ALGORITHMS FOR EXPLORATORY QUERIES OVER WEB DATABASE

by

MD FARHADUR RAHMAN

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 2018

Copyright © by Md Farhadur Rahman 2018

All Rights Reserved

To My Parents

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my professor Dr. Gautam Das for his continuous guidance, motivation, and support. I am very lucky to have him as my Ph.D supervisor. The regular meetings I had with him were very helpful. I have learned a lot from him. He has also introduced me to a group of excellent collaborators, especially Dr. Nan Zhang of Pennsylvania State University and Dr. Nick Koudas of University of Toronto. I would also like to also thank my thesis committee members: Dr. Chengkai Li, Dr. Ramez Elmasri, and Dr. Vassilis Athitsos for their suggestions and comments.

I would like to thank my former labmate Saravanan, who acted as my mentor during the early years of my Ph.D. His guidance helped me a lot in my Ph.D career. I would also like to thank my other labmates Habib, Azade, and Abol and others for their support.

Last but not least I would like to thank my parents, my sisters Ritu and Fariha, and my wonderful wife Tuba for their unconditional love and care.

August 13, 2018

ABSTRACT

ALGORITHMS FOR EXPLORATORY QUERIES OVER WEB DATABASE

Md Farhadur Rahman, Ph.D.

The University of Texas at Arlington, 2018

Supervising Professor: Gautam Das

In recent years we have seen an increase in the popularity of many web applications. The functionality of these applications range from allowing users to interact using online social network, to assist users in their everyday activity such as selecting a hotel in an area, locating a nearby restaurant etc. Google Maps, WeChat, FourSquare, AirBnB, TripAdvisor, and Hotels.com are a few such examples. The backed database of these applications can be a rich source of information for the corresponding application domain. For example, using Google Maps a user can find the ratings, reviews, and price of a restaurant, using Zillow users compare the price distributions of houses in different areas of a city.

The public query interfaces of these applications may be abstractly modeled as a kNN interface over the backend database that returns k results that are nearest to the user query, where k is typically a small number such as 10 or 50. Moreover, Access to the un-

derlying database is further restricted by enforcing *query rate limitation*, i.e., they limit the number of requests from an IP address or API account for a certain time period. Because of these restrictions, it is quite impossible for a third-party to crawl the data from the backend database.

In this dissertation, we present efficient techniques for exploratory analysis over web database. Specifically, we propose several algorithms that enable third-party applications to perform the analytics and mining of the backend database by using nothing but the restrictive, public-access, interface of the application. In addition, we also studied the problem of answering the exploratory queries with full access to the underlying dataset.

First, we consider a special category of web application know as *Location based services* (LBS). In general, an LBS provides a public search interface over its backend database, taking as input a 2D query point and returning k tuples in the database that are closest to the query point. In this work, we propose several algorithms that enable *approximate estimate* of SUM and COUNT aggregates by using the public search interface provided by the LBS.

Second, we enable density-based clustering over the backend database of an LBS using nothing but limited access to the kNN interface provided by the LBS. Our goal here is to mine from the LBS a *cluster assignment function* $f(\cdot)$, such that for any tuple t in the database (which may or may not have been accessed), $f(\cdot)$ can produce the cluster assignment of t with high accuracy.

Third, we investigate a novel problem on the *privacy implications* of database ranking, which has not been studied before. We show how privacy leakage (through the top-k interface) can be caused by a seemingly innocent design of the ranking function in ranked retrieval models. We introduce a comprehensive taxonomy of the problem space. Then, for each subspace of the problem, we develop a novel technique which either guarantees

the successful inference of private attributes, or accomplishes the attack for a significant portion of real-world tuples.

Finally, we study the problem of *subspace skyline discovery* over web database where the attributes are mostly categorical. Skyline queries are an effective tool in assisting users in data exploration. In accordance with common practice in traditional database query processing, we design solutions for two important practical instances of this problem, namely: (i) assuming that no indices exist on the underlying dataset, and (ii) assuming that indices exist on each individual attribute of the dataset.

TABLE OF CONTENTS

Chapter Page

1. Introduction 1
   - 1.1 Aggregate Estimations over Location Based Services 2
   - 1.2 Efficient Analytics over Location Based Services 3
   - 1.3 Density based Clustering over Location Based Services 4
   - 1.4 Privacy Implications of Database Ranking 5
   - 1.5 Subspace Skyline over Categorical Domains 6
2. Aggregate Estimations over Location Based Services 8
   - 2.1 Introduction 9
     - 2.1.1 LBS with a kNN Interface 9
     - 2.1.2 Aggregate Estimations 10
     - 2.1.3 Outline of Technical Results 11
     - 2.1.4 Summary of Contributions 13
   - 2.2 Background 14
     - 2.2.1 Model of LBS 14
     - 2.2.2 Voronoi Cells 16
     - 2.2.3 Problem Definition 18

2.3 LR-LBS-AGG 19
2.3.1 Key Idea: Precisely Compute Voronoi Cells 20
2.3.2 Error Reduction 23
2.3.3 Algorithm LR-LBS-AGG 32
2.4 LNR-LBS-AGG 32
2.4.1 Voronoi Cell Computation: Key Idea 32
2.4.2 Extension to $k > 1$ 36
2.4.3 Tuple Position Computation 38
2.5 Discussions 40
2.5.1 Aggregates with Selection Conditions 40
2.5.2 Leveraging External Knowledge 41
2.5.3 Special LBS Constraints 43
2.5.4 Extension to Higher Dimensions 44
2.6 Experimental Results 45
2.6.1 Experimental Setup 45
2.6.2 Experiments over Real-World Datasets 47
2.6.3 Online Demonstrations 49
2.7 Related Work 52
2.8 Final Remarks 53
2.9 Binary Search Process 54
3. ANALOC: Efficient ANAlytics over LOCation Based Services 58
3.1 Introduction 59
3.2 System Architecture 62
3.2.1 Web Server 62
3.2.2 Sampling Server 63
3.2.3 Web-DB Interface Server 65

3.2.4 User Credentials Database 66
3.3 Demo Plan 66
3.3.1 Overview 66
3.3.2 Demo Scenario 1: Aggregate Estimation 67
3.3.3 Demo Scenario 2: Hotspots and Explanations 68
3.4 Summary 69
4. Density based Clustering over Location Based Services 70
4.1 Introduction 71
4.2 Background 77
4.2.1 Model of LBS 77
4.2.2 Problem Definition 78
4.2.3 Density Based Clustering 81
4.3 1D Case 82
4.3.1 1D Baseline and Problem 83
4.3.2 Algorithm HDBSCAN-1D 83
4.4 HDBSCAN 87
4.4.1 Overview 88
4.4.2 Mapping 2D to 1D 90
4.4.3 Merging Mini-Clusters 92
4.4.4 Clustering of New Points 93
4.5 Discussion 94
4.5.1 Leveraging External Knowledge 94
4.5.2 Special LBS Constraints 94
4.6 Experimental Results 96
4.6.1 Experimental Setup 96
4.6.2 Experiments over Benchmark Datasets 99

4.6.3 Experiments over Real-World LBS 102
4.7 Related Work 104
4.8 Final Remarks 105
5. Privacy Implications of Database Ranking 106
5.1 Introduction 107
5.1.1 Motivation 107
5.1.2 Novel Problem: Rank-Based Inference 110
5.1.3 Overview of Technical Results 111
5.2 Preliminaries 114
5.2.1 Model of Web Databases 114
5.2.2 Properties of Ranking Function 116
5.3 Problem Space 118
5.3.1 Adversary Model 118
5.3.2 Two Dimensions 119
5.3.3 Problem Definition 120
5.4 Point Query Interface 121
5.4.1 Goal: Finding Differential Queries 121
5.4.2 Q&I adversary 122
5.4.3 Q-only adversary 131
5.5 IN Query Interface 136
5.5.1 Q&I Adversary 136
5.5.2 Q-only adversary 141
5.6 Discussions 144
5.6.1 Numeric Attributes 144
5.6.2 Defense Against Rank-based Inference 145
5.7 Experimental Results 146

5.7.1 Experimental Setup 146
5.7.2 Experiments over Real-World Datasets 148
5.7.3 Online Demonstration 151
5.8 Additional Details for Online Experiments 154
5.9 Related Work 158
5.10 Final Remarks 158
6. Eicient Computation of Subspace Skyline over Categorical Domains 160
6.1 Introduction 161
6.1.1 Motivation 161
6.1.2 Technical Highlights 163
6.1.3 Summary of Contributions 165
6.1.4 Paper Organization 165
6.2 Preliminaries 166
6.2.1 Skyline 166
6.2.2 Sorted Lists 167
6.2.3 Problem Definition 167
6.3 Skyline Computation Over Categorical Attributes 168
6.3.1 Organizing Tuples Tree 169
6.3.2 Skyline using Tree 174
6.3.3 Extension for Categorical Attributes 179
6.4 Subspace Skyline using Sorted
Lists 180
6.4.1 TOP-DOWN 182
6.4.2 TA-SKY 183
6.5 Related Work 189
6.6 Experimental Evaluation 191

6.6.1 Experimental Setup 191
6.6.2 Experiments over Synthetic Datasets 194
6.6.3 Experiments over AirBnB Dataset 197
6.6.4 Experiments over Zillow Dataset 199
6.7 Final Remarks 199
6.8 Appendix 200
6.8.1 Tree Data Structure Optimization 200
6.8.2 TOP-DOWN 201
6.8.3 Proofs 207

LIST OF ILLUSTRATIONS

Figure | Page

2.1 Concavity of top-k Voronoi Diagrams 17
2.2 Illustration of Theorem 1 18
2.3 Illustration of LR-LBS-AGG 18
2.4 Faster Initialization - Success 18
2.5 Faster Initialization - Failure 24
2.6 Leveraging History 24
2.7 Upper/Lower Bounds 24
2.8 Illustration of Algorithm LNR-LBS-AGG 35
2.9 Handling Concavity of top-k Voronoi Diagrams 36
2.10 Demonstration of Tuple Position Computation 39
2.11 Voronoi Decomposition of Starbucks in US 48
2.12 Unbiasedness of Estimators 50
2.13 Impact of Sampling Strategy 50
2.14 COUNT(schools) 50
2.15 COUNT(restaurants) 50
2.16 SUM(enrollment) in Schools 50

2.17 AVG(ratings) in Austin, TX Restaurants 50
2.18 Varying Database Size 50
2.19 Varying k 50
2.20 Query Savings of Error Reduction Strategies 52
2.21 Localization Accuracy 52
3.1 Architecture of ANALOC 62
3.2 Input Interface 64
3.3 Output Interface 64
3.4 Voronoi Tessellation of Samples 67
3.5 Weibo Hotspots 69
4.1 Mapping 2D space to 1D using Adaptive SFC 87
4.2 After clustering in 1D space 87
4.3 Merging nearby clusters in 2D space 87
4.4 Assigning new points to closest cluster 87
4.5 Illustration of popular SFC: (a) Hilbert curve (b) Z-curve (c) Peano curve . . 89
4.6 Mapping 2D points to 1D through Hilbert SFC 89
4.7 Impact of Minimum Grid Size on Clusters 89
4.8 Standard vs Adaptive SFC 92
4.9 Adaptive SFC 92
4.10 Chameleon t4.8k Dataset 96
4.11 Output of HDBSCAN 96
4.12 Chameleon t8.8k Dataset 96
4.13 Output of HDBSCAN 96
4.14 Banana Dataset 96
4.15 Output of HDBSCAN 96
4.16 Birch R2 Dataset 96

4.17 Output of HDBSCAN 96
4.18 Comparison with baseline Algorithm 96
4.19 Cluster Quality for different SFCs 96
4.20 Cluster Quality Vs Query Cost 97
4.21 Varying K 97
4.22 Varying min. grid size 97
4.23 Varying min. grid size 97
4.24 Varying number of Partitions per node 97
4.25 Varying percentage of Noise in dataset 97
4.26 Clustering of Places in DC using Yahoo Flickr geotagged photo 102
4.27 Clustering of houses in DFW area using Redfin 102
4.28 Clustering of houses in DFW area on Zillow dataset 102
4.29 Price distribution of house for sales in DFW area 102
4.30 Clustering of Capital Bikeshare stations in DC 103
4.31 Average Rent per hour distribution for Cluster 1 103
5.1 Demonstration of an attack over Renren 109
5.2 Enumeration Tree in Algorithm Q-Point 133
5.3 Attack Success Rate 146
5.4 Varying k 146
5.5 Varying n 146
5.6 Varying m 146
5.7 Varying m' 147
5.8 Varying w'_1 147
5.9 Fraction of Uncompromised Accounts 147
5.10 Varying Domain Size of Inferred Attribute 147
5.11 Query Cost to Infer all Private Attributes 150

