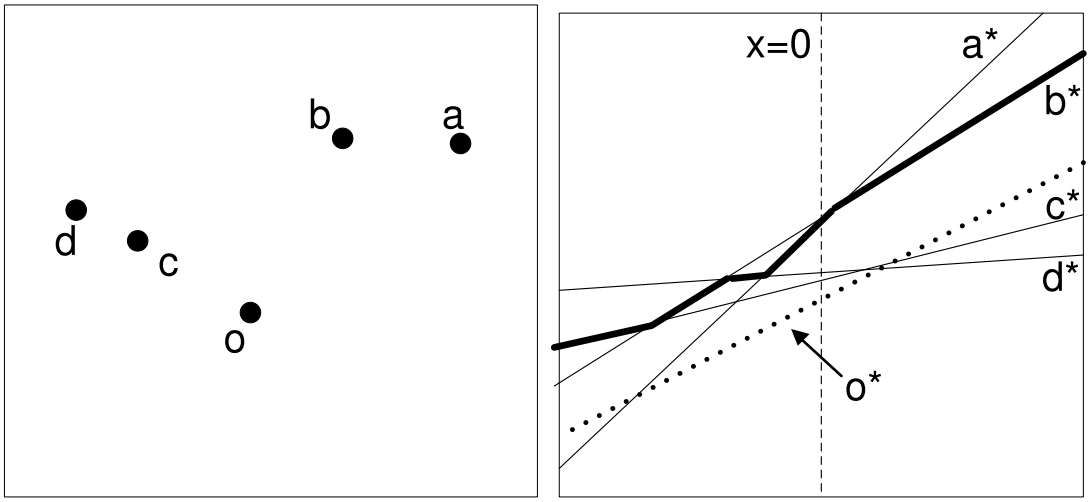
5.3.2 SkyRider: An I/O Economical Version of Rider Algorithm

A major problem with the rider algorithm is that it calls Algorithm 14 as many times as the number of vertices of the k-upper envelope. This results in a very high I/O cost because the rider algorithm accesses the R-tree each time Algorithm 14 is called. Next, we present an observation that significantly reduces the I/O cost of the rider algorithm.

Fig. 5.4 shows a few data objects in primal space and their corresponding lines in the dual space. In Fig. 5.4(b), the 2-upper envelope is shown using bold lines. Note that the line o^* (the dotted line) does not contribute to 2-upper envelope and can be pruned. In the following, we identify conditions required to prune such a line o^*. We divide the dual space into two regions: the space for which $x \geq 0$ is called positive space and the space for which $x \leq 0$ is called negative space. In Fig. 5.4(b), the space on right (resp. left) of the broken vertical line is positive (resp. negative) space.

LEMMA 5 : A line L cannot be a part of k-upper envelope in the positive space if there exist at least k lines that have y-intercepts greater than the y-intercept of L and slopes greater than the slope of L.

PROOF. A line L cannot be a part of the k-upper envelope in the positive space if there exist at least k lines that lie strictly above L in the positive space. Any line L' lies strictly above L in the positive space if L' has y-intercept greater than the y-intercept of L and L' has slope greater than the slope of L (e.g., in Fig. 5.4(b), both the lines a^* and b^* lie strictly above o^* in the positive space). Hence, L cannot be a part of the k-upper envelope in the positive space if there exist at least k such lines. □



(a) Primal Space (b) Dual Space

Figure 5.4: Pruning irrelevant data points

LEMMA 6 : A line L cannot be a part of k-upper envelope in the negative space if there exist at least k lines that have y-intercepts greater than the y-intercept of L and slopes smaller than the slope of L.

The proof is similar to the proof of Lemma 5 and is omitted. In Fig. 5.4(b), both of the lines c^* and d^* have y-intercepts greater than the y-intercept of o^* and slopes

smaller than the slope of o^*. Hence, o^* is not a part of 2-upper envelope in the negative space.

A line o^* that satisfies the conditions in both Lemma 5 and Lemma 6 is not required to compute k-upper envelope and can be pruned. Based on Lemma 5 and Lemma 6, we define pruning condition in primal space to prune the objects for which their corresponding lines in dual can be pruned. For a point p, let $p[x]$ and $p[y]$ denote its x and y coordinate values in the primal space.

LEMMA 7 : An object o can be pruned (i.e., its dual line cannot be a part of k-upper envelope) if both of the following conditions hold: i) there exist at least k objects such that for each such object r, $r[x] > o[x]$ and $r[y] > o[y]$; and ii) there exist at least k objects such that for each such object s, $s[y] > o[y]$ and $s[x] < o[x]$.

PROOF. For an object o mapped to a line o^* in dual space, $o[x]$ corresponds to the slope of o^* and $o[y]$ corresponds to the y-intercept of o^*. If the first condition is satisfied then o^* satisfies Lemma 5 and hence o^* is not a part of k-upper envelope in the positive space. If the second condition is satisfied then o^* satisfies Lemma 6 and cannot be a part of k-upper envelope in the negative space. Hence, o can be pruned. □

In Fig. 5.4(a), the object o can be pruned because 2 objects (a and b) satisfy the first condition and 2 objects (c and d) satisfy the second condition.

Note that the conditions defined in Lemma 7 have similarity to the concept of *dominance* [PTFS05]. An object o' dominates another object o if o' is preferable to o on every attribute. A k-skyband [PTFS05] consists of every object that is dominated by at most $k-1$ objects. Assume that the preference function f_1 prefers larger values on both coordinates x and y. Then, an object o satisfies the first condition of Lemma 7 if it is dominated by at least k objects according to f_1

(i.e., in Fig. 5.4(a), o is dominated by a and b according to f_1). In other words, o satisfies the first condition if it is not a k-skyband object according to preference function f_1. Similarly, assume that another preference function f_2 prefers larger values on y-coordinate and smaller values on x-coordinate. The object o satisfies the second condition of Lemma 7 if o is not a k-skyband according to f_2.

The above discussion implies that the objects that are k-skyband objects according to f_1 or f_2 can be used to correctly compute the k-upper envelope. Hence, SkyRider algorithm first computes these two k-skybands using BBS [PTFS05]. Note that BBS stores the k-skyband objects in a main-memory R-tree. This R-tree is then used by the rider algorithm (Algorithm 13) to compute the k-upper envelope.

5.4 KnightRider: An I/O Optimal Algorithm

5.4.1 Outline

For a node R of the R-tree, the cardinality of R is the number of data objects contained in the sub-tree of this node. As described earlier, each rectangle in primal space is represented as a spectrum in dual space. Each spectrum is assigned a number that denotes the cardinality of the rectangle. Starting from the root node, we iteratively explore the entries of R-tree. At each step, we select one or more entries of R-tree and, for each such entry, we insert its children (i.e., corresponding spectrums) in a queue. Then, using the spectrums in the queue, we compute two approximations of k-upper envelope named *best envelope* and *worst envelope* (to be defined later) such that k-upper envelope lies on or below best envelope and lies on or above worst envelope. To achieve optimality, we employ a certain access order and pruning rule to ensure that the algorithm accesses only the entries that must

be accessed. Terminating condition ensures that the best envelope is the same as k-upper envelope when the algorithm terminates.

5.4.2 Best (Worst) Envelope

Top-layer of a spectrum R^* is the upper boundary of the spectrum and *bottom-layer* of a spectrum is the lower boundary of the spectrum. In Fig. 5.3(b), the top-layer of R^* is the upper boundary shown in bold and the bottom-layer is the lower boundary (also shown in bold).

DEFINITION 8 : **Best k-upper envelope**. Assume a set of top-layers where each layer is assigned a number that denotes the cardinality of the corresponding spectrum. For a point p, *upper cardinality* (resp. *lower cardinality*) is the sum of the cardinalities of all top-layers that lie above (resp. below) it. Best k-upper envelope is the closure of the set of points that lie on top-layers, and have upper cardinality at most $k-1$ and lower cardinality at most $n-k$.

The worst k-upper envelope is defined similarly with the only difference that bottom-layers are used instead of top-layers. Unless specifically mentioned, hereafter we use the term best (resp. worst) envelope to refer to best (resp. worst) k-upper envelope. In Fig. 5.5, three spectrums R_1^* (the dotted spectrum), R_2^* (the shaded spectrum) and R_3^* (the spectrum with broken lines) are shown with cardinalities 3, 4 and 2, respectively. Assuming that $k = 2$, the best envelope and worst envelope are shown using bold lines (note that $n = 9$).