5.12 Renren: Success Rate vs. k . . . 150
5.13 Varying k (YA) . . . 151
5.14 Varying n (YA) . . . 151
5.15 Varying m (YA) . . . 151
5.16 Varying m' (YA) . . . 151
5.17 Varying w'_1 (YA) . . . 151
5.18 Fraction of Uncompromised Accounts (YA) . . . 151
5.19 Varying Domain Size of Inferred Attribute (YA) . . . 151
5.20 Query Cost to Infer all Private Attributes (YA) . . . 151
5.21 Varying k (BOOL) . . . 152
5.22 Varying n (BOOL) . . . 152
5.23 Varying m (BOOL) . . . 152
5.24 Varying m' (BOOL) . . . 152
5.25 Varying w'_1 (BOOL) . . . 152
5.26 Fraction of Uncompromised Accounts (BOOL) . . . 152
5.27 Varying p (BOOL) . . . 152
5.28 Query Cost to Infer all Private Attributes (BOOL) . . . 152
5.29 Query q_1 where Anya is top ranked . . . 155
5.30 Query q_2 where Anya is not top ranked . . . 155
5.31 Demonstration of an attack over Goodreads . . . 157
6.1 Tree structure for relation in Example 1 . . . 170
6.2 Prune dominated tuples . . . 170
6.3 Tree after removing dominated tuples . . . 171
6.4 Check if tuple t is dominated . . . 171
6.5 Expected cost of IS-DOMINATED and PRUNE-DOMINATED-TUPLES operations as a function of s . . . 179

6.6 Example: Sorted Lists, Organization 1 181
6.7 Example: Sorted Lists, Organization 2 181
6.8 Varying query size 191
6.9 Varying query size 191
6.10 Varying number of tuples 191
6.11 Varying cardinality 191
6.12 Time vs number of skylines returned 191
6.13 Tuples accessed vs number of skylines returned 191
6.14 AirBnB: Varying query size 191
6.15 AirBnB: TA-SKY performance v.s. Skyline size 191
6.16 AirBnB: Varying the number of tuples 196
6.17 AirBnB: Time vs the number of skylines returned 196
6.18 AirBnB: Number of accessed tuples vs the number of skylines 196
6.19 Zillow: Varying query size 196
6.20 Zillow: Varying the number of tuples 196
6.21 Zillow: Time vs the number of skylines returned 196
6.22 Zillow: Number of accessed tuples vs the number of skylines 196
6.23 Example: Early termination 201
6.24 Nodes traversed by TOP-DOWN Algorithm 201
6.25 Expected number of nodes queried vs. query length 207

LIST OF TABLES

Table Page

2.1 Summary of Online Experiments 51

4.1 Parameter values used in the experiments 99

5.1 Feasibility, Worst- and Practical Query Cost 112

5.2 Summary of Online Experiments 154

6.1 A sample hosts dataset 162

6.2 Table of notations 168

6.3 Example 1 relation 170

6.4 Example: Input Table 181

CHAPTER 1

Introduction

In recent years we have seen an increase in the popularity of many web applications. The functionality of these applications range from allowing users to interact using online social network, to assist users in their everyday activity such as selecting a hotel in an area, locating a nearby restaurant etc. Google Maps, WeChat, FourSquare, AirBnB, TripAdvisor, and Hotels.com are a few such examples. We can conceptualize these applications as client-server model where a client specifies and sends queries via a web interface to a backend database. The public query interfaces of these applications may be abstractly modeled as a kNN interface over the backend database that returns k results that are nearest to the user query. The backend databases of these applications contain information that can be beneficial for the third-party users to build additional services over the data. However, many of these applications impose certain interface restrictions: One is the aforementioned top-k restriction (i.e., only the k nearest tuples are returned). Another common one is a *query rate limit* - i.e., many applications limit the maximum number of kNN queries one can issue per unit of time.

In this dissertation, we present efficient techniques for exploratory analysis over web database. First, we consider a special category of application know as *Location based services* (LBS) and propose algorithms to perform *aggregate estimation* and *clustering*

over the backend database. We then study a novel problem of privacy leakage in real-world web databases which is caused by the ranking function design. Finally, we consider the problem of *subspace skyline discovery* over web database containing categorical attributes.

1.1 Aggregate Estimations over Location Based Services

Location based services (LBS) have become very popular in recent years. They range from map services (e.g., Google Maps) that store geographic locations of points of interests (POIs), to online social networks (e.g., WeChat, Sina Weibo, FourSquare) that leverage user geographic locations to enable various recommendation functions. The underlying data model of these services may be viewed as a database of tuples that are either POIs (in case of map services) or users (in case of social networks), along with their geographical coordinates (e.g., latitude and longitude) on a plane.

For many interesting third-party applications, it is important to collect *aggregate statistics* over the tuples contained in such hidden databases − such as *sum*, *count*, or *distributions* of the tuples satisfying certain selection conditions. However, third-party applications and/or end users do not have complete and direct access to this entire database. The database is essentially "hidden", and access is typically limited to a restricted public web query interface or API by which one can specify an arbitrary location as a query, which returns at most k nearest tuples to the query point (where k is typically a small number such as 10 or 50). Of course, such aggregate information can be obtained by entering into data sharing agreements with the location-based service providers, but this approach can often be extremely expensive, and sometimes impossible if the data owners are unwilling to share their data.

In this work, we consider the problem of obtaining *approximate estimates* of such aggregates by only querying the database via its restrictive public interface. First, we cat-

egorize the LBS applications based on the type of information that is returned along with the k tuples. Some services (e.g., Google maps) return the locations (i.e., the x and y coordinates) of the k returned tuples. We refer to such services as *Location-Returned LBS* (LR-LBS). Other services (e.g., WeChat, Sina Weibo) return a ranked list of k nearest tuples, but suppress the location of each tuple, returning only the tuple ID and perhaps some other attributes (such as tuple name). We refer to such services as *Location-Not-Returned LBS* (LNR-LBS). For each category, we developed efficient sampling algorithms such that one can adhere to the rate limits or budgetary constraints imposed by the interface, and yet make the aggregate estimations as accurate as possible.

1.2 Efficient Analytics over Location Based Services

We demonstrate ANALOC, a web based system that enables fast analytics over an LBS by issuing a small number of queries through its restricted kNN interface. ANALOC stands in sharp contrast with existing systems for analyzing geospatial data, as those systems mostly assume complete access to the underlying data. Specifically, ANALOC supports the *approximate processing* of a wide variety of SUM, COUNT and AVG aggregates over user-specified selection conditions. In the demonstration, we shall not only illustrate the design and accuracy of our underlying aggregate estimation techniques, but also showcase how these estimated aggregates can be used to enable exciting applications such as hotspot detection, infographics, etc. Our demonstration system is designed to query real-world LBS (systems or modules) such as Google Maps, WeChat and Sina Weibo at real time, in order to provide the audience with a practical understanding of the performance of ANALOC.

1.3 Density based Clustering over Location Based Services

The backend database of an LBS is often a gold mine of information for understanding the corresponding application domain. For example, data stored in real estate LBS such as redfin.com offer critical insights into the geographic spread of wealth, education quality, etc., while POI data such as those in Google Maps can support mining the spatial patterns of lifestyle choices such as bar themes, restaurant cuisines, etc. Unfortunately, due to the aforementioned limitations, access to such invaluable data is restricted to the LBS provider itself, making it extremely difficult for unaffiliated *third parties*, e.g., social-science researchers, business analysts, etc., to take advantage of the data. We aim to enable the analytics and mining of such data by using nothing but the restrictive, public-access, interface of the LBS, making it possible for third parties to enjoy the value of LBS data without the lengthy and expensive negotiation process with the LBS provider.

The objective of this work is to study a novel problem of enabling *spatial clustering over an online LBS database* by issuing only a small number of kNN queries supported by the LBS interface. While many spatial clustering algorithms have been studied in the literature, the objective of this work is *not* to select the best-performing algorithm for LBS data, but to instead demonstrate the feasibility of enabling spatial clustering using nothing but a few kNN query answers. For this purpose, we consider as a baseline a fundamental yet popular density-based clustering algorithm, DBSCAN [1], and develop a DBSCAN-like algorithm for LBS data with only a kNN interface for data access.

We first develop algorithms for the special case of one dimensional data, and then extend them to two (and higher) dimensions. In the 1D case, each cluster is essentially a dense segment, and our goal is to discover the boundaries of each dense segment. Our algorithm starts from a dense point within a cluster and discovers the two boundary points by going to the left and right using a binary search-like process. For the 2D case, we use

an innovative approach of mapping the points in 2D space to 1D using a *space filling curve* (SFC [2–4]), and then discover the clusters using the 1D clustering algorithm.

1.4 Privacy Implications of Database Ranking

While traditional structured databases generally support the Boolean Retrieval model (i.e., return all tuples that exactly match the search query selection condition), in recent years there has been much research into exploring the applicability of an alternate Ranked Retrieval model (e.g., a kNN interface that returns top-k tuples according to a suitable ranking function). The ranked retrieval model has become an important component of many databases, especially in a client-server environment (e.g., web databases, where a client specifies and sends queries via a web interface to a backend database). In this work, we investigate a novel problem on the *privacy implications* of database ranking, which has not been studied before. We show how privacy leakage (through the top-k interface) can be caused by a seemingly innocent design of the ranking function in such ranked retrieval models.

To understand how the privacy leakage occurs, note that many databases in a client-server environment feature both public and *private* attributes. For example, social networking websites often allow users to specify privacy settings that hide certain attributes from the public's view, e.g., profile demographics such as race, gender, income; location; past posts, etc. These websites honor the privacy settings by omitting the private attributes from being displayed in the returned query answers. Thus, the results include a ranked list of k tuples, but with only the public attributes displayed, and the private attributes hidden. The problem here, however, is that many websites indeed include these private attributes as *input* to the ranking function. From the privacy perspective, this design might look harmless as well - after all, while a ranking function might take as input a large number of attributes,

its output is merely the (relative) rank of a tuple among returned results - not even the actual ranking score! Naturally, the traditional belief here is that it is impossible to infer private attribute values from just the ranking of a returned tuple.

In our investigation of real-world client-server databases (including popular web databases), we found this traditional belief to be *wrong*. Specifically, in this work, we develop a novel technique which, by asking a carefully constructed sequence of top-k queries and observing the corresponding change of tuple ranks in the query answers, may successfully infer the value of private attributes.

1.5 Subspace Skyline over Categorical Domains

Skyline queries are widely used in applications involving multi-criteria decision making [5], and are further related to well-known problems such as top-k queries [6], preference collection [7], and nearest neighbor search [8]. Given a set of tuples, skylines are computed by considering the dominance relationships among them. A tuple p *dominates* another tuple q, if q is not better than p in any dimension and p is better than q in at least one dimension. The *Skyline* is the set of tuples that are not dominated by any other tuple in the dataset [9].

In recent years, several applications have gained popularity in assisting users in tasks ranging from selecting a hotel in an area to locating a nearby restaurant. AirBnB, TripAdvisor, hotels.com, Craigslist, and Zillow are a few such examples. The underlying datasets have numerous attributes that are mostly Boolean or categorical. They also include a few numeric attributes, but in most cases the numeric attributes are discretized and transformed into categorical attributes [10]. In this work, we consider the problem of *subspace skyline discovery* over such datasets, in which given an ad-hoc subset of attributes as a query, the goal is to identify the tuples in the skyline involving only those attributes[1]. Such subspace

[1]Naturally this definition includes skyline discovery over all attributes of a relation.

skyline queries are an effective tool in assisting users in data exploration (e.g., an AirBnB customer can explore the returned skyline to narrow down to a preferred host).

In accordance with common practice in traditional database query processing, we design solutions for two important practical instances of this problem, namely: (a) assuming that no indices exist on the underlying dataset, and (b) assuming that indices exist on each individual attribute of the dataset. For the case when no indices are available, we design a tree structure to arrange the tuples in a "candidate skyline" set. The tree structure supports efficient dominance tests over the candidate set, thus reducing the overall cost of skyline computation. We then propose two novel algorithms called **ST-S** (Skyline using Tree Sorting-based) and **ST-P** (Skyline using Tree Partition-based) that incorporate the tree structure into existing sorting- and partition-based algorithms. Then, we utilize precomputed sorted lists [11] and design efficient algorithms for the index-based version of our problem. As one of the main results of our work, we propose the Threshold Algorithm for Skyline (**TA-SKY**) capable of answering subspace skyline queries.

CHAPTER 2

Aggregate Estimations over Location Based Services

Location based services (LBS) have become very popular in recent years. They range from map services (e.g., Google Maps) that store geographic locations of points of interests, to online social networks (e.g., WeChat, Sina Weibo, FourSquare) that leverage user geographic locations to enable various recommendation functions. The public query interfaces of these services may be abstractly modeled as a kNN interface over a database of two dimensional points on a plane: given an arbitrary query point, the system returns the k points in the database that are nearest to the query point. In this paper [12] we consider the problem of obtaining *approximate estimates* of SUM and COUNT aggregates by only querying such databases via their restrictive public interfaces. We distinguish between interfaces that return location information of the returned tuples (e.g., Google Maps), and interfaces that do not return location information (e.g., Sina Weibo). For both types of interfaces, we develop aggregate estimation algorithms that are based on novel techniques for precisely computing or approximately estimating the Voronoi cell of tuples. We dis-

cuss a comprehensive set of real-world experiments for testing our algorithms, including experiments on Google Maps, WeChat, and Sina Weibo.

2.1 Introduction

2.1.1 LBS with a kNN Interface

Location based services (LBS) have become very popular in recent years. They range from map services (e.g., Google Maps) that store geographic locations of points of interests (POIs), to online social networks (e.g., WeChat, Sina Weibo, FourSquare) that leverage user geographic locations to enable various recommendation functions. The underlying data model of these services may be viewed as a database of tuples that are either POIs (in case of map services) or users (in case of social networks), along with their geographical coordinates (e.g., latitude and longitude) on a plane.

However, third-party applications and/or end users do not have complete and direct access to this entire database. The database is essentially "hidden", and access is typically limited to a restricted public web query interface or API by which one can specify an arbitrary location as a query, which returns at most k nearest tuples to the query point (where k is typically a small number such as 10 or 50). For example, in Google maps it is possible to specify an arbitrary location and get the ten nearest Starbucks. Thus, the query interfaces of these services may be abstractly modeled as a "nearest neighbor" kNN interface over a database of two dimensional points on a plane: given an arbitrary query point, the system returns the k points in the database that are nearest to the query point.

In addition, there are important differences among the services based on the type of information that is returned along with the k tuples. Some services (e.g., Google maps) return the locations (i.e., the x and y coordinates) of the k returned tuples. We refer to such services as *Location-Returned LBS* (LR-LBS). Other services (e.g., WeChat, Sina Weibo)

return a ranked list of k nearest tuples, but suppress the location of each tuple, returning only the tuple ID and perhaps some other attributes (such as tuple name). We refer to such services as *Location-Not-Returned LBS* (LNR-LBS).

Both types of services impose additional querying limitations, the most important being a per user/IP limit on the number of queries one can issue over a given time frame (e.g., by default, Google map API imposes a query rate limit of 10,000 per user per day).

2.1.2 Aggregate Estimations

For many interesting third-party applications, it is important to collect *aggregate statistics* over the tuples contained in such hidden databases – such as *sum*, *count*, or *distributions* of the tuples satisfying certain selection conditions. For example, a hotel recommendation application would like to know the average review scores for Marriott vs Hilton hotels in Google Maps; a cafe chain startup would like to know the number of Starbucks restaurants in a certain geographical region; a demographics researcher may wish to know the gender ratio of users of social networks in China etc.

Of course, such aggregate information can be obtained by entering into data sharing agreements with the location-based service providers, but this approach can often be extremely expensive, and sometimes impossible if the data owners are unwilling to share their data. Therefore, in this paper we consider the problem of obtaining *approximate estimates* of such aggregates by only querying the database via its restrictive public interface. Our goal is to minimize *query cost* (i.e., ask as few queries as possible) in an effort to adhere to the rate limits imposed by the interface, and yet make the aggregate estimations as accurate as possible.

The closest prior work is [13]. This approach is based on generating random point queries, estimating the area of the *Voronoi cell* [14] of the returned top-1 tuple for each query, and estimating the aggregate from these top-1 tuples by making corrections for sam-

pling bias using the area of the Voronoi cell. However, there are several limitations of this work. First, this approach works only for LR-LBS, but does not work for LNR-LBS, and is thus inapplicable over a large variety of location based services such as WeChat and Sina Weibo that do not return precise location or distance information. Second, the approximate nature of the technique used for estimating the area of a Voronoi cell makes the overall aggregate estimation *inherently biased*. Third, the method only uses the top-1 returned tuple for each query in its calculations (the remaining $k-1$ tuples are ignored) thus leading to inefficiency in the estimation procedure. We discuss this and other related work in §2.7.

2.1.3 Outline of Technical Results

Results over LR-LBS Interfaces: We first describe our results over LR-LBS interfaces. Like [13], our approach is also based on generating random point queries and computing the area of Voronoi cells of the returned tuples, but a key differentiator is that our approach provides an efficient yet *precise* computation of the area of Voronoi cells. This procedure is fundamentally different from the approximate procedure used in [13] for estimating the area of Voronoi cells, and is one of the significant contributions of our paper. This leads to *unbiased estimations* of SUM and COUNT aggregates over LR-LBS interfaces; in contrast, the estimations in [13] have inherent (and unknown) sampling bias.

Moreover, we also leverage the top-k returned tuples of a query (and not just the top-1) by generalizing to the concept of a *top-k Voronoi cell*, leading to significantly more efficient aggregate estimation algorithms. We also developed four different techniques for reducing the estimation error (and thereby estimation error) over LR-LBS interfaces: faster initialization, leveraging history, variance reduction through dynamic selection of query results, and a Monte Carlo method which leverages current knowledge of upper/lower bounds on the Voronoi cell without sacrificing the unbiasedness of estimations.

We combine the above ideas to produce Algorithm LR-LBS-AGG, a completely unbiased estimator for COUNT and SUM queries with or without selection conditions. We note that AVG queries can be computed as SUM/COUNT.

Results over LNR-LBS Interfaces: We also consider the problem of aggregate estimations over LNR-LBS interfaces. To the best of our knowledge, this is a novel problem with no prior work. Recall that such type of kNN interfaces only return a ranked list of the top-k tuples in response to a point query, and location information for these tuples is suppressed. None of the prior work for LR-LBS interfaces can be extended to LNR-LBS interfaces. For such interfaces, we develop aggregate estimation algorithms that are not completely bias-free, but can guarantee an arbitrarily small sampling bias. The key idea here is the inference of a tuple's Voronoi cell to an arbitrary precision level with a small number of queries from *merely the ranks of the returned tuples*.

On a related note, we also show how one can infer the position of a tuple in LNR-LBS, again at a level of arbitrary precision - a problem, while of independent interest, is also critical for enabling the estimations of aggregates that feature selection conditions on tuples' locations (e.g., the COUNT of social network users within 10 meters of major highways). We also study a subtle extension to cases where $k > 1$; in particular we study the challenge brought by this case by the (possibly) concave nature of top-k Voronoi cells, and develop an efficient algorithm to detect potential concaveness and guarantee the accuracy of the inferred Voronoi cell.

We combine the above ideas to produce Algorithm LNR-LBS-AGG, an estimator for COUNT and SUM queries with or without selection conditions. Unlike Algorithm LR-LBS-AGG, this estimator may be biased, but the bias can be controlled to any arbitrary desired precision. As before, we note that AVG queries can be computed as SUM/COUNT.

2.1.4 Summary of Contributions

- Location based services have become very popular in recent years, and aggregate estimation over such "hidden" databases with their restricted kNN query interfaces is an important problem with numerous applications. In our work, we consider both LR-LBS (locations returned) as well as the more novel LNR-LBS (locations not returned) interfaces.
- For LR-LBS interfaces, we develop Algorithm LR-LBS-AGG for estimating COUNT and SUM aggregates with or without selection conditions. It represents a significant improvement over prior work along multiple dimensions: a novel way of precisely calculating Voronoi cells lead to completely unbiased estimations; top-k returned tuples are leveraged rather than only top-1; several innovative techniques developed for reducing error and increasing efficiency.
- For LNR-LBS interfaces, we develop Algorithm LNR-LBS-AGG for estimating COUNT and SUM aggregates with or without selection conditions.This is a novel problem with no prior work. The estimated aggregates are not bias-free, but the sampling bias can be controlled to any desired precision. Among several key ideas, we show how a Voronoi cell can be inferred to an arbitrary degree of precision from merely the ranks of returned tuples to point queries.
- Our contributions also include a comprehensive set of real-world experiments. Specifically, we conducted online tests over a number of real-world LBS, e.g., Google Maps (LR-LBS) for estimating the number of Starbucks in US (and compared the results with the ground truth published by Starbucks); WeChat and Sina Weibo for estimating the percentage of male/female users in China.

2.2 Background

2.2.1 Model of LBS

A location based service (LBS) supports kNN queries over a database D of tuples with location information. These tuples can be points of interest (e.g. Google Maps) or users (e.g. WeChat, Sina Weibo). A kNN query q takes as input a location (e.g., longitude/latitude combination), and returns the top-k nearest tuples selected and ranked according to a pre-determined ranking function. Since the only input to a query is a location, we use q to also denote the query's location without introducing ambiguity. Most of the popular LBS follow kNN query model. For most parts of the paper, we consider the ranking function to be Euclidean distance between the query location and each tuple's location. Extensions to other ranking functions are discussed in §2.5.3.

Note that tuples in an LBS system contain not only location information but other many other attributes - e.g., a POI in Google Maps includes attributes such as POI name, average review ratings etc. Depending on which attributes of a tuple are returned by the kNN interface - more specifically, whether the location of a tuple is returned - we can classify LBS into two main categories:

LR-LBS: A Location-Returned-LBS (LR-LBS) returns the precise location for each of the top-k returned tuples, along with possibly other attributes. Google Maps, Bing Maps, Yahoo! Maps, etc., are prominent examples of LR-LBS, as all of them display the precise location of each returned POI. Note that some LBS may return a variant of the precise locations - e.g., Skout and Momo returns not the precise location of a tuple, but the precise distance between the query location and the tuple location. We consider such LBS to be in the LR-LBS category because, through previously studied techniques such as trilateration (e.g., [15]), one can infer the precise location of a tuple with just 3 queries.

LNR-LBS: A Location-Not-Returned-LBS (LNR-LBS), on the other hand, does *not* return tuple locations - i.e., only other attributes such as name, review rating, etc., are returned. This category is more prevalent among location based social networks (presumably because of potential privacy concerns on precise user locations). Examples here include WeChat, which returns attributes such as name, gender, etc., for each of the top-k users, but not the precise location/distance. Other examples include Sina Weibo, WeChat, etc., which feature a similar query return semantics.

Common Interface Features and Limitations: Generally speaking, there are two ways through which an LBS (either LR- or LNR-LBS) supports a kNN query. One is an interactive web or API interface which allows a location to be explicitly specified as a latitude/longitude pair. Google Maps is an example to this end. Another common way is for the LBS (e.g., as a mobile app) to directly retrieve the query location from a positioning service (such as GPS, WiFi or Cell towers) and automatically issue a kNN query accordingly. In the second case, there is no explicit mechanism to enter the location information. Nonetheless, it is important to note that, even in this case, we have the ability to issue a query from any arbitrary location *without* having to physically travel to that location. All mobile OS have debugging features that allow arbitrary location to be used as the output of the positioning (e.g., GPS) service.

Many LBS also impose certain interface restrictions: One is the aforementioned top-k restriction (i.e., only the k nearest tuples are returned). Another common one is a *query rate limit* - i.e., many LBS limit the maximum number of kNN queries one can issue per unit of time. For example, by default Google Maps allows 10,000 location queries per day while Sina Weibo allows only 150 queries per hour. This is an important constraint for our purpose because it makes the *query-issuing* process the bottleneck for aggregate estimation. To understand why, note that even with the generous limit provided by Google

Maps, one can issue only 7 queries per minute - this 8.6 second per query overhead[1] is orders of magnitude higher than any offline processing overhead that may be required by the aggregate estimation algorithm. Thus, this interface limitation essentially makes *query cost* the No. 1 performance metric to optimize for aggregate estimation. An LBS might impose other, more subtle constraints - e.g., a maximum coverage limit which forbids tuples far away (say more than 5 miles away) from a query location to be returned. We shall discuss about these constraints in §2.5.3.