Assuming that higher the k-upper envelope is the better it is, the best (resp. worst) envelope denotes the best (resp. worst) possible k-upper envelope, i.e., k-upper envelope always lies between the best and worst envelope. We give an intuitive explanation of the proof. Note that every object $o \in R$ has its dual line o^*

inside R^*. This implies that o^* is on or below the top-layer of R^* and on or above the bottom-layer of R^*. Hence, it is easy to verify that the k-upper envelope lies on or below the best envelope and lies on or above the worst envelope.

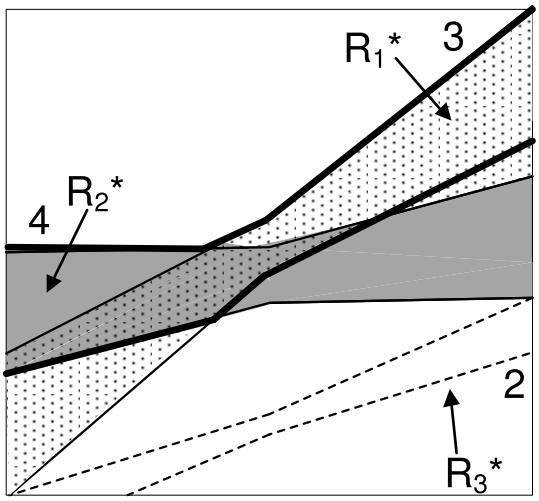


Figure 5.5: Best and worst k-upper envelopes are shown using bold lines

Next, we define a condition that prunes the entries of R-tree that are not required to compute k-upper envelope. We say that a spectrum R^* overlaps an envelope if at least one point of the spectrum lies on or above the envelope.

PRUNING RULE 3 : An entry R can be pruned if its spectrum R^* does not overlap the worst envelope.

PROOF. Let p be a point on k-upper envelope. By definition, upper score of p is $k-1$. As stated earlier, k-upper envelope always lies on or above the worst envelope. Since R^* does not overlap the worst envelope, for every object $o \in R$, o^* passes below p. Hence, o^* does not affect the upper score of p and can be pruned. □

In Fig. 5.5, R_3 can be pruned because its spectrum does not overlap the worst envelope.

Algorithm 15 provides the details of an I/O optimal algorithm to compute k-upper envelope. The algorithm initializes two sets Q and S where Q contains the

Algorithm 15: KnightRider Algorithm

```
1  initialize a queue Q with root of the R-tree;
2  insert top and bottom layers of root of R-tree in S;
3  Compute best and worst envelope using S;
4  while Q is not empty do
5      for each entry e in Q do
6          remove top and bottom layers of e from S;
7          for each child c of e do
8              if c cannot be pruned using pruning rule 3 then
9                  insert top and bottom layers of c* in S;
10     recompute best and worst envelopes using S;
11     remove entries from S using pruning rule 3;
12     Q ← intermediate or leaf nodes that contribute a line to best envelope;
13 Return the best envelope;
```

root of the R-tree and S contains top and bottom layers of the root node. Throughout the execution of the algorithm, Q maintains the entries of R-tree that are to be opened in next iteration and S maintains the spectrums that are used to compute best and worst envelopes. For each entry e in Q, the algorithm first removes its corresponding top and bottom layers from S (line 6). Then, the algorithm uses pruning rule 3 and inserts every child c in S that cannot be pruned (line 9).

After every entry e of Q is processed as described above, the algorithm recomputes the best and worst envelopes using the updated S (line 10). Since the worst envelope has been recomputed, there may be some entries in S that can be pruned using pruning rule 3. Hence, the algorithm prunes such entries (line 11). Then, the algorithm chooses the entries that are to be opened in next iteration. More specifically, during the computation of the best envelope at line 10, the algorithm keeps track of each entry e such that e^* contributes a line to the best envelope (in Fig. 5.5, R_1^* and R_2^* contribute lines to the best envelope whereas R_3^* does not). Among these entries, the entries that are intermediate or leaf nodes of the R-tree are inserted in Q and will be accessed in next iteration (line 12). The while loop terminates when no such entry e is found in Q. It can be shown that the best envelope at this stage is the same as k-upper envelope (see Lemma 8).

Note that the best (worst) envelopes can be easily computed using a slightly modified version of the rider algorithm. More specifically, at each intersection, the rider decides whether to make a turn or not based on which of the two lines satisfies the definition of best (worst) envelope.

5.4.3 Proof of Correctness

LEMMA 8 : Best envelope is the same as k-upper envelope if i) no entry e^* that overlaps the best envelope is pruned; and ii) every entry e for which e^* contributes

a line to the best envelope is a data object.

PROOF. First, we show that our algorithm satisfies both conditions when it terminates.

i) Every entry e^* that overlaps the best envelope is guaranteed to overlap the worst envelope. Hence, the pruning rule 3 cannot prune any such entry e.

ii) If there exists such an entry e^* which is not a data object, it will be inserted in Q (at line 12) which contradicts that the algorithm has been terminated.

Now, we show that the best envelope is the same as k-upper envelope when the algorithm terminates. By definition, every point p on the best envelope has upper cardinality at most $k-1$ and lower cardinality at most $n-k$. We show that every such point p is also a point on k-upper envelope, i.e., upper score of p is equal to $k-1$. Since the number of lines that lie below p is at most $n-k$, the number of lines that lie on or above p is at least k. Since p lies on the best envelope, the cardinality of line e^* passing through p is one (e^* is a data object). Hence, the number of lines above p is at least $k-1$. By definition of best envelope, the upper cardinality of p is at most $k-1$. Hence, the number of lines above p is exactly equal to $k-1$. Hence, p is a point on k-upper envelope. □

5.4.4 Proofs of Optimality

We focus our discussion for $k \leq n/2$. This is because k-depth contour is empty (and is not required to be computed) when $k > n/2$ [RR96]. Similarly, when $k > n/2$, k-snippet contains all of the data objects and does not require computation. Finally, k-upper envelope is the same as $(n-k+1)$-lower envelope [Mat02]. Hence, if $k > n/2$, the k-upper envelope can be obtained by computing k'-lower envelope where $(k' = n-k+1) \leq n/2$.

Optimality for k-upper envelope. We prove the optimality by showing that i)

every entry R must be accessed if R^* overlaps the k-upper envelope (Lemma 9); and ii) our algorithm accesses only the entries for which their spectrums overlap the k-upper envelope (Lemma 11).

LEMMA 9 : Every entry R must be accessed if its spectrum R^* overlaps k-upper envelope.

PROOF. There may be two cases: i) R^* lies completely above k-upper envelope (i.e., every point of R^* lies above k-upper envelope); ii) R^* does not lie completely above k-upper envelope. We show that the first case can never happen (Lemma 10). For the second case, we show that such R^* must be accessed. We prove this by contradiction. Assume that k-upper envelope can be computed without accessing such a rectangle R. Let E_R denote the part of k-upper envelope that is contained in the spectrum R^*. For any point p on E_R, its upper score depends on the locations of objects in R, e.g., for each object $o \in R$, o^* may or may not lie above p. Hence, it cannot be determined whether p is a point on k-upper envelope unless such rectangle R is accessed. □

LEMMA 10 : There does not exist any rectangle R such that R^* lies completely above k-upper envelope.

PROOF. As stated earlier in Section 5.3, two lines (origin line L_o and the destination line L_d) are always the lines on k-upper envelope. Origin line L_o has k-th smallest slope and the destination line L_d has k-th largest slope. Consider a rectangle R and its spectrum R^* (both shown in Fig. 5.6(a)). For a line L, let $L.slope$ denote its slope. R^* cannot lie completely above k-upper envelope unless both of the following hold: i) a^* lies above L_d as a^* tends to ∞ (i.e., $a^*.slope \geq L_d.slope$); and ii) b^* lies above L_o as b^* tends to $-\infty$. (i.e., $b^*.slope \leq L_o.slope$).