2.2.2 Voronoi Cells

Voronoi cell [14] is a key geometry concept used extensively by our algorithms developed in the paper. Thus, we introduce this concept here as part of the preliminaries. Consider each tuple $t \in D$ as a point on a Euclidean plane bounded by a box B (which covers all tuples in D). We have the following definition.

Definition 1 (Voronoi Cell). *Given a tuple $t \in D$, the Voronoi cell of t, denoted by $V(t)$, is the set of points on the B-bounded plane that are closer to t than any other tuple in D.*

Note that the B-bound ensures that each Voronoi cell is a finite region. The Voronoi cells of different tuples are mutually exclusive - i.e., the *Voronoi diagram* is the subdivision of the plane into regions, each corresponding to all query locations that would return a certain tuple as the nearest neighbor[2].

For the purposes of our paper, we define an extension of the Voronoi cell concept to accommodate the top-k (when $k > 1$) query return semantics. Specifically, given a tuple $t \in D$, we define the *top-k Voronoi cell* of t, denoted by $V_k(t)$, as the set of query locations

[1] Of course, one can shorten it with multiple IP addresses and API accounts - but similarly, one can use parallel processing to speed up offline processing as well.

[2] We assume general positioning [14] - i.e., no two tuples have the exact same location and no four points on the same circle.

on the plane that return t as one of the top-k results. There are four important observations about this concept:

First, the top-k Voronoi cells for different tuples are no longer mutually exclusive. Each location l belongs to exactly k top-k Voronoi cells corresponding to the top-k tuples returned by query over l. Second, our concept of top-k Voronoi cells is *fundamentally different* from the k-th ordered Voronoi cells in geometry [14] - each of which is formed by points with the exact same k closest tuples. The difference is illustrated in Figure 2.1 - while each colored region is a k-th ordered Voronoi cell, a top-k Voronoi cell may cover multiple regions with different colors. For example, the top-2 Voronoi cell for tuple A is marked by a red border and consists of two different k-th ordered Voronoi cells (AB and AE).

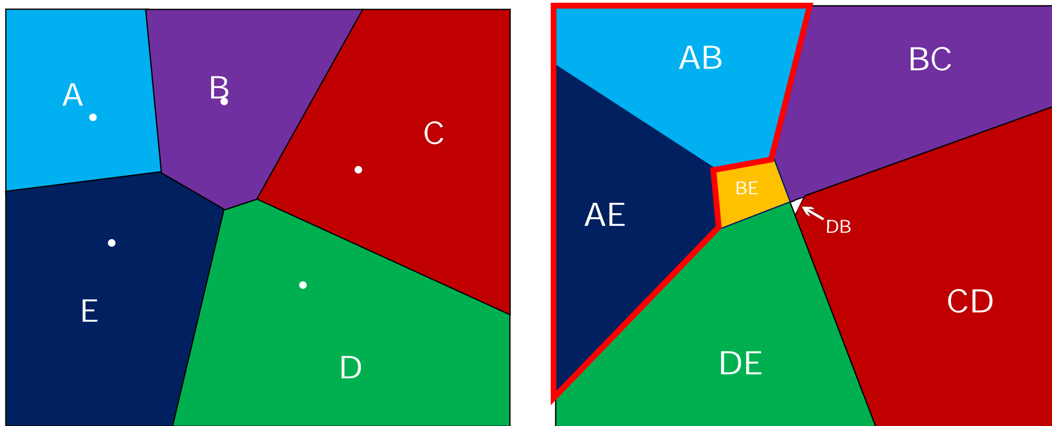


Figure 2.1: Concavity of top-k Voronoi Diagrams

Third, while both top-1 Voronoi cells and k-th order Voronoi cells are guaranteed to be convex [14], the same does not hold for top-k Voronoi cells when $k > 1$. For example, from Figure 2.1 we can see that the aforementioned top-2 Voronoi cell for tuple A is concave. Fourth, a top-k Voronoi cell tend to contain many more edges than a top-1 Voronoi

cell. As we shall discuss later in the paper, the larger number of edges and the potential concaveness makes computing the top-k Voronoi cell of a tuple t more difficult.

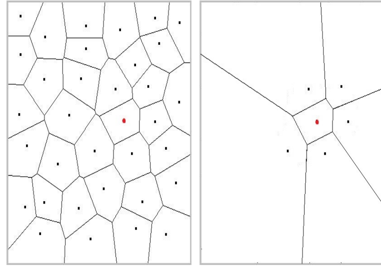


Figure 2.2: Illustration of Theorem 1

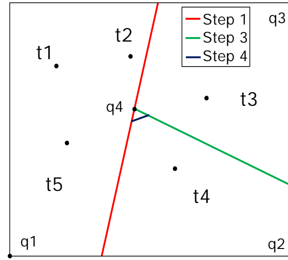


Figure 2.3: Illustration of LR-LBS-AGG

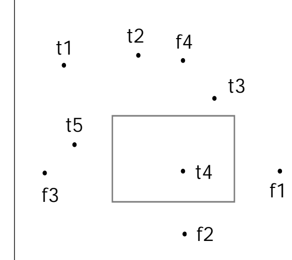


Figure 2.4: Faster Initialization - Success

2.2.3 Problem Definition

In this paper, we address the problem of aggregate estimations over LBS. Specifically, we consider aggregate queries of the form SELECT AGGR(t) FROM D WHERE *Cond* where AGGR is an aggregate function such as SUM, COUNT and AVG that can be evaluated over one or more attributes of t, and *Cond* is the selection condition. Examples include the COUNT of users in WeChat or AVG rating of restaurants in Texas at Google Maps.

There are two important notes regarding the selection condition *Cond*. First, we support any selection condition that can be independently evaluated over a single tuple - i.e., it is possible to determine whether a tuple t satisfies *Cond* based on nothing but t. Second, for both LR- and LNR-LBS, we support the specification of a tuple's location as part of *Cond* - even when such a location is not returned, like in LNR-LBS. This is possible thanks to what we shall discuss in §2.4.3 - i.e., even with LNR-LBS, one can derive the location of a tuple to arbitrary precision after issuing a small number of queries. As such, we support aggregates such as the percentage of female WeChat users in Washington, DC).

In most part of the technical sections, we focus on aggregates without selection conditions - the straightforward extensions to various types of selection conditions will be discussed in §2.5.

Performance Measures: The performance of an aggregate estimation algorithm is measured in terms of efficiency and accuracy. Given the query-rate limit enforced by all LBS, the efficiency is measured by *query cost* - i.e. the number of queries and/or API calls that the algorithm issues to LBS. Often, we are given a fixed budget (based on the rate limits) and hence designing an efficient algorithm that generates accurate estimates within the budgetary constraints is crucial. The accuracy of an estimation $\tilde{\theta}$ of an aggregate θ could be measured by the standard measure of *relative error* $|\tilde{\theta} - \theta|/\theta$. Note that, for any sampling-based approach (like ours), the relative error is determined by two factors: *bias*, i.e. $|E(\tilde{\theta} - \theta)|$, and *variance* of $\tilde{\theta}$. The mean squared error, MSE of the estimation is computed as MSE = bias^2 + variance.

An interesting question often arises in practice is how we can determine the relative error achieved by our estimation. If the population variance is known, then one can apply standard statistics techniques to compute the confidence interval of aggregate estimations [16]. Absence of such knowledge, a common practice is to approximate the population variance with *sample* variance, which can be computed from the samples we use to generate the final estimation and use Bessel's correction [16] to correct the result.

2.3 LR-LBS-AGG

In this section, we develop LR-LBS-AGG, our algorithm for generating unbiased SUM and COUNT estimations over an LR-LBS query interface. Specifically, we start with introducing our key idea of precisely computing the (top-k) Voronoi cell of a given tuple, which enables the unbiased aggregate estimations. While this idea guarantees unbi-

asedness, it may require a large number of queries per (randomized) estimation, leading to a large estimation variance (and therefore, error) when the query budget is limited. Hence we develop four techniques for reducing the estimation error while *maintaining* the complete unbiasedness of aggregate estimations. Finally, we combine all ideas to produce Algorithm LR-LBS-AGG at the end of this section.

2.3.1 Key Idea: Precisely Compute Voronoi Cells

Reduction to Computing Voronoi Cells: We start by describing a baseline design which illustrates why the problem of aggregate estimations over an LBS's kNN interface ultimately boils down to computing the volume of the Voronoi cell corresponding to a tuple t. As an example, consider the estimation of COUNT(*) (over a given region) through an LR-LBS with a top-1 interface.

We start by choosing a location q uniformly at random from the region, and then issue a query at q. Let t be the tuple returned by q. Suppose that we can compute the Voronoi cell of t (as defined in §2.2), say $V(t)$. A key observation here is that the sampling probability of t, i.e., the probability for the above-described randomized process to return t, is exactly $p(t) = \frac{|V(t)|}{|V_0|}$ where $|V(t)|$ and $|V_0|$ are the volume of $V(t)$ and the entire region, respectively. Note that knowledge of $p(t)$ directly leads to a completely unbiased estimation of COUNT(*): $r = 1/p(t)$, because

$$\text{Exp}(r) = \sum_{t \in D} p(t) \cdot \frac{1}{p(t)} = |D|, \quad (2.1)$$

where $\text{Exp}(\cdot)$ is the expected value of the estimation (taken over the randomness of the estimation process), and $|D|$ is the total number of tuples in the database. From (2.1), one can see that every SUM and COUNT aggregate we support can be estimated without bias - the only change required is on the numerator of estimation. Instead of having 1 as in the COUNT(*) case, it should be the evaluation of the aggregate over t - e.g., if we need

to estimation SUM(A_1) where A_1 is an attribute, then the numerator should be $t[A_1]$, i.e., the value of A_1 for t. If the aggregate is COUNT with a selection condition, then the numerator should be either 1 if t satisfies the condition, or 0 if it does not. One can see from the above discussions that, essentially, the problem of enabling unbiased SUM and COUNT estimations is reduced to that of *precisely* computing the volume of $V(t)$, i.e., the Voronoi cell of a given tuple t.

Computing Voronoi Cells: For computing the Voronoi cell of a given tuple, a nice feature of the LR-LBS interface is that it returns the precise location of every returned tuple. Clearly, if we can somehow "collect" all tuples with Voronoi cells adjacent to that of t, then we can precisely compute the Voronoi cell of t based on the locations of these tuples (and t). As such, the key challenges here become: (1) how do we collect these tuples and (2) how do we know if/when we have collected all tuples with adjacent Voronoi cells to t? Both challenges are addressed by the following theorem which forms the foundation of design of Algorithm LR-LBS-AGG.

Theorem 1. *Given a tuple $t \in D$ and a subset of tuples $D' \subseteq D$ such that $t \in D'$, the Voronoi cell of t defined according to D', represented by P', is the same as that according to the entire dataset D, denoted by P, if and only if for all vertices v of P', all tuples returned by the nearest neighbor query issued at v over D belong to D'.*

Proof. First, note that there must be $P \subseteq P'$, because for a given location q, if there is already a tuple t' in D' that is closer to q than t, then there must at least one tuple in D that is closer to q than t. Second, if $P \neq P'$ (i.e., $P \subset P'$), then there must at least one vertex of P', say v, that falls outside P. i.e. there must exist a tuple $t_0 \in (D \backslash D')$ that is closer to v than all tuples in D'. □

Example 1: Figure 2.2 provides an illustration for Theorem 1. In order to compute the Voronoi cell of the tuple corresponding to the red dot, it suffices to know the location

of the adjacent tuples. Since each Voronoi edge is a perpendicular bisector between the adjacent tuples, the entire Voronoi cell can be computed as the convex shape induced by the intersections of the edges.

Theorem 1 answers both challenges outlined above: it tells us when we have collected all "adjacent" tuples - when all vertices of t's Voronoi cell computed from the collected tuples return only collected tuples. It also tells us how to collect more "adjacent" tuples when not all of them have been collected - any vertex which fails the test naturally returns some tuples that have not been collected yet, adding to our collection and starting the next round of tests.

According to the theorem, a simple algorithm for constructing the exact Voronoi cell for t is as follows: We start with $D' = \{t\}$. Now the Voronoi cell is the entire region (say an extremely large rectangle). We issue queries corresponding to its four vertices. If any query returns a point we have not seen yet - i.e., not in D' - we append it to D', recompute the Voronoi cell, and repeat the process. Otherwise, if all queries corresponding to vertices of the Voronoi cell return points in D', we have obtained the real Voronoi cell for $t \in D$. One can see that the query complexity of this algorithm is $O(n)$, where n is the number of points in the database D, because each query issued either confirms a vertex of the final Voronoi cell (which has at most $n - 1$ vertices), or returns us a new point we have never seen before (there are at most $n - 1$ of these too). It is easy to see that the bound is tight - as one can always construct a Voronoi cell that has $n-1$ edges and therefore requires $\Omega(n)$ top-1 queries to discover (after all, each such query returns only 1 tuple). An example here is when t is in the center of a circle, on which the other $n-1$ points are located. Algorithm 1 shows the pseudocode of the baseline approach which we improve in Section 2.3.2.

Example 2: Figure 2.3 provides a simple run-through of the algorithm for a dataset with 5 tuples $\{t_1, \ldots, t_5\}$. Suppose we wish to compute $V(t_4)$. Initially, we set $D' = \{t_4\}$ and $V(t_4) = V_0$, the entire bounding box. We issue query q_1 that returns tuple t_5 and

Algorithm 1 LNR-LBS-AGG-Baseline

1: **while** query budget is not exhausted
2: q = location chosen uniformly at random; t = query(q)
3: $V(t) = V_0$; $D' = \{t\}$
4: **repeat** till D' does not change between iterations
5: **for** each vertex v of $V(t)$: $D' = D' \cup$ query(v)
6: Update $V(t)$ from D'
7: Produce aggregate estimation using samples

hence $D' = \{t_4, t_5\}$. We now obtain a new Voronoi edge that is the perpendicular bisector between t_4 and t_5. The Voronoi cell after step 1 is highlighted in light grey. In step 2, we issue query q_2 that returns t_4 resulting in no update. In step 3, we issue query q_3 that returns t_3. $D' = \{t_3, t_4, t_5\}$ and we obtain a new Voronoi edge as the perpendicular bisector between t_3 and t_4 depicted in dark medium gray. In step 4, we issue query q_4 that returns t_2 resulting in the final Voronoi edge depicted in dark grey. Further queries over the vertices for $V(t_4)$ does not result in new tuples concluding the invocation of the algorithm.

Extension to $k > 1$: Interestingly, no change is required to the above algorithm when we consider the top-k Voronoi cell rather than the traditional, i.e., top-1 Voronoi cell. To understand why, note that Theorem 1 directly extends to top-k Voronoi cells - as a top-k Voronoi computed from D' still must completely cover that for D; and any vertex of the top-k Voronoi from D' which is outside that from D must return at least one tuple outside D'. We further describe how to leverage $k > 1$ in Sections 2.3.2.3 and 2.4.2.

2.3.2 Error Reduction

Before describing the various error reduction techniques we develop for aggregate estimations over LR-LBS, we would like to first note that, while we use the term "error

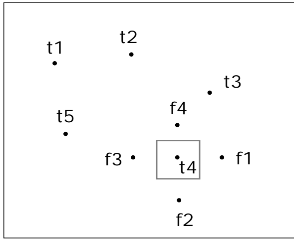


Figure 2.5: Faster Initialization - Failure

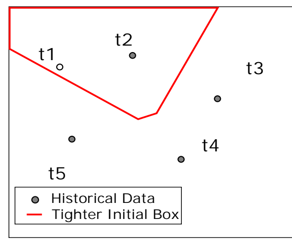


Figure 2.6: Leveraging History

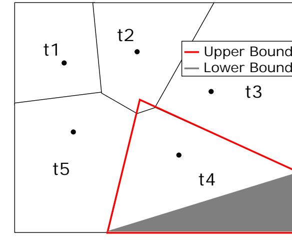


Figure 2.7: Upper/Lower Bounds

reduction" as the title of this subsection, some of the techniques described below indeed focus on making the computation of a Voronoi cell more efficient. The reason why we call all of them "error reduction" is because of the inherent relationship between efficiency and estimation error - if the Voronoi-cell computation becomes more efficient, then we can do so for more samples, leading to a larger sample size and ultimately, a lower estimation error (which is inversely proportional to the square root of sample size [16]).

2.3.2.1 Faster Initialization

A key observation from the design in §2.3.1 is its bottleneck: the initialization process. At the beginning, we know nothing about the database other than (1) the location of tuple t, and (2) a large bounding box corresponding to the area of interest for the aggregate query. Naturally, $D' = \{t\}$, leading to the initial Voronoi cell being the bounding box, and our first four queries being the corners of these bounding boxes. Of course, the tentative Voronoi cell will quickly close in to the real one with speed close to a binary search - i.e., the average-case query cost is at $\log$ scale of the bounding box size. Nonetheless, the initialization process can still be very costly, especially when the bounding box is large.

To address this problem, we develop a *faster initialization* technique which features a simple idea: Instead of starting with $D' = \{t\}$, we insert four fake tuples into D', say $D' = \{t, t_1^{\mathrm{F}}, \ldots, t_4^{\mathrm{F}}\}$, where t_1^{F}, …, t_4^{F} form a bounding box around t. The size of the

bounding box should be conservatively large - even though a wrongly set size will not jeopardize the accuracy of our computation - as we shall show next.

By computing the initial Voronoi cell from D' and then issue queries corresponding to its vertices, there are two possible outcomes: One is that these queries return enough real tuples (besides t, of course) that, after excluding the fake ones from D', we still get a bounded Voronoi cell for t. One can see that, in this case, we can simply continue the computation while having saved a significant number of initialization queries. The other possible outcome, however, is when the bounding box is set too small, and we do not have enough real tuples to "bound" t with a real Voronoi cell. Specifically, in the extreme-case scenario, all four vertices of the initial Voronoi cell could return t itself. In this case, we simply revert back to the original design, wasting nothing but four queries.

One can see that the faster initialization process still guarantees the exact computation of a tuple's Voronoi cell. It has the potential to save a large amount of initialization queries in the average-case scenario, while in the worst case, it wastes at most four queries. Algorithm 2 provides the pseudocode for faster initialization strategy.

Algorithm 2 Fast-Init

1: **Input:** t; **Output:** $V(t)$
2: $D' = \{t, t_1^F, t_2^F, t_3^F, t_4^F\}$; Update $V(t)$ based on D'
3: **If** all queries over vertices of $V(t)$ return t, **then return** V_0
4: **repeat** till D' does not change between iterations
5: **for** each vertex v of $V(t)$: $D' = D' \cup$ query(v)
6: Update $V(t)$ from D'
7: **return** $V(t)$

Example 3: Figures 2.4 and 2.5 show two different scenarios where the strategy is successful and not successful respectively based on whether the bounding box due to fake tuples is conservatively large. Given a small dataset with tuples $\{t_1, \ldots, t_5\}$, we initialize them with a bounding box corresponding to fake tuples $\{f_1, \ldots, f_4\}$. In Figure 2.4, the initial bounding box is tight enough and results in the computation of the precise $V(t_4)$ with much lower query cost (i.e. only tuples $\{t_3, t_5\}$ are visited as against tuples $\{t_2, t_4, t_5\}$ for the example of Algorithm 1. On the other hand, if the bounding box is not tight (as in Figure 2.5), then queries over all the vertices of the bounding box return t_4. We then revert back to the original bounding box V_0 that covers the entire region.

2.3.2.2 Leverage history on Voronoi-cell computation

Another natural optimization is to leverage the information that is gleaned from computing the Voronoi cells of past tuples to compute a tighter initial Voronoi cell. Recall that our algorithm to compute Voronoi cell of a tuple t (i.e $V(t)$), using Theorem 1 starts with an initial Voronoi cell that is an extremely large bounding box that covers the entire plane that then converges to $V(t)$. In the process of computing this Voronoi cell, our algorithm retrieved additional new tuples (by issuing queries for each vertex of the bounding box). Notice that for a LBS with static tuples (such as POIs in Google Maps), the results of location query ordered by distance remains static. Hence it is not necessary to restart every iteration of the algorithm with the same large bounding box. Specifically, when computing the Voronoi cell for the next tuple, we could leverage history by starting with a "tighter" initial bounding box whose vertices are the set of tuples that we have seen so far. In other words, we reuse the tuples that we have seen so far and make them as input to further rounds. Notice that this approach remains the same for both $k = 1$ and $k > 1$. Since the location of each tuple in top-k are returned in LR-LBS, each of these tuples could be leveraged. As we see more tuples, the initial Voronoi cell becomes more granular resulting

in substantial savings in query cost. Algorithm 3 provides the pseudocode for the strategy. While the pseudocode uses the simple perpendicular bisector half plane approach [14], it could also use more sophisticated approaches such as Fortune's algorithm [14] to compute the bounding box around tuple t using the tuples from historic queries.

Algorithm 3 Leverage-History

1: **Input:** t and H (set of tuples obtained from historic queries)

2: **Output:** Bounding box $V'(t)$

3: $V'(t) = V_0$

4: **for** each tuple $h \in H$

5: Update $V'(t)$ with perpendicular bisector between h and t

6: **return** $V'(t)$ with the tightest bounding box around t

Example 4: As part of computing $V(t_4)$ (see Example 1), we have the locations of $t_2, \ldots, t_5$. Using this information, we can compute the initial bounding box for t_2 (shown in red around t_2 in Figure 2.6) *offline* - i.e. without issuing any queries.

2.3.2.3 Variance reduction with larger k

When the system has $k > 1$, we can of course still choose to use the top-1 Voronoi cell as if only the top result is returned. Or we can choose from any of the top-h Voronoi cells as long as $h \leq k$. While intuitively it might appear that using all k returned tuples is definitely better than using just the top-1, the theoretical analysis suggests otherwise - indeed, whether top-1 or top-h Voronoi cell is better depends on the exact aggregate being estimated - specifically, whether the distribution of the attribute being aggregated is better "aligned" with the size distribution of top-1 or top-h Voronoi cells. To see why, simply consider an extreme-case scenario where the aggregate being estimated is AVG(Salary),

and the salary of each user (tuple) is exactly proportional to the size of its top-1 Voronoi cell. In this case, nothing beats using the top-1 Voronoi cells as doing so produces zero variance and thus zero estimation error.

Having said that, however, many aggregates can indeed be better estimated using top-h Voronoi cells, because the sizes of these top-h cells are more uniform than those of the top-1 cells, which can vary extremely widely (see Figure 2.11 in the experiments section for an example), while many real-world aggregates are also more uniformly distributed than the top-1 cell volume (again, see experiments for justification). But simply increasing h also introduces an undesired consequence: recall from §2.2 that the larger h is, the more "complex" the top-h Voronoi cell becomes - in other words, the more queries we have to spend in order to pin down the exact volume of the Voronoi cell.

Thus, the key is to make a proper tradeoff between the benefit received (i.e., smaller variance per sample) and the cost incurred (i.e., larger query cost per sample). Our main idea is a combination of two methods: leveraging history in §2.3.2.2 and upper/lower bound approximation in §2.3.2.4. Specifically, for each of the k returned tuples, we perform the following process:

Consider t_i returned as the No. i result. We need to decide which version of the Voronoi cell definition to use for t_i. The answer can be anywhere from 1 to k. To make the determination, for all $h \in [2, k]$, we compute $\lambda_h(t_i)$, the upper bound on the volume of the top-k Voronoi cell of t_i, as computed from all historically retrieved tuples. Then, we choose the largest h which satisfies $\lambda_h(t_i) \leq \lambda_0$, where λ_0 is a pre-determined threshold (the intuitive meaning of which shall be elaborated next). Let the chosen value be $h(t_i)$. If none of $h \in [2, k]$ satisfies the threshold, we take $h(t_i) = 1$. Then, if $h(t_i) \leq i$, we compute the top-h Voronoi cell for t_i. The final estimation from the k returned results becomes:

$$\sum_{t_i : h(t_i) \leq i \leq k} \frac{Q(t_i)}{|V_h(t_i)|} \tag{2.2}$$

for any SUM or COUNT query Q, where $|V_h(t_i)|$ is the volume for the top-h Voronoi cell of t_i.

We now explain the intuition behind the above approach, specifically the threshold λ_0. First, note that if the top-h (say top-1) Voronoi cell of t_i is already large, then there is no need to further increase h. The reason can be observed from the above-described justification of variance reduction - note that a large top-1 Voronoi cell translates to a large selection probability p - i.e., a small $1/p$ which adds little to the overall variance. Further increasing h not only contributes little to variance reduction, but might actually increase the variance if $1/p$ is already below the average value.