We show that these two conditions cannot hold simultaneously. Without loss of generality, assume that the first condition holds, i.e., $a^*.slope \geq L_d.slope$. We prove by contradiction that the second condition cannot hold. Note that $L_o.slope \leq L_d.slope$ because $k \leq n/2$ and L_o is k-th smallest slope whereas L_d is k-th largest slope. This implies that $a^*.slope \geq L_o.slope$. This implies that if the second condition holds then $a^*.slope \geq b^*.slope$. However, we know that $a^*.slope < b^*.slope$ because[1] a^* corresponds to the lower left corner of R in primal space and b^* corresponds to the lower right corner of R (see Fig. 5.6(a)).

□

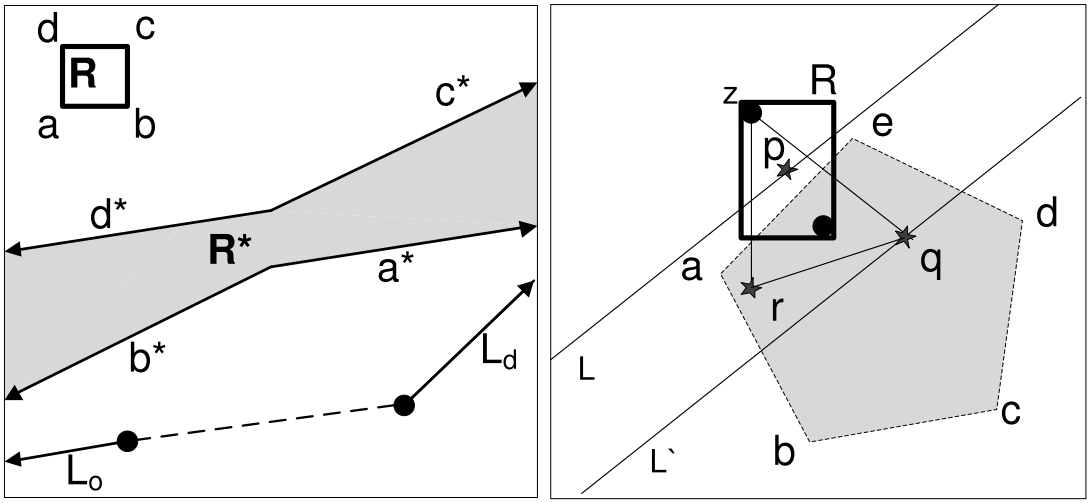
LEMMA 11 : Our algorithm accesses only the entries for which their spectrums overlap k-upper envelope.

PROOF. Note that at line 12 of the algorithm, we choose the spectrums that must be accessed in each round. We always choose a spectrum R^* that contributes a line to the best envelope. Since k-upper envelope is always on or below the best envelope, it implies that the chosen spectrum R^* has at least one point that lies on or above k-upper envelope. Hence, R^* overlaps k-upper envelope. □

Optimality for k-depth contour. To compute k-depth contour, we need to compute both k-upper and k-lower envelopes. It is easy to modify Algorithm 15 such that it computes both k-upper and k-lower envelopes in one traversal of R-tree. Specifically, a pruning rule similar to pruning rule 3 is defined for pruning the entries that are not required to compute k-lower envelope. Then, during each iteration, best and worst k-upper and k-lower envelopes are computed and the entries that are not required for computing both k-upper and k-upper envelopes

[1]Note that the proof does not hold if $a^*.slope = b^*.slope$, i.e., if entry R is a vertical line. In such special case, during the execution of Algorithm 15, we can prune a rectangle that lies completely above the best envelope.

are pruned. In each iteration, every intermediate or leaf node that contributes a line to best k-upper envelope or best k-worst envelope is chosen to be accessed in the next round.



(a) R^* cannot lie completely above k-upper envelope

(b) R must be accessed to compute k-depth contour

Figure 5.6: Proving the optimality

It is easy to show that this algorithm only accesses the spectrums that overlap with k-upper or k-lower envelopes. Although the algorithm is I/O optimal for computing k-upper and k-lower envelopes, it cannot be shown optimal for k-depth contour computation. This is because k-depth contour computation does not require exact computation of k-upper (lower) envelope (only the convex vertices are to be computed). Nevertheless, we show that this k-depth contour algorithm is I/O optimal when k is smaller than the minimum number of objects in a leaf node of the R-tree. We prove this by showing that: i) every rectangle R must be accessed if its cardinality is greater than k and it does not completely lie within the k-depth contour (Lemma 12); and ii) for a rectangle R that does not lie completely within k-depth contour, its spectrum overlaps either k-upper envelope or k-lower envelope (Lemma 13).

LEMMA 12 : In order to compute the k-depth contour, every rectangle R that does not completely lie within the k-depth contour and has a cardinality greater than k must be accessed.

PROOF. We prove by contradiction. Assume that k-depth contour has been computed without accessing such a rectangle R. Fig. 5.6(b) shows this k-depth contour (the shaded polygon *abcde*) and such a rectangle R. Since R has not been accessed, we do not know anything about the orientation of objects inside it (except that it contains more than k objects and is a minimum bounding rectangle of these objects). We show that k-depth contour may be incorrect if R is not accessed (i.e., there exists at least one point p outside the contour that has depth at least equal to k).

Since k-depth contour is a convex polygon, at least one corner of R lies outside it (e.g., z in Fig. 5.6(b)). Without loss of generality, assume that k objects lie infinitely close to the this corner and the remaining objects lie elsewhere (e.g., on the opposite corner). Construct a triangle by joining z with any two arbitrary points inside the k-depth contour (e.g., see $\triangle qrz$). Since z lies outside k-depth contour, there exists at least one point p that lies inside $\triangle qrz$ and outside the k-depth contour. We show that p has depth at least equal to k which implies that it must be a part of k-depth contour (hence, a contradiction of the assumption).

Note that any line L passing through p has the corner z on one side and at least one of q or r on the other side. Without loss of generality, assume that the upper half-plane $\overrightarrow{L}$ contains the corner z and the lower half-plane $\underrightarrow{L}$ contains q. We show that for every such line L, the half-plane depth of $\overrightarrow{L}$ and $\underrightarrow{L}$ is at least equal to k (hence, depth of p is at least k). Since z contains k objects, it is easy to see that the half-plane depth of $\overrightarrow{L}$ is k. Now, we show that the half-plane depth of $\underrightarrow{L}$ cannot be less than k. We draw a line L' parallel to L and passing through

point q. Since q lies inside k-depth contour, the depths of both $\underrightarrow{L'}$ and $\overrightarrow{L'}$ are at least k. Since the depth of $\underrightarrow{L'}$ is at least k, the depth of $\underrightarrow{L}$ is also at least k. □

LEMMA 13 : For every rectangle R that does not lie completely within k-depth contour, R^* overlaps either k-upper envelope or k-lower envelope.

PROOF. Let $p \in R$ be a point outside or on k-depth contour. By definition, location depth of p is at most k. This implies that, in dual space, there exists at least one point z on p^* such that the upper (or lower) score of z is at most $k-1$. Hence, z either lies on or above k-upper envelope or lies on or below k-lower envelope. Hence, p^* overlaps with k-upper or k-lower envelope. □

Optimality for k-Snippet. As shown in Lemma 13, for every object o that does not lie within k-depth contour (i.e., o is a k-snippet object), o^* overlaps k-upper or k-lower envelope. Hence, k-snippet can be computed when k-upper and k-lower envelopes are computed. Next, we show that our algorithm is I/O optimal for computing k-snippet.

LEMMA 14 : To compute k-snippet, every I/O optimal algorithm must access every spectrum R^* that overlaps k-upper envelope.

PROOF. We prove this by showing that every object $o \in O$ is an object in k-snippet if o^* overlaps k-upper envelope. Since o^* overlaps k-upper envelope, there exists at least one point z on o^* that lies on or above k-upper envelope. By definition of k-upper envelope, the upper score of such point z is at most equal to $k-1$. Let z^* be the line in primal space mapped from the point z in dual space. Since the upper score of z is at most $k-1$, the number of objects lying above z^* is at most $k-1$. This implies that o is one of the top-k objects for the linear scoring function that corresponds to the line z^*. Hence, o is a k-snippet object. □

Following the similar arguments, we can prove that the algorithm must access every spectrum R^* that overlaps k-lower envelope. Since our algorithm accesses only the spectrums that overlap k-upper or k-lower envelope, it is I/O optimal.

5.4.5 Discussion

Handling special cases. For the ease of presentation, we assumed that the dual mapping does not contain two parallel lines and more than two concurrent lines. However, our techniques can be easily applied even when this assumption does not hold. More specifically, the former case can be handled by assuming that the parallel lines intersect each other at infinity [EW86]. To handle the latter case, the rider algorithm is to be modified such that when the rider reaches at an intersection of more than two lines it continues traveling on a line that has upper score equal to $k-1$.

Extension to higher dimensionality. Like most of the existing techniques, the focus of this work is two dimensional data sets. Nevertheless, the framework of KnightRider can be used to solve these problems in higher dimensions. In a d-dimensional space, a point in primal is mapped to a hyper-plane in dual (e.g., see [YAY12]). The pruning rules presented in this chapter can be extended to higher dimensionality because the basic properties of dual space mapping are preserved in higher dimensionality, e.g., a point p in primal lies above a hyper-plane L if and only if p^* lies above L^*. Hence, our framework can be used to prune the intermediate and leaf nodes of the R-tree. The best and worst envelope in multidimensional space can be computed using existing main-memory algorithms [AM95, AS98] because the number of unpruned entries is expected to be small (i.e., logarithmic to the data size [PTFS05]).

5.5 Experiments

5.5.1 Experimental settings

We mainly focus on evaluating our algorithm for computing k-depth contour. This is because our algorithms for other two problems have lower costs (since these problems do not require computing the convex vertices).

Synthetic data. We generate several data sets each following a different data distribution. More specifically, we generate data sets following Normal (norm for short), Correlated (corr), Anti-correlated (anti) [BKS01] and Uniform (unif) distributions in a unit square. We also generate data sets that follow Uniform distribution in a unit circle (circ for short). This type of distribution is expected to be more challenging because it increases the number of edges of k-depth contour.

Real data. We use roads data set which contains 2,249,727 streets of California (*http://www.rtreeportal.org*). We generate 5 million objects such that each street contains around 2 objects on average. Each object represents a house and has two attributes. The first attribute indicates its distance to the nearest beach and the second attribute corresponds to the distance to nearest airport. The locations of beaches and airports are taken from a collection of points of interest in California [LCH+05]. The users may prefer houses close to (far from) a beach and an airport. k-snippet then represents the set of houses such that each house is among top-k houses for at least one preference function. k-depth contour may find outliers, e.g., the houses that do not have any beach or airport nearby.

The table 5.1 shows different parameters used in our experiments and the bold values are the default values used in the experiments unless mentioned otherwise. The objects are indexed by an R-tree with page size set to 4KB (the minimum number of objects in any leaf node was 36)

Parameter	Range
Data distribution	real, norm, unif, **anti**, corr, circ
# of objects n (in millions)	1, 2, **5**, 10, 15, 20
k	1, 10, 20, 50, **100**, 150, 200

Table 5.1: Experiment Parameters for Depth-related Problems

5.5.2 Competitors and Benchmarks

CPU time. We compare our algorithms with the best known main-memory algorithms and assume that their algorithms have access to enough main memory to store the whole data set. Specifically, we compare our algorithm with FDC algorithm [JKN98] which is shown to be the most efficient main-memory algorithm. To compute k-depth contour, FDC needs to compute all m-depth contours ($1 \leq m \leq k$). For a strict evaluation, we only count the initialization (convex hull computation) time and the computation time for the last depth contour (when $m = k$). We rename this algorithm to FDC* because it is superior to the original FDC algorithm.

Computational geometry community also proposed several algorithms with complexities close to the optimal. Despite the fact that these algorithms provide nice complexity guarantees, unfortunately, these do not work well in practice. Nevertheless, we choose one such algorithm called BELT algorithm [EW86] that computes k-upper and k-lower envelopes with complexity guarantees close to optimal.

I/O cost. As shown in Section 5.4.4, KnightRider algorithm is I/O optimal for k-upper envelope and k-snippet problems. Unfortunately, KnightRider is not I/O optimal for k-depth contour problem when k is large. To evaluate the I/O cost, we assume the existence of an oracle and design an algorithm that achieves optimal I/O cost for k-depth contour problem.

It is easy to show that every I/O optimal algorithm must access every spectrum

R^* that contains a convex vertex (this can be shown by following similar arguments as in the proof of Lemma 9). Hence, to obtain the lower bound I/O cost, we assume that an oracle computes all the convex vertices (without incurring any I/O). Then, we traverse R-tree accessing only the entries such that their spectrums overlap one of these convex vertices. These I/Os are counted and correspond to the lower bound I/O cost. To evaluate the I/O costs of our algorithms, we use this lower bound cost as a benchmark.

5.5.3 Performance Analysis

Effect of data set size

In Fig. 5.7, we vary the data set size and evaluate I/O and CPU costs of our algorithm. Fig. 5.7(a) shows that I/O cost of our algorithm meets the lower bound I/O cost and is significantly lower than the I/O cost of SkyRider algorithm. Fig. 5.7(b) shows that our algorithms are more than an order of magnitude faster than FDC* and BELT algorithms.

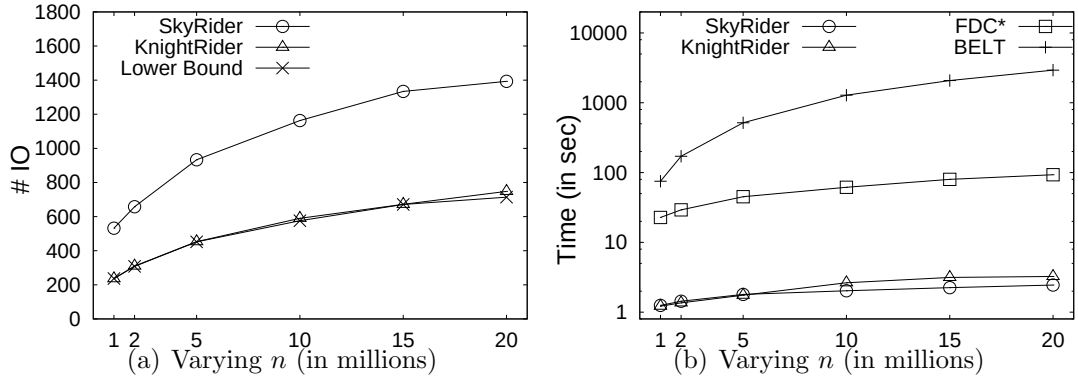


Figure 5.7: Effect of data sizes

Effect of k

In Fig. 5.8, we vary the value of k and evaluate the performance of our algorithms. Fig. 5.8(a) shows that the I/O cost of KnightRider meets the lower bound I/O cost and is significantly smaller than the I/O cost of SkyRider. Also, KnightRider scales better than SkyRider in terms of I/O cost.