Second, admittedly, $\lambda_h(t_i)$ is only an upper-bound estimate - i.e., even though we showed above that an already large top-h Voronoi cell does not need to have h further increased, there remains the possibility that $\lambda_h(t_i)$ is large because of an overly loose bound (from history), rather than the real volume of the Voronoi cell. Nonetheless, note that this is still a negative signal for using such a large h - as it means that we have not thoroughly explored the neighborhood of t_i. In other words, we may need to issue many queries in order to reduce our estimation (or computation) of $|V_h(t_i)|$ from $\lambda_h(t_i)$ to the correct value. As such, we may still want to avoid using such a large h in order to reduce the query cost.

While the above explanation is heuristic in nature it is important to note that, regardless of how we set $h(t_i)$, the estimation we produce for SUM and COUNT aggregates in (2.2) is always unbiased.

2.3.2.4 Upper/lower bounds on Voronoi-cell

Note that in the entire process of Voronoi-cell computation (barring the very first step of the faster initialization idea discussed in §2.3.2.1), we maintain a tentative polygon that covers the entire real Voronoi cell - i.e., an upper bound on its volume. What often arises in

Algorithm 4 Variance-Reduction

1: **Input:** H; **Output:** Aggregate estimate from all top-k tuples

2: q = location chosen uniformly at random

3: **for** each tuple t_i returned from query(q)

4: $h(t_i) = \max\{h | h \in [2, k], \lambda_h(t_i) \leq \lambda_0\}$

5: $h(t_i) = 1$ if no h satisfied the condition $\lambda_h(t_i) \leq \lambda_0$

6: Generate estimate for t_i using Equation 2.2

practice, especially when computing top-k Voronoi cells (which tend to have many edges), is that even though the bounding polygon is very close to the real Voronoi cell in volume, it has far fewer edges - meaning we still need to issue many more queries to pin down the exact Voronoi cell.

The key idea we develop here is to avoid such query costs *without* sacrificing the accuracy of our aggregate estimations. Specifically, consider a simple Monte Carlo approach which chooses uniformly at random a point from the current bounding polygon, and then issues a query from that point. If the query returns t - i.e., it is in the Voronoi cell of t, we stop. Otherwise, we repeat this process. Interestingly, the number of trials it takes to reach a point that returns t, say r, is an unbiased estimation of $|V'(t)|/|V(t)|$, where $|V'(t)|$ and $|V(t)|$ are the volumes of the bounding polygon and the real Voronoi cell of t, respectively.

$$\text{Exp}(r) = \sum_{i=1}^{\infty} \left[i \cdot \left(1 - \frac{|V(t)|}{|V'(t)|}\right)^{i-1} \cdot \frac{|V(t)|}{|V'(t)|} \right] = \frac{|V'(t)|}{|V(t)|}.$$

In other words, we can maintain the unbiasedness of our estimation without issuing the many more queries required to pin down the exact Voronoi cell. Instead, when $V'(t)$ is close enough to $V(t)$, we can simply use call upon above-described method which, in most likelihood, requires just one more query to produce an unbiased SUM or COUNT estimation. For example, we can simply multiply the number of trials r by $|V_0|/|V'(t)|$, where $|V_0|$ is the volume of the entire region under consideration, to produce an unbiased

estimation for COUNT(*). Other SUM and COUNT aggregates can be estimated without bias in analogy.

Before concluding this idea, there is one more optimization we can use here: a lower bound on the top-k Voronoi cell of t. In the following, we first discuss how to use such a lower bound to further reduce query cost, and then describe the idea for computing such a lower bound. Note that once we have knowledge of a region R that is covered entirely by the real (top-k) Voronoi cell, if in the above process, we randomly choose a point q (from $V'(t)$) which happens to belong in R, then we no longer need to actually query q - instead, we immediately know that q must belong to $V(t)$ and can produce an unbiased estimation accordingly. This is the cost saving produced by knowledge of a lower bound R.

To understand how we construct this lower bound region, a key understanding is that, at anytime during the execution of our algorithm, we have tested certain vertices of $V'(t)$ which are already confirmed to be part of $V(t)$. Consider such a vertex v. Let $C(v,t)$ be a circle with v being the center and the distance between t and v being the radius. Note that we are guaranteed to have *observed all tuples* within $C(v,t)$. This essentially leads to a lower-bound estimation of $V(t)$. Specifically, a point q is in this lower-bound region if and only if $C(q,t)$, i.e., a circle centered on q with radius being the distance between q and t, is entirely covered by the union of $C(v,t)$ for all vertices v of $V'(t)$ that have been confirmed to be within $V(t)$. As such, for any q in this region, we can save the query on it in the above process.

Example 5: The upper bound $V'(t_4)$ of $V(t_4)$ after Step 3 in the Example 2 (i.e. run-through of Algorithm LR-LBS-AGG-Baseline) is shown in Figure 2.7 as a quadrilateral with red edges. The three lower vertices of $V'(t_4)$ are guaranteed to be in $V(t_4)$ using the criteria described above and hence the polygon induced by them provides a lower bound estimate for $V(t_4)$.

2.3.3 Algorithm LR-LBS-AGG

By combining the baseline idea for precisely computing the Voronoi cells with the 4 techniques for error reduction, we can design an efficient algorithm LR-LBS-AGG for aggregate estimation over LR-LBS. Algorithm 5 shows the pseudocode for LR-LBS-AGG.

Algorithm 5 LR-LBS-AGG

1: **while** query budget is not exhausted
2: q = location chosen uniformly at random
3: **for** each tuple t_i in query(q)
4: Compute optimal h for t_i
5: Construct initial $V_h(t_i)$ using Algorithms 2 and 3
6: D'= vertices of $V_h(t_i)$
7: **repeat** till D' is not updated or Voronoi bound is tight
8: **for** each vertex v of $V_h(t_i)$: $D' = D' \cup$ query(v)
9: Update $V_h(t_i)$ and $V'_h(t_i)$ from D'
10: Produce aggregate estimation using samples

2.4 LNR-LBS-AGG

2.4.1 Voronoi Cell Computation: Key Idea

We now consider the case where only a ranked order of points are returned - but not their locations. We shall start with the case of $k = 1$, and then extend to the general case of $k > 1$.

We start by defining a primitive operation of "binary search" as follows. Consider the objective of finding the Voronoi cell of a tuple t in the database. Given any location c_1 and c_2 (not necessarily occupied by any tuple), where c_1 returns t, consider the half-

line from c_1 passing through c_2. Since a Voronoi cell is convex and c_1 resides within the Voronoi cell, this half-line has one and only one intersection with the Voronoi cell - which is associated with one or two edges of the Voronoi cell. We define the primitive operation of *binary search* for given c_1, c_2 to be the binary search process of finding one Voronoi edge associated with the intersection. Please refer to § 2.9 for the detailed design of this process.

Naturally, such a binary search process is associated with an error bound on the precision of the derived edge. For example, we can set an upper bound ϵ on the maximum distance between any point on the real Voronoi edge (i.e., a line segment) and its closest point on the derived edge, which we refer to as the *maximum edge error*, and use ϵ as the objective of the binary search operation. One can see that the number of queries required for this binary search is proportional to $\log(1/\epsilon)$. See § 2.9 for exact query cost.

Given this definition, we now show that one can discover the Voronoi cell of t (up to whatever precision level afforded to us by the binary search operation) with a query complexity of $O(m \log(1/\epsilon))$, where m is the number of edges for the Voronoi cell. Here is the corresponding process:

We start with one query at point q which returns t. Then, we construct 4 points that bound q (say $q_1 : \langle x(q) - 1, y(q) \rangle$, $q_2 : \langle x(q) + 1, y(q) \rangle$, $q_3 : \langle x(q), y(q) - 1 \rangle$, $q_4 : \langle x(q), y(q) + 1 \rangle$, where $x(\cdot)$ and $y(\cdot)$ are the two dimensions, e.g., longitude and latitude, of a location, respectively) and call upon the binary search operation to find the corresponding Voronoi edges intercepting the half lines from q to $q_1, \ldots, q_4$, respectively. One can see that, no matter what the discovered edges might be, they must form a closed polygon[3] which we can use to initiate the testing process described in §2.3.1. If all vertices pass the test, then we have already obtained the Voronoi cell of t. Otherwise, for each vertex (say v) that fails the test, we perform the binary search operation on the location of v to discover another Voronoi edge. We repeatedly do so until all vertices pass the test - at which time we have

[3]In the extreme-case, some edges of this polygon might be part of the bounding box.

obtained the real Voronoi cell - subject to whatever error bound specified for the binary search process (as described above).

To compute the query cost of this process, a key observation is that each call of the binary search process after the initial step (i.e., a call caused by a vertex failing the test) increases the number of discovered (real) edges for the Voronoi cell by 1. Thus, the number of times we have to call the binary search process is $O(m)$, leading to the overall query-cost complexity of $O(m \log(1/\epsilon))$. For the estimation error, we have the following theorem.

Theorem 2. *The estimation bias for COUNT(*) is at most*

$$|E(\tilde{\theta} - \theta)| \leq \sum_{t \in D} \frac{\epsilon^2 - 2 \cdot d(t) \cdot \epsilon}{(d(t) - \epsilon)^2}, \tag{2.3}$$

where $d(t)$ is the nearest distance between t and another tuple in D, and ϵ is the aforementioned maximum edge error.

Estimation bias for other aggregates can be derived accordingly (given the distribution of the attribute being aggregated). One can make two observations from the theorem: First, the smaller maximum edge error ϵ is or the large inter-tuple distance $d(t)$ is, the smaller the bias will be. Second, we can make the bias arbitrarily small by shrinking ϵ - which leads to a log-scale increase of the query cost.

Algorithm 6 shows the pseudocode for LNR-LBS-AGG that also utilizes some of the error reduction ideas from §2.3.2.

Example 6: We consider the same dataset as Example 1, except that in LNR-LBS the locations of tuples are not returned. Figure 2.8 shows a run-through of the algorithm by which one of the Voronoi edges of $V(t_4)$ is identified. Initially, the bounding box contains the entire region, i.e. V_0. ℓ_1 and ℓ_2 are two lines starting from t_4 constructed as per Algorithm 7. p_1 and p_2 are mid points of small line segments on ℓ_1 and ℓ_2 such that points on either side of them return different tuples when queried. The new estimated Voronoi

Algorithm 6 **LNR-LBS-AGG**

1: **while** query budget is not exhausted
2: q = location chosen uniformly at random; t=query(q)
3: Construct four points $q_1, \ldots, q_4$ bounding t
4: e_i = Binary-Search(q_i) $\forall i \in [1, 4]$
5: $V(t)$ = closed polygon from Voronoi edges $e_1, \ldots, e_4$
6: D'= vertices of $V(t)$
7: **repeat** till D' is not updated
8: **for** each vertex v of $V(t)$: $D' = D' \cup$ query(v)
9: Find Voronoi edges $\forall d \in D'$ and update $V(t)$
10: Produce aggregate estimation using samples

edge is computed as the line segment connecting p_1 and p_2. Please refer to §-2.9 for further details.