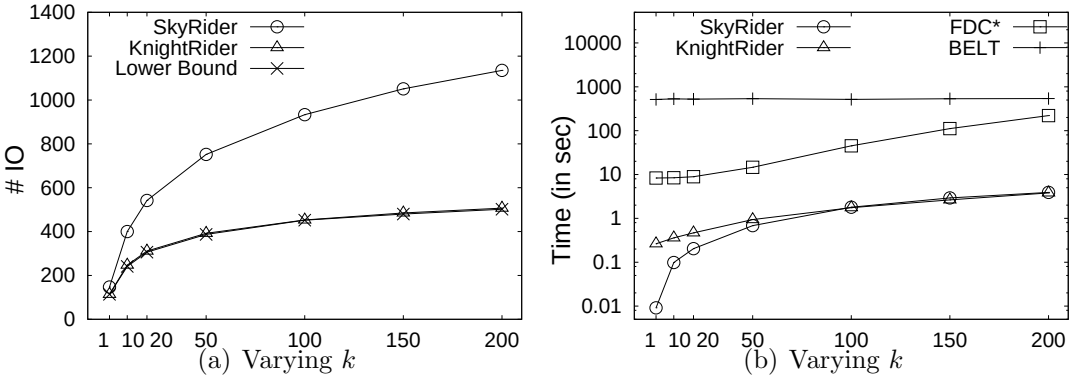


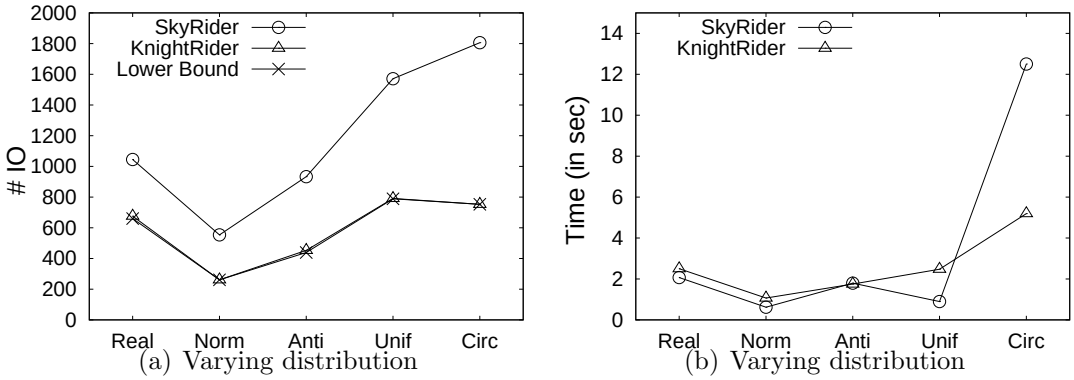
Figure 5.8: Effect of k

Fig. 5.8(b) shows that our algorithms are 1 to 3 orders of magnitude faster than FDC* and BELT algorithms. BELT is not significantly affected by the value of k because the time to initialize the dynamic convex hull [OvL81] used by BELT is the dominant cost. When k is 1, FDC* is identical to computing convex hull using Graham scan algorithm. Note that the cost of our algorithms even for $k = 200$ is lower than the cost of computing convex hull ($k = 1$) using Graham scan algorithm. We remark that this impressive result is partly due to the fact that our algorithm uses indexes whereas the existing algorithms do not use any index. We are not aware of any algorithm that computes k-depth contour utilizing any pre-built index.

Effect of data distribution

Fig. 5.9 studies the effect of data distribution on our algorithms. Since computational costs of FDC* and BELT are quite high, in Fig. 5.9(b), we only show the

costs of our algorithms for a better illustration of their comparison. In terms of CPU cost, KnightRider performs better than SkyRider on circ data set whereas its performance on other data sets is slightly worse.



(a) Varying distribution (b) Varying distribution

Figure 5.9: Effect of different data distributions

I/O cost for large k

As stated earlier, KnightRider is I/O optimal for k-depth contour computation only when k is small. In this section, we run experiments for large values of k and evaluate the I/O costs of our algorithms. Table 5.2 demonstrates that the I/O cost of KnightRider algorithm is almost the same as the lower bound I/O cost even when $k = 100,000$. Recall that we were unable to show I/O optimality because k-depth contour may be computed correctly even if only the convex vertices are computed instead of k-upper and k-lower envelopes. We observe that it is extremely rare that a rectangle affects the k-upper and k-lower envelopes but does not affect the convex vertices. Hence, the I/O cost of our algorithm is almost the same as lower bound I/O cost.

k	1	10	100	1000	10,000	100,000
SkyRider	147	400	933	2081	9864	53907
KnightRider	114	241	453	670	2444	12294
Lower Bound	114	241	452	667	2442	12293

Table 5.2: Number of I/O accesses for k-depth contour problem

Effectiveness of the rider algorithm

An alternative approach to compute k-upper envelope is to first compute k-skybands using BBS (as discussed in Section 5.3.2) and then use some existing main-memory algorithm (e.g., FDC*) instead of the rider algorithm. We call such algorithm SkyFDC* algorithm. In this section, we demonstrate the effectiveness of the rider algorithm by showing that SkyRider is significantly more efficient than SkyFDC* algorithm. Fig. 5.10 shows the results for varying n and varying k and demonstrates that SkyRider is more than an order of magnitude faster than SkyFDC* (note that Fig. 5.10(b) uses logscale). I/O costs are not shown because both SkyRider and SkyFDC* have the same I/O cost (both algorithms use BBS to compute k-skybands).

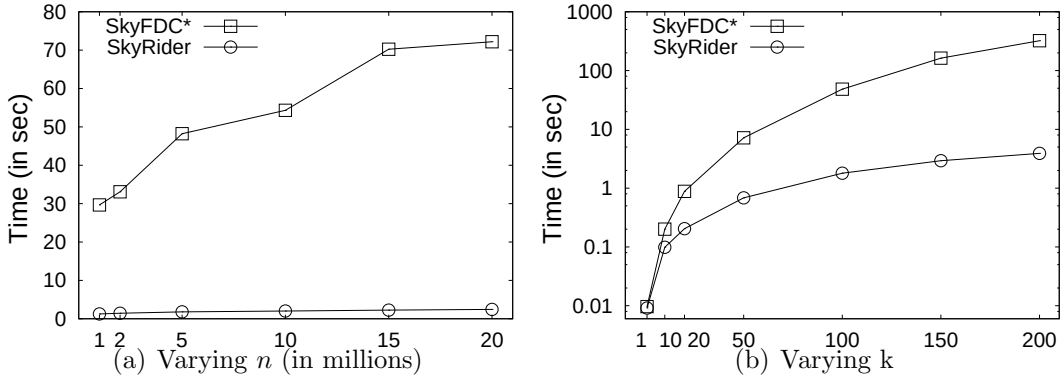


Figure 5.10: Effectiveness of the rider algorithm

5.5.4 k-skyband vs k-snippet

k-skyband [PTFS05] is a problem quite closely related to k-snippet. Specifically, assume that a user prefers larger values on both x and y coordinates. A k-skyband returns a set of objects such that each object o is one of the top-k objects for at least one *monotonic* scoring function. We define k-snippet++ as a set of objects such that each object o is one of the top-k objects for at least one monotonic linear scoring function. We call it k-snippet++ because this set is smaller than k-snippet and only considers scoring functions that prefer larger values on both x and y coordinates.

k-skyband is useful in the applications that use top-k queries involving any monotonic function (not necessarily linear functions). However, k-snippet++ may be preferable in the scenarios where top-k queries involve only linear functions. This is because k-snippet++ is expected to have smaller size as compared to k-skyband. In this section, we evaluate the effectiveness of k-snippet++ and our KnightRider algorithm. Both of our algorithms can be easily extended to return k-snippet++ by computing the k-upper envelope only in the positive dual space (see Section 5.3.2). We omit the proof of its optimality and correctness since it is straightforward.

Result size

k-skyband and k-snippet++ both are useful tools for returning a small subset of objects that may be among top-k objects for a broad class of top-k queries. Hence, the size of the returned subset is an important measure. Since k-snippet++ considers only linear functions, its size is expected to be smaller. This is demonstrated in Fig. 5.11 where we vary the value of k and data distribution and show the size of k-skyband and k-snippet++. The size of k-snippet++ is up to 5 times smaller

than the size of k-skyband. This shows that k-snippet++ may be a more useful tool to summarize the data when the queries involve only linear scoring functions.

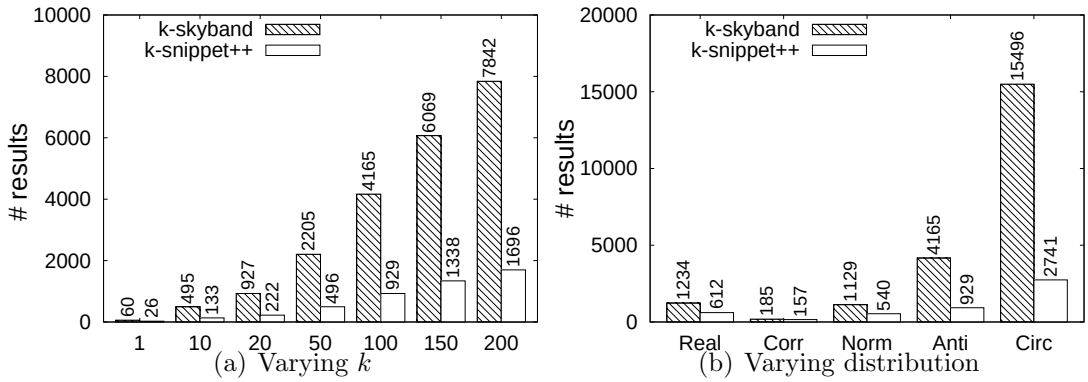


Figure 5.11: Size of query results

Performance evaluation

In this section, we compare the performance of our KnightRider algorithm with BBS [PTFS05] which is an I/O optimal algorithm for k-skyband. In Fig. 5.12, we compare the I/O costs of both algorithms. Note that both algorithms are I/O optimal for their respective problems. As expected, the I/O cost of KnightRider is significantly lower because it requires to return a smaller set of objects.

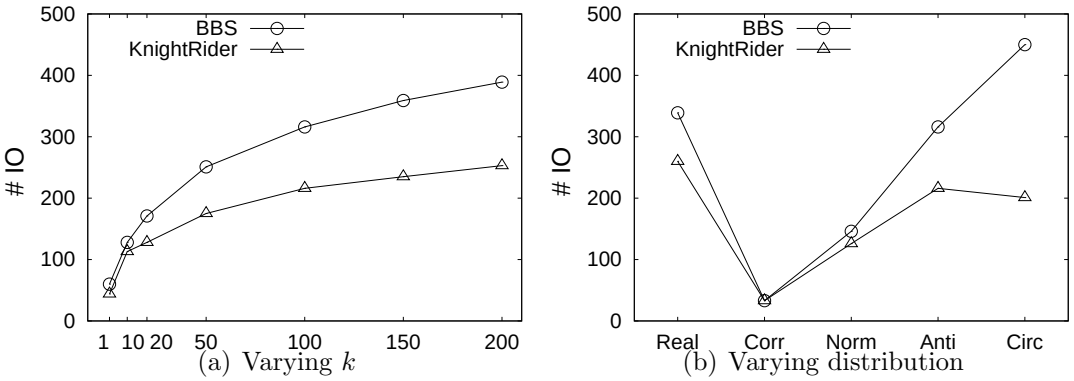
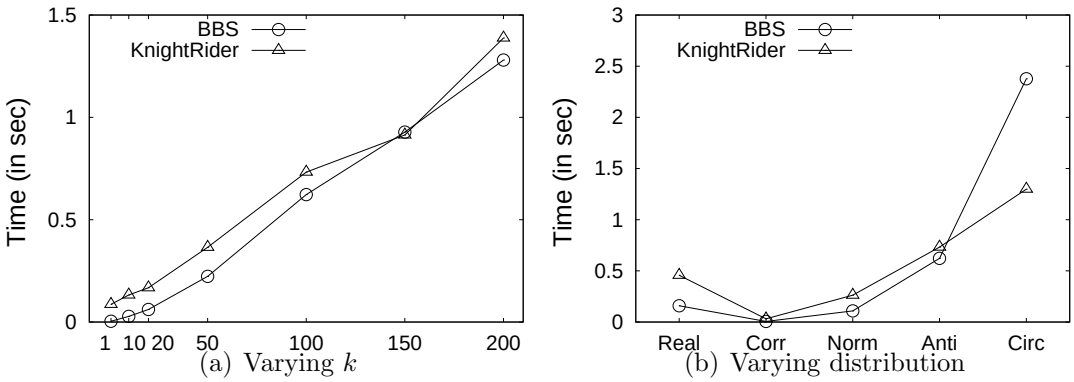


Figure 5.12: I/O cost comparison with BBS

In Fig. 5.13, we compare computational costs of both algorithms. While the difference is not significant, BBS performs slightly better than KnightRider except

for circ data set which is the most challenging data set for both of the algorithms. The performance of BBS is better because the requirement to prune objects for k-snippet is more challenging than k-skyband. Nevertheless, KnightRider has reasonably good performance as compared to BBS.



(a) Varying k (b) Varying distribution

Figure 5.13: CPU time comparison with BBS

The experimental results in this section demonstrate that k-snippet++ has a significantly smaller size and can be obtained almost as efficiently as k-skyband while incurring considerably less I/Os. Hence, k-snippet++ is more preferable than k-skyband for the applications that use top-k queries involving only linear scoring functions.

5.6 Conclusions

We are the first to propose efficient disk-based algorithms to solve depth-related problems over large data sets. One of our proposed algorithms is I/O optimal for k-upper envelope and k-snippet problems. We also show that it is I/O optimal for k-depth contour problem when k is smaller than the minimum number of objects in a leaf node of the R-tree. Our experimental results demonstrate the efficiency of our proposed algorithms.

Chapter 6

Final Remarks

6.1 Conclusions

In this thesis, we present efficient techniques to answer various top-k queries over spatial and temporal data under different settings. Chapter 3 provides our unified approach on top-k objects and pairs queries over sliding windows. Chapter 4 presents our research to answer top-k loyalty queries over data streams. Chapter 5 presents our I/O efficient algorithms for answering depth-related problems. Below are the details.

In Chapter 3, we study the problem of continuously monitoring top-k pairs and top-k objects. We present efficient techniques to answer a broad class of top-k pairs and top-k objects queries over sliding windows. The queries are answered using a small subset of pairs called K-skyband. We present efficient query answering techniques and skyband maintenance techniques. We provide a detailed complexity analysis and show that the storage requirement and the performance of our algorithms is reasonably close to the lower bound. The proposed framework can handle arbitrary scoring functions, supports queries with any window size and

works for out-of-order data streams. We verify this by an extensive experimental evaluation and demonstrate the efficiency of our approach.

In Chapter 4, we introduce the loyalty queries for a variety of applications. We present efficient algorithms and data structures to answer the top-k loyalty queries as well as threshold loyalty queries. The algorithm is based on the idea of line sweeping over the loyalty-time plane. We prove the lower bound cost of the problem and present a detailed complexity analysis to show that our algorithm is optimal. We verify this by an experimental evaluation and demonstrate the efficiency of our approach.

In Chapter 5, we study the depth-related problems which have a wide range of applications and have been extensively in the communities of statistics and computational geometry. We are the first to propose efficient disk-based algorithms to solve depth-related problems over large data sets. One of our proposed algorithms is I/O optimal for k-upper envelope and k-snippet problems. We also show that it is I/O optimal for k-depth contour problem when k is smaller than the minimum number of objects in a leaf node of the R-tree.

6.2 Future Work

Observing the increasing popularity of spatial and temporal data in modern applications such as location-based services, researchers have dedicates to accommodate such data sets into real-life applications. Top-k queries have been extensively studied in the last decade and novel techniques have been proposed to support top-k queries in different applications. However, there are still a number of interesting problem to be investigated in the future.

6.2.1 Top-k Spatial-keyword Search over Data Streams

In the real world applications, spatial data is always coexists with text information such as labels and comments. The problems of spatial-keyword search have been extensively studied in the recent years and novel techniques have been proposed to support top-k spatial-keyword queries over static databases in many applications. However, there are still a number of interesting problem to be investigated in the future. One direction is that we may consider the problem of online processing spatial-keyword queries over data streams. In application of Facebook and Twitters, people may attach their location information with the text information. Therefore, users may want to find the twitters or comments which contain the similar keywords and appear in a certain place. However, no existing work studies the problem of top-k spatial-keyword search in the context of data streams. Due to the high volume of data streams, online algorithms are demanded to efficiently answer spatial-keyword queries by the limited memory usage.

6.2.2 Top-k Diversity Queries over Data Streams

An interesting case is provided by spatial objects, which are produced in great quantity by location-based services that let users attach content to places, and arise also in many applications such as trip planning, new analysis, and real estate scenarios. The issued queries for retrieving the best set of objects relevant to given user criteria and well distributed over the region of interest. Therefore, diversifying top-k results of queries is crucial for such cases. Straightforward method for top-k diversity queries based on the existing methods are too costly, as they evaluate diversity by accessing and scanning all relevant objects, even if only a small subset is needed. Thus, efficient algorithm is necessary to processing the top-k diversity queries. The top-k diversity queries have been studied over the static databases by

Fraternali et al. [FMT12]. Motivated by the wide range of applications over data streams, top-k diversity queries over data streams could be a future direction of the research. Furthermore, it is more challenging to diversifying the results over data streams, as the available results vary dynamically.