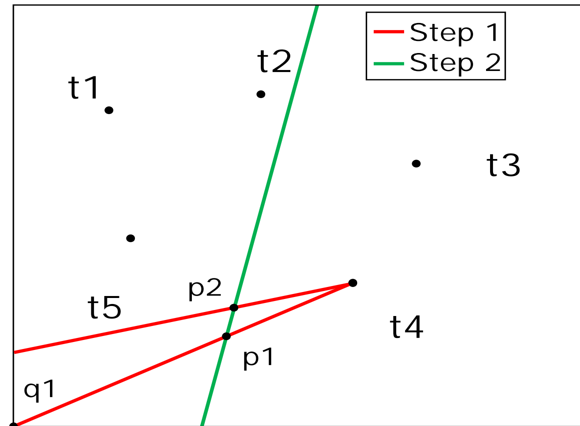


Figure 2.8: Illustration of Algorithm LNR-LBS-AGG

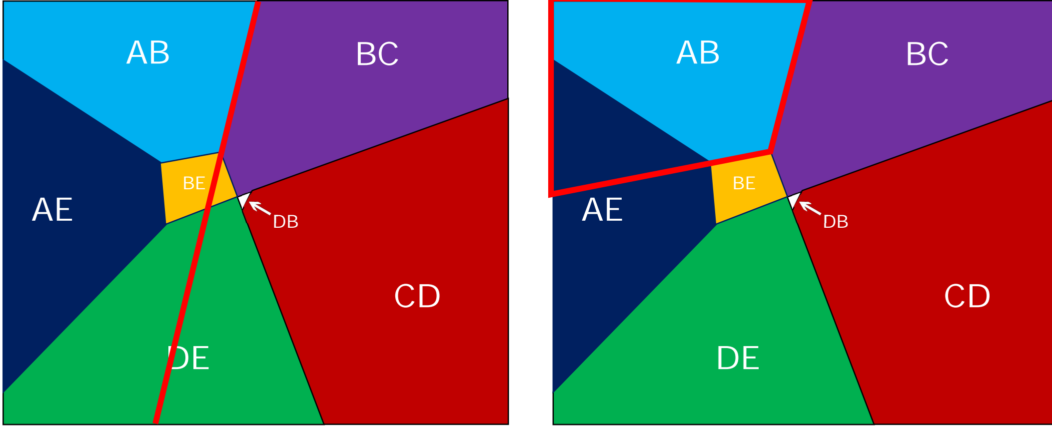


Figure 2.9: Handling Concavity of top-k Voronoi Diagrams

2.4.2 Extension to $k > 1$

A complication brought by the rank-only return semantics is the extension to cases with $k > 1$. Specifically, recall from §2.2 that the (extended) top-k Voronoi cell might be *concave* when $k > 1$. In the case LR-LBS case, this does not cause any problem because, at any moment, our derived top-k Voronoi cell is computed from the exact tuple locations of all observed tuples and (therefore) completely covers the real top-k Voronoi cell. For LNR-LBS case, however, this is no longer the case: Since we unveil the top-k Voronoi cell edge after edge, if we happen to come across one of the "concave edges" early, then we may settle on a sub-region of the real top-k Voronoi cell. Figure 2.9 demonstrates an example for such a scenario.

Fortunately, there is an efficient fix to this situation. To understand the fix, a key observation is that any "inward" (i.e., concave) vertex of a top-k Voronoi cell, say that of t, must be at a position with equal distance to three tuples, one of them being t (Note: this might not hold for "outward" vertices). This property is proved in the following lemma.

Lemma 1. *Any inward vertex of the top-k Voronoi cell of t must be of equal distance to t and two other tuples in the database.*

Proof. Consider a partition of the entire region into *base cells*, each of which returns a different combination of top-k tuples. One can see that the top-k Voronoi cell of t must be the union of one or more adjacent base cells. In addition, for general positioning (i.e., barring special positions such as bounding edges, etc.), any vertex of the top-k Voronoi cell is formed by three edges (of some base cells in the partition). Now consider the three edges which form an inward vertex v, denoted by e_1, e_2, e_3. Note that, given v is inward, one of the three edges must be inside the top-k Voronoi cell of t. Let this edge be e_1. One can see that both e_2 and e_3 separate the top-k Voronoi cell from the outside - i.e., $\forall i \in \{1, 2\}$, we have locations on one side of e_i returning t in top-k while locations on the other side do not. That is, each of e_2 and e_3 must be the perpendicular bisector of the line segment connecting t and another tuple in the database. Let these two tuples be t_2 and t_3 for e_2 and e_3, respectively. In other words, v must have equal distance to t, t_2 and t_3. □

Given this property, the extension to $k > 1$ becomes straightforward: Let D' be the set of all tuples we have observed which appear along with t in the top-k result of a query answer. Let $t \in D'$. First, note that if the polygon we output is not the top-k Voronoi cell of t, then it must be a sub-region of it missing at least one inward vertex. According to the above lemma, each inward vertex is formed by two perpendicular bisectors, each between t and another tuple. A key observation here is that at least one of the *missed* inward vertices must be entirely formed by tuples in D'. The reason is simple: if no missed inward vertex satisfies this property, then we must have found the correct top-k Voronoi cell of t over D' - i.e., what we get so far must be a super-region of the correct top-k Voronoi of t over the entire database, contradicting our previous conclusion that it is a sub-region.

Now our task is reduced to finding such a missing inward vertex. Note that this is equivalent with finding the perpendicular bisector of t and every other tuple in D' - as once these perpendicular bisectors are identified, the rest is simply getting their intersections

which can be done offline. For each tuple in D', we either have already identified the perpendicular bisector through one of the previous calls to the binary search process - or we can initiate a new one as follows.

Specifically, to find the perpendicular bisector of t and $t' \in D'$, note that t' being in D' means that (1) at least one of the vertices of the polygon we currently have must return t', and (2) at least one of the vertices of the polygon we currently have must not return t'. In other words, there must exist an edge of our current polygon which has two vertices once returning t' and the other does not - i.e., this edge intercepts with the perpendicular bisector of t and t'. As such, we simply need to return the binary search process over this edge to find the perpendicular bisector, and then use it to update our polygon. We repeat this process iteratively until we have enumerated all perpendicular bisectors of t and other nodes in D' - at which time we can conclude that there is no missing inward vertex. In other words, we have found the top-k Voronoi cell of t. One can see that the query complexity of this process remains at $O(m \log(1/\epsilon))$, as every new binary search process called will return us a new edge for the top-k Voronoi cell.

2.4.3 Tuple Position Computation

Another important problem in the LNR-LBS case is the computation of a tuple's position, since such information is not returned in query answers as in the LR-LBS case. As discussed in the introduction, this problem can be of independent interest - it can also be called upon as a subroutine for aggregate query processing when the selection condition involves a tuple's location. For example, one might be interested in the number of WeChat users within 20 meters of major highways (i.e., those who are likely driving). To estimate this aggregate, we need to compute the location of a tuple (i.e., a WeChat user) in order to determine whether it satisfies the selection condition for the aggregate query.

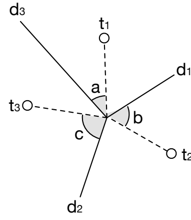


Figure 2.10: Demonstration of Tuple Position Computation

Once we compute the Voronoi cell for a tuple t, the computation of t's exact location takes only two additional calls to the binary search process. The key idea of this computation is demonstrated in Figure 2.10. The figure depicts one vertex of the top-1 Voronoi cell of t_1. Let the vertex (at the center of the figure) be the origin point o. The figure includes two edges of the Voronoi cell, d_1 and d_3, corresponding to the perpendicular bisector of (t_1, t_2) and (t_1, t_3), respectively. Note that since o is of equal distance to t_1, t_2, and t_3, it must be attached to a third edge which is part of the Voronoi cell for t_2 and t_3 - this is depicted as d_2 in the figure.

In the following, we describe the computation of t_1's location in three steps: First, we show that, with knowledge of d_1, d_2 and d_3, one can readily compute the line from o to t_1 - i.e., the angle a in the figure. Note that this indicates as long as one can do the same for another vertex of the Voronoi cell (say o'), then the location of t_1 can be derived as the intersection of two lines: (o, t_1) and (o', t_1). Of course, in practice we only know d_1 and d_3 from the Voronoi cell computation, not d_2. Thus, we demonstrate in the second step that deriving d_2 from d_1 and d_3 takes only a single call to the binary search process.

First, to understand how angle a can be derived from d_1, d_2, d_3, a key observation from Figure 2.10 is that the lines from o to any two tuples must form equal angle to the Voronoi edge between them - e.g., (o, t_1) and (o, t_3) must form equal angles to d_3. In

other words, the angle between (o, t_3) and d_3 is also a. Equipped with this observation, it becomes obvious that:

$$(a + b) + (b + c) + (c + a) = 2\pi \tag{2.4}$$

$$\Rightarrow a + b + c = \pi \tag{2.5}$$

Since $b + c$ is exactly the angle between d_1 and d_2, we can easily compute a as $\pi - (b + c)$. As such, we computed the line from o to t_1 based on knowledge of only d_1, d_2 and d_3.

Now we explain how one can compute d_2 - the only one of the three edges not part of the Voronoi cell of t_1 - with a single call to the binary search process. Note from the fact that we have computed both d_1 and d_3 that we must have issues at least one query which returns t_2 as the top result, say q_2, and a query which returns t_3 on the top, say q_3. Obviously, d_2 intercepts the line segment between q_2 and q_3 exactly once. Thus, we simply need to call the binary search process over (q_2, q_3) to derive d_2 and enable the computation of t_1's exact location. One can see that, overall, the query complexity for computing both the Voronoi cell and the location of a tuple remains $O(m \log(1/\epsilon))$, where m is the number of edges for its Voronoi cell.

2.5 Discussions

2.5.1 Aggregates with Selection Conditions

In most of the previous discussions, we considered aggregates without selection conditions (i.e., every tuple in the bounding region is aggregated). There is indeed a straightforward extension to aggregates with selection conditions - specifically, there are two possible scenarios:

The first is when the selection condition can be "passed through" to LBS. For example, if our goal is to COUNT "STARBUCKS" within the bounding region, the selection condition NAME = 'STARBUCKS' can be passed through to LBS - i.e., we simply ap-

pend to each query we issue the exact same selection condition as the aggregate, NAME = 'STARBUCKS'. One can see that no other change is required to the aggregate estimation process.

The other scenario is when the LBS does *not* support the selection condition. For example, if we want to COUNT all businesses with at least an average review score of four stars within the bounding region, then we cannot simply pass this selection condition to an LBS that does not support filtering by average review scores. In this case, we simply need to "post-process" the selection condition - e.g., for the above example, this means that after randomly choosing a query and obtain the returned tuple (as in §2.3.1), we first determine if the tuple satisfies the filtering condition. If so, we continue with the original process and return the same estimation. Otherwise, we return 0 (i.e., the aggregate query applied over the returned tuple, again divided by the sampling probability) as the estimation. One can see that the result remains an unbiased estimation for the aggregate, now with selection conditions.

In the experiments, we shall demonstrate online tests over real-world LBS on aggregates with selection conditions in both categories - e.g., COUNT of STARBUCKS over Google Maps, which can be passed through, and COUNT(restaurants) that are open on Sundays, which cannot.

2.5.2 Leveraging External Knowledge

In previous discussions, we focused on how to process the results returned by a randomly chosen query (e.g., how to compute the top-k Voronoi cell of a returned tuple). The way the initial query is chosen, however, remains a simple design of choosing a location uniformly at random from the bounding region. Admittedly, without any knowledge of the distribution of tuple locations, uniform distribution appears the natural choice. Nonethe-

less, in real-world applications, we often have certain *a priori* knowledge of the tuple distributions, which we can leverage to optimize the sampling distribution of queries.

For example, if our goal is to estimate an aggregate, say COUNT, of Point-Of-Interests (POIs, e.g., restaurants) in the US, a reasonable assumption is that the density of POIs in a region tends to be positively correlated with the region's population density. Thus, we have two choices: either to sample a location uniformly at random - which leads to POIs in rural areas to be returned with a much higher probability (because their Voronoi cells tend to be larger); or to sample a location with probability proportional to its population density - which hopefully leads to a more-or-less uniform selection probabilities over all POIs. Clearly, the second strategy is likely better for COUNT estimation, as a more uniform selection probability distribution directly leads to a smaller estimation variance (and therefore error). For example, in the extreme-case scenario where all POIs are selected with equal probability, our COUNT estimation will be precise with zero error. Thus, an optimization technique we adopt in this case is to design the initial sampling distribution of queries according to the population density information retrieved from external sources, e.g., US Census data [17].

There are two important notes regarding this optimization: First, no matter if the external knowledge is accurate or not, the COUNT and SUM estimations we produce always remain unbiased. This is obvious from (2.1) in §2.3.1, which guarantees unbiasedness no matter what the sampling distribution $p(t)$ is. Second, the optimal sampling distribution depends on both the tuple distribution and the aggregate query itself. For example, if we want to estimate the SUM of review counts for all POIs, then the optimal sampling distribution is to sample each tuple with probability proportional to its review count (as this design produces zero estimation variance and error). Given the difficulty of predicting the aggregate (e.g., review COUNT in this case) ahead of time, leveraging external knowledge is better considered as heuristics (a very effective one nonetheless, as we shall demonstrate

in experimental results) rather than a practice that guarantees the reduction of estimation errors.

2.5.3 Special LBS Constraints

We now consider two special constraints that are enforced by the query interfaces of some real-world LBS. The first one is a *maximum radius* on the returned results - i.e., the distance between the query location q and the returned tuples is bounded by a pre-determined threshold $d_{\max}$. If no tuple in the database falls within the circle centered at q with radius $d_{\max}$, then the query returns empty. Google Maps and Weibo both enforce this constraint, with the threshold being 50 KM [18] and 11 KM[4], respectively.