6.2.3 Disk-based Approach for Other Depth Measures

In Chapter 5, we assume that the data is indexed by a branch and bound data structure. An interesting direction for further work is to design disk-based algorithms for the case when the data is not indexed. While the algorithms we propose in this chapter can be extended to compute *Tukey median* [Tuk74] and *Bagplot* [RRT99], the algorithms specially designed for these problems may be more effective. Tukey median is the center of gravity of the deepest k-depth contour and Bagplot is a two-dimensional generalization of of Tukey's univariate boxplot [Tuk77]. It will also be interesting to design disk-based algorithms for other depth measures such as convex peeling depth [Bar76], simplicial depth [Liu90] and regression depth [RH99] etc.

Bibliography

[AdBMS98] Pankaj K. Agarwal, Mark de Berg, Jirí Matousek, and Otfried Schwarzkopf. Constructing levels in arrangements and higher order voronoi diagrams. *SIAM J. Comput.*, 27(3):654–667, 1998.

[AM95] P.K. Agarwal and J. Matoušek. Dynamic half-space range reporting and its applications. *Algorithmica*, 13(4):325–345, 1995.

[AP05] Fabrizio Angiulli and Clara Pizzuti. An approximate algorithm for top-k closest pairs join query in large high dimensional data. *Data Knowl. Eng.*, 53(3):263–281, 2005.

[AS98] Pankaj K. Agarwal and Micha Sharir. Arrangements and their applications. In *Handbook of Computational Geometry*, pages 49–119, 1998.

[Bar76] V. Barnett. The ordering of multivariate data (with discussion). *Journal of the Royal Statistical Society. Series A*, 139:318–354, 1976.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In Lucian Popa, editor, *PODS*, pages 1–16. ACM, 2002.

[BDMM04] Brian Babcock, Mayur Datar, Rajeev Motwani, and Rajeev Motwani. Load shedding for aggregation queries over data streams. In *ICDE*, pages 350–361, 2004.

[BFP+73] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *JCSS*, 1973.

[BGH99] Julien Basch, Leonidas J. Guibas, and John Hershberger. Data structures for mobile data. *J. Algorithms*, 31(1):1–28, 1999.

[BK01] Christian Böhm and Hans-Peter Kriegel. Determining the convex hull in large multidimensional databases. In *DaWaK*, pages 294–306, 2001.

[BKS01] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[BO79] Jon Louis Bentley and Thomas Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Computers*, 28(9):643–647, 1979.

[BOPY07] Christian Böhm, Beng Chin Ooi, Claudia Plant, and Ying Yan. Efficiently processing continuous k-NN queries on data streams. In *ICDE*, 2007.

[Bor94] Alberto Borobia. Mirror property for nonsingular mixed configurations of lines and points in r^3. *Discrete & Computational Geometry*, pages 311–320, 1994.

[CBL+10] Muhammad Aamir Cheema, Ljiljana Brankovic, Xuemin Lin, Wenjie Zhang, and Wei Wang. Multi-guarded safe zone: An effective technique to monitor moving circular range queries. In *ICDE*, pages 189–200, 2010.

[CE87] Bernard Chazelle and Herbert Edelsbrunner. An improved algorithm for constructing k th-order voronoi diagrams. *IEEE Trans. Computers*, pages 1349–1354, 1987.

[Cha00] Timothy M. Chan. Random sampling, halfspace range reporting, and construction of (<= k)-levels in three dimensions. *SIAM J. Comput.*, pages 561–575, 2000.

[CKT08] Graham Cormode, Flip Korn, and Srikanta Tirthapura. Time-decaying aggregates in out-of-order streams. In *PODS*, pages 89–98, 2008.

[Cla87] Kenneth L. Clarkson. New applications of random sampling in computational geometry. *Discrete & Computational Geometry*, 2:195–222, 1987.

[CLW+11] Muhammad Aamir Cheema, Xuemin Lin, Haixun Wang, Jianmin Wang, and Wenjie Zhang. A unified approach for computing top-k pairs in multidimensional space. In *ICDE*, pages 1031–1042, 2011.

[CLZZ11] Muhammad Aamir Cheema, Xuemin Lin, Wenjie Zhang, and Ying Zhang. Influence zone: Efficiently processing reverse k nearest neighbors queries. In *ICDE*, pages 577–588, 2011.

[CMTV00] Antonio Corral, Yannis Manolopoulos, Yannis Theodoridis, and Michael Vassilakopoulos. Closest pair queries in spatial databases. In *SIGMOD*, pages 189–200, 2000.

[CMY+12] Graham Cormode, S. Muthukrishnan, Ke Yi, Qin Zhang, and Qin Zhang. Continuous sampling from distributed streams. page 10, 2012.

[CP86] Bernard Chazelle and Franco P. Preparata. Halfspace range search: An algorithmic application of k-sets. *Discrete & Computational Geometry*, pages 83–93, 1986.

[CSY84] Richard Cole, Micha Sharir, and Chee-Keng Yap. On k-hulls and related problems. In *STOC*, pages 154–166, 1984.

[DGKS07] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Nikos Sarkas. Ad-hoc top-k query answering for data streams. In *VLDB*, pages 183–194, 2007.

[DKR06] Antonios Deligiannakis, Yannis Kotidis, and Nick Roussopoulos. Processing approximate aggregate queries in wireless sensor networks. *Inf. Syst.*, 31(8):770–792, 2006.

[ERvK96] Hazel Everett, Jean-Marc Robert, and Marc J. van Kreveld. An optimal algorithm for the (<= k)-levels, with applications to separation and transversal problems. *Int. J. Comput. Geometry Appl.*, 6(3):247–261, 1996.

[EW86] Herbert Edelsbrunner and Emo Welzl. Constructing belts in two-dimensional arrangements with applications. *SIAM J. Comput.*, 15:271–284, 1986.

[FKS03] Ronald Fagin, Ravi Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *SIGMOD*, 2003.

[FLN03] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[FMT12] Piero Fraternali, Davide Martinenghi, and Marco Tagliasacchi. Top-k bounded diversification. In *SIGMOD Conference*, pages 421–432, 2012.

[FRC11] Tobias Farrell, Kurt Rothermel, and Reynold Cheng. Processing continuous range queries with spatiotemporal tolerance. *IEEE Trans. Mob. Comput.*, 10(3):320–334, 2011.

[GBK00] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Optimizing multi-feature queries for image databases. In *VLDB*, 2000.

[GBÖ06] Lukasz Golab, Kumar Gaurav Bijay, and M. Tamer Özsu. Multi-query optimization of sliding window aggregates by schedule synchronization. In *CIKM*, pages 844–845, 2006.

[GL04] Bugra Gedik and Ling Liu. Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *EDBT*, pages 67–87, 2004.

[GWYL06] Bugra Gedik, Kun-Lung Wu, Philip S. Yu, and Ling Liu. Processing moving queries over moving objects using motion-adaptive indexes. *IEEE Trans. Knowl. Data Eng.*, 18(5):651–668, 2006.

[Her89] John Hershberger. Finding the upper envelope of n line segments in o(n log n) time. *Inf. Process. Lett.*, 33(4):169–174, 1989.

[HRSS06] J. Hugg, E. Rafalin, K. Seyboth, and D. Souvaine. An experimental study of old and new depth measures. In *ALENEX*, pages 51–64, 2006.

[HS98] Gísli R. Hjaltason and Hanan Samet. Incremental distance join algorithms for spatial databases. In *SIGMOD*, 1998.

[IBS08] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top- query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[JKN98] Theodore Johnson, Ivy Kwok, and Raymond T. Ng. Fast computation of 2-dimensional depth contours. In *KDD*, pages 224–228, 1998.

[KCRR06] Ram Keralapura, Graham Cormode, Jeyashankher Ramamirtham, and Jeyashankher Ramamirtham. Communication-efficient distributed monitoring of thresholded counts. In *SIGMOD Conference*, pages 289–300, 2006.

[KMV02] Shankar Krishnan, Nabil H. Mustafa, and Suresh Venkatasubramanian. Hardware-assisted computation of depth contours. In *SODA*, pages 558–567, 2002.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition.* Addison-Wesley, 1973.

[KPKK09] Maleq Khan, Gopal Pandurangan, V. S. Anil Kumar, and V. S. Anil Kumar. Distributed algorithms for constructing approximate minimum spanning trees in wireless sensor networks. pages 124–139, 2009.

[KZ10] Linglong Kong and Yijun Zuo. Smooth depth contours characterize the underlying distribution. *J. Multivariate Analysis*, 101(9):2222–2226, 2010.

[LCH+05] Feifei Li, Dihan Cheng, Marios Hadjieleftheriou, George Kollios, and Shang-Hua Teng. On trip planning queries in spatial databases. In *SSTD*, pages 273–290, 2005.

[Liu90] R.Y. Liu. On a notion of data depth based on random simplices. *The Annals of Statistics*, 18(1):405–414, 1990.

[LLG+09] Mo Liu, Ming Li, Denis Golovnya, Elke A. Rundensteiner, and Kajal T. Claypool. Sequence pattern query processing over out-of-order event streams. In *ICDE*, pages 784–795, 2009.

[LMT+05] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD Conference*, pages 311–322, 2005.

[LPS99] Regina Y. Liu, Jesse M. Parelius, and Kesar Singh. Multivariate analysis by data depth: Descriptive statistics, graphics and inference. *The Annals of Statistics*, 27(3):783–840, June 1999.

[LYH04] Yifan Li, Jiong Yang, and Jiawei Han. Continuous k-nearest neighbor search for moving objects. In *SSDBM*, pages 123–126, 2004.

[LYL10] Feifei Li, Ke Yi, and Wangchao Le. Top-k queries on temporal data. *VLDB J.*, 19(5):715–733, 2010.

[LYWL05] Xuemin Lin, Yidong Yuan, Wei Wang, and Hongjun Lu. Stabbing the sky: Efficient skyline computation over sliding windows. In *ICDE*, pages 502–513. IEEE Computer Society, 2005.

[Mat02] J. Matoušek. *Lectures on discrete geometry*. Springer Verlag, 2002.

[MBP06] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD Conference*, 2006.

[McC85] Edward M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, May 1985.

[MP07a] Kyriakos Mouratidis and Dimitris Papadias. Continuous nearest neighbor queries over sliding windows. *IEEE Trans. Knowl. Data Eng*, 19(6):789–803, 2007.

[MP07b] Kyriakos Mouratidis and Dimitris Papadias. Continuous nearest neighbor queries over sliding windows. *IEEE TKDE*, 2007.

[MRR+01] Kim Miller, Suneeta Ramaswami, Peter Rousseeuw, Joan Antoni Sellarès, Diane L. Souvaine, Ileana Streinu, and Anja Struyf. Fast implementation of depth contours using topological sweep. In *SODA*, pages 690–699, 2001.

[MXA04] Mohamed F. Mokbel, Xiaopeng Xiong, and Walid G. Aref. Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD Conference*, pages 623–634, 2004.

[MYCC07] Nikos Mamoulis, Man Lung Yiu, Kit Hung Cheng, and David W. Cheung. Efficient top- aggregation of ranked inputs. *ACM Trans. Database Syst.*, 32(3):19, 2007.

[NNRS08] Kanthi Nagaraj, K. V. M. Naidu, Rajeev Rastogi, and Scott Satkin. Efficient aggregate computation over data streams. In *ICDE*, pages 1382–1384, 2008.

[NR99] Surya Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *ICDE*, 1999.

[OvL81] Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Syst. Sci.*, 23(2):166–204, 1981.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: an Introduction*. Springer, Berlin, 1985.

[PTFS05] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, pages 41–82, 2005.

[RH99] P.J. Rousseeuw and M. Hubert. Depth in an arrangement of hyperplanes. *Discrete & Computational Geometry*, 22(2):167–176, 1999.

[RKGG07] Daniel Russel, Menelaos I. Karavelas, Leonidas J. Guibas, and Leonidas J. Guibas. A package for exact kinetic data structures and sweepline algorithms. pages 111–127, 2007.

[RR96] I. Ruts and P.J. Rousseeuw. Computing depth contours of bivariate point clouds. *Computational Statistics & Data Analysis*, 23:153–168, 1996.

[RR98] Peter J. Rousseeuw and Ida Ruts. Constructing the bivariate tukey median. *Statistica Sinica*, pages 827–839, 1998.

[RRT99] Peter J. Rousseeuw, Ida Ruts, and John W. Tukey. The bagplot: A bivariate boxplot. *The American Statistician*, 53(4):382–387, November 1999.

[SCL12a] Zhitao Shen, Muhammad Aamir Cheema, and Xuemin Lin. Loyalty-based selection: Retrieving objects that persistently satisfy criteria. In *CIKM*, 2012.

[SCL+12b] Zhitao Shen, Muhammad Aamir Cheema, Xuemin Lin, Wenjie Zhang, and Haixun Wang. Efficiently monitoring top-k pairs over sliding windows. In *ICDE*, 2012.

[SCL+12c] Zhitao Shen, Muhammad Aamir Cheema, Xuemin Lin, Wenjie Zhang, and Haixun Wang. Efficiently monitoring top-k pairs over sliding windows. In *ICDE*, 2012.

[Smi97] Michiel Smid. Closest-point problems in computational geometry. In *Handbook on Computational Geometry, published by Elsevier Science*, 1997.

[SZS03] Jing Shan, Donghui Zhang, and Betty Salzberg. On spatial-range closest-pair query. In *SSTD*, pages 252–269, 2003.

[TPK+03] Panayiotis Tsaparas, Themistoklis Palpanas, Yannis Kotidis, Nick Koudas, and Divesh Srivastava. Ranked join indices. In *ICDE*, 2003.

[TPP05] Yufei Tao, Dimitris Papadias, and Dimitris Papadias. Historical spatio-temporal aggregation. pages 61–102, 2005.

[Tuk74] John W. Tukey. Mathematics and picturing of data. In *International Congress of Mathematicians*, 1974.

[Tuk77] J.W. Tukey. Exploratory data analysis. *Reading, MA*, 1977.

[TZZ06] Nesime Tatbul, Stanley B. Zdonik, and Stanley B. Zdonik. Window-aware load shedding for aggregation queries over data streams. In *VLDB*, pages 799–810, 2006.

[UMY07] Leong Hou U, Nikos Mamoulis, and Man Lung Yiu. Continuous monitoring of exclusive closest pairs. In *SSTD*, 2007.

[Vid04] Ganapathy Vidyamurthy. *Pairs Trading: quantitative methods and analysis*. John Wiley & Sons, Inc., 2004.

[WRG+06] Song Wang, Elke A. Rundensteiner, Samrat Ganguly, Sudeept Bhatnagar, and Sudeept Bhatnagar. State-slice: New paradigm of multi-query optimization of window-based stream queries. In *VLDB*, pages 619–630, 2006.

[XCH06] Dong Xin, Chen Chen, and Jiawei Han. Towards robust indexing for ranked queries. In *VLDB*, pages 235–246, 2006.

[XMA05] Xiaopeng Xiong, Mohamed F. Mokbel, and Walid G. Aref. Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *ICDE*, pages 643–654, 2005.

[YAY12] Albert Yu, Pankaj K. Agarwal, and Jun Yang. Processing a large number of continuous preference top-k queries. In *SIGMOD Conference*, pages 397–408, 2012.

[YL02] Congjun Yang and King-Ip Lin. An index structure for improving nearest closest pairs and related join queries in spatial databases. In *IDEAS*, 2002.

[YLz+07] Xiaoyan Yang, Hock-Beng Lim, M. Tamer zsu, Kian-Lee Tan, and Kian-Lee Tan. In-network execution of monitoring queries in sensor networks. In *SIGMOD Conference*, pages 521–532, 2007.

[ZKOS05] Rui Zhang, Nick Koudas, Beng Chin Ooi, and Divesh Srivastava. Multiple aggregations over data streams. In *SIGMOD Conference*, pages 299–310, 2005.

[ZLZ+09] Wenjie Zhang, Xuemin Lin, Ying Zhang, Wei Wang, and Jeffrey Xu Yu. Probabilistic skyline operator over sliding windows. In *ICDE*, pages 1060–1071, 2009.

[ZZS+05] Panfeng Zhou, Donghui Zhang, Betty Salzberg, Gene Cooperman, and George Kollios. Close pair queries in moving object databases. In *GIS*, 2005.