Interestingly, no change is required for our algorithms (both LR-LBS-AGG and LNR-LBS-AGG) to handle this situation. One can see that, as long as a query result is non-empty, the nearest neighbor is always returned, enabling the usage of our algorithms. When a query returns empty, we simply return 0 as the COUNT or SUM estimation (for this sample query). The unbiasedness is unaffected - note from (2.1) in §2.3.1 that unbiasedness is guaranteed no matter if the sampling probability $p(t)$ of all tuples sum up to 1 or not - as long as each tuple still has a positive probability to be returned. With this constraint, there is $\sum_t p(t) < 1$ with the remaining probability returning 0 - still leading to an unbiased SUM or COUNT estimation.

The second constraint we have observed from real-world LBS is a more complex ranking function that involves not only the distance between query location and a tuple but also other factors such as the static rank of certain attributes for the tuple. Google Places API is an example here, as it allows ranking by "prominence" which takes into account both distance and tuple popularity[5].

[4] `http://open.weibo.com/wiki/2/place/nearby/users`

[5] Note that Google Places API also supports traditional distance-based ranking, enabling the direct usage of our algorithms.

For this constraint, the applicability of our results is no longer straightforward. The key challenge here is that the area returning a tuple may become segregated across many disjoint regions, making it extremely difficult to compute the sampling probability ($p(t)$ in (2.1) in §2.3.1) for a tuple. To understand why, consider an example where tuples are ranked according to the SUM of two scores, one is distance, awarding a higher score to a tuple closer by, but 0 to tuples more than 50 miles away. The other is a static score such as popularity. What might happen here is that the most popular tuple (in the bounding region, say US) is returned by queries on all places without a tuple within 50 miles (say the middle of a desert in Nevada). Clearly, it becomes extremely difficult to enumerate all the disjoint regions that return this tuple.

Fortunately, for LR-LBS in practice, it is still highly likely for our LR-LBS-AGG algorithm to successfully handle the constraint - because the algorithm works properly as long as the nearest neighbor is always included in the top-k results. Since an LR-LBS returns tuple locations, we can always post-process the query answer to obtain the nearest neighbor according to distance, and then apply our algorithm. Given that $k \gg 1$ in real-world LBS, we anticipate a near-certain probability for the nearest neighbor to be included in the top-k results, thus enabling LR-LBS-AGG.

2.5.4 Extension to Higher Dimensions

While LBS in practice is mostly confined to 2D, we would like to point out here (if only for theoretical interests) that our algorithm readily applies to kNN queries over higher-dimensional data where Euclidean distance is used as the ranking function. Specifically, note that for LR-LBS, Theorem 1 holds no matter what dimensionality the tuple locations have - as a higher-dimensional Voronoi cell computed from a subset of tuples still completely encompasses the real one. Similarly, all the optimizations discussed in §2.3.2 readily apply as well. For LNR-LBS-AGG, the only change required is on the binary search

process: instead of finding the perpendicular bisecting *line* between two tuples as in the 2D case, we now need to find the perpendicular bisecting $(d-1)$-dimensional plane in the d-dimensional case. Correspondingly, each vertex of the d-D Voronoi cell is now the interception of $\binom{d}{2}$ such $(d-1)$-dimensional planes. In other words, we still only need two vertices of the Voronoi cell to derive a tuple's location in LNR-LBS - enabling the usage of LNR-LBS-AGG.

2.6 Experimental Results

2.6.1 Experimental Setup

Hardware and Platform: All our experiments were performed on a quad-core 2.5 GHz Intel i7 machine running Ubuntu 14.10 with 16 GB of RAM. The algorithms were implemented in Python.

Offline Real-World Dataset: To verify the correctness of our results, we started by testing our algorithms locally over OpenStreetMap [19], a real-world spatial database consisting of POIs (including restaurants, schools, colleges, banks, etc.) from public-domain data sources and user-created data.

We focused on the USA portion of OpenStreetMap. To enrich the SUM/COUNT/AVG aggregates for testing, we grew the attributes of POIs (specifically, restaurants and schools) by "JOINing" OpenStreetMap with two external data sources, Google Maps [18] and US Census [17]. Specifically, we added for each (applicable) restaurant POI its review ratings from Google Maps; and each school POI its enrollment number from US Census. The US Census data is also used as the (optional) external knowledge source - i.e., to provide the population density data for the optimization technique discussed in §2.5.

Note that we have complete access to the enriched dataset and full control over its query interface. Thus, we implemented a kNN interface with ranking function being the Euclidean distance; returned attributes either containing all attributes including location (for testing LR-LBS) or without location (for LNR-LBS); and varying k to observe the change of performance for our algorithms.

Online LBS Demonstrations: In order to showcase the efficacy of our algorithms in real-world applications, we also conducted experiments *online* over 3 very popular real-world LBS: Google Maps [18], WeChat [20], and Sina Weibo [21]. Each of these services has at least hundreds of millions of users. Unlike the offline experiments, we do not have direct access to the ground-truth aggregates due to the lack of partnership with these LBS. Nonetheless, we did attempt to verify the accuracy of our aggregate estimations with information provided by external sources (e.g., news reports) - more details later in the section.

In online experiments for LR-LBS, we used Google Maps, specifically its Google Places API [18], which takes as input a query location (latitude/longitude pair) and (optionally) filtering conditions such as keywords (e.g., “Starbucks”) or POI type (e.g., “restaurant”), and returns at most $k = 60$ POIs nearby, ordered by distance from low to high, with location and other relevant information (e.g., review ratings) returned for each POI.

For LNR-LBS, we tested WeChat and Sina Weibo using their respective Android apps. Both directly fetch locations from the OS positioning service and search for nearby users, with WeChat returning at most $k = 50$ and Sina Weibo returning $k = 100$ nearest users. Unlike Google Maps, these two services do *not* return the exact locations of these nearby users - but only provide attributes such as name, gender, etc.

An implementation-related issue regarding WeChat is that, unlike its mobile apps, its API does not support nearest-neighbor search. Thus, we conducted our experiments by running its Android app (with support for nearest-neighbor search) on the official Android

emulator, and used the debugging feature of location spoofing to issue queries from different locations. We then used the MonkeyRunner tool[6] for Android emulator to interact with the app - i.e., sending queries and receiving results. Specifically, to extract query answers from the Android emulator, we first took a screenshot of the query-answer screen, and then parsed the results through an OCR engine, tesseract-ocr[7].

Algorithms Evaluated: We mainly evaluated three algorithms in our experiments: LR-LBS-AGG and LNR-LBS-AGG from §2.3 and §2.4, respectively, along with the only existing solution for LR-LBS (note there is no existing solution for LNR-LBS), which we refer to as LR-LBS-NNO [13]. LR-LBS-NNO has a number of tuneable parameters - we picked the parameter settings and optimizations from [13] that provided the best performance. We also tested variants of our algorithms that lack certain variance-reduction techniques discussed in §2.3 and the weighted sampling in order to demonstrate the effectiveness of these techniques.

Performance Measures: As discussed in §2.2, we measure efficiency through query cost, i.e., the number of queries issued to the LBS. Our estimation accuracy is measured experimentally by relative error. Each data point is obtained as the average of 25 runs.

2.6.2 Experiments over Real-World Datasets

Unbiasedness of Estimators: Our first experiment seeks to show the unbiasedness of our estimators for LR-LBS-AGG and LNR-LBS-AGG even after incorporating the various error reduction strategies. LR-LBS-NNO is known to be unbiased from [13] after an expensive bias correction step. Figure 2.12 shows a trace of the three algorithms when estimating COUNT of all restaurants in US by plotting the current estimate periodically after fixed number of queries have been issued to LBS. We can see that LR-LBS-NNO has a high

[6] http://developer.android.com/tools/help/monkeyrunner_concepts.html

[7] https://code.google.com/p/tesseract-ocr/

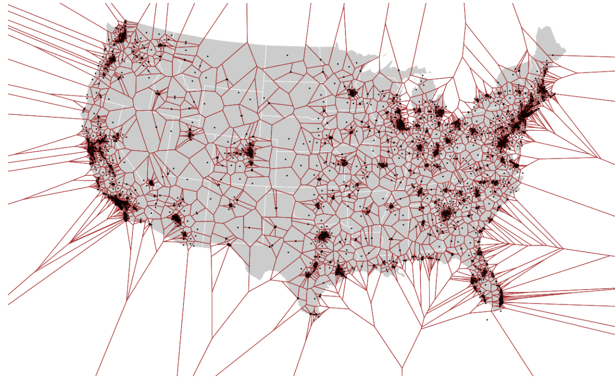


Figure 2.11: Voronoi Decomposition of Starbucks in US

variance and takes significantly longer to converge while our estimators quickly converge to the ground truth much before LR-LBS-NNO. This indicates that the error reduction techniques successfully reduce the variance of our estimators.

Query Cost versus Relative Error: We start by testing the key tradeoff - i.e., query cost vs. relative error - for all three algorithm over various aggregates. Specifically, Figures 2.14, 2.15, 2.16 and 2.17 show the results for four queries, COUNT of schools in US, COUNT of restaurants in US, SUM of school enrollments in US, and AVG of restaurant ratings in Austin, Texas, respectively. One can see that not only our LR-LBS-AGG algorithm significantly outperform the previous LR-LBS-NNO [13] in all cases, even our algorithm for the LNR-LBS case achieves much better performance than the previous algorithm (despite the lack of tuple locations in query results).

Query Cost versus LBS Size: Figure 2.18 shows the impact of LBS database size (in terms of number of POIs or users) on query cost to estimate the COUNT of schools in US for a fixed relative error of 0.1 . We varied the database size by picking a subset of the database (such as 25%, 50%, etc) uniformly at random and estimating the aggregate over it. As expected for a sampling-based approach, the increase in database size do not have any major impact and only results in a slight increase in overall query cost (due to the more complex topology of Voronoi cells).

Query Cost versus k: Figure 2.19 shows how the value of k (the number of tuples returned by k-NN interface) affects the query cost. Again, we measure the query cost required to achieve a relative error of 0.1 on the aggregate COUNT of schools in US. We compared an variant that leverages our variance reduction strategy that adaptively decides which subset of tuples (i.e. h of top-k) to use with fixed variants that uses all the top-k tuples. As expected, our adaptive strategy has a lower query cost and consistently achieves a saving of 10% of query cost.

Efficacy of Error Reduction Strategies: We started by verifying the effectiveness of weighted sampling using external knowledge - Figure 2.13 compares the performance of the two sampling strategies - uniform and weighted - while estimating the COUNT of schools in US. One can see that the weighted sampling variants result in significant savings in query cost.

In our final set of experiments, we evaluated the efficacy of the various error reduction strategies we described in the paper. We compared 5 different variants of our algorithm for LR-LBS ranging from no error reduction strategies (LR-LBS-AGG-0) to sequentially adding them one by one in the order discussed in the paper culminating in LR-LBS-AGG that incorporates all of them. Figure 2.20 shows the results of this experiment. As expected the first two strategies of faster initialization and leveraging history caused a significant reduction in query cost. We observed that the results for LNR-LBS were very similar.

2.6.3 Online Demonstrations

Google Places: Our first online demonstration of LR-LBS-AGG was on Google Places API and estimating two aggregates with different selection conditions. The first involves selection conditions that can be passed over to LBS (COUNT of Starbucks in US) while the

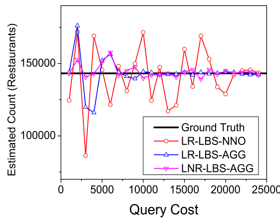


Figure 2.12: Unbiasedness of Estimators

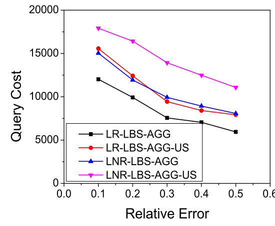


Figure 2.13: Impact of Sampling Strategy

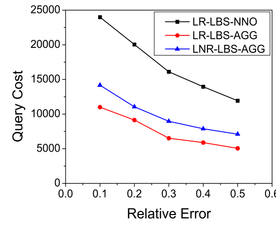


Figure 2.14: COUNT(schools)

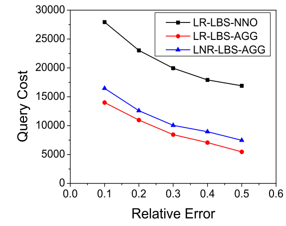


Figure 2.15: COUNT(restaurants)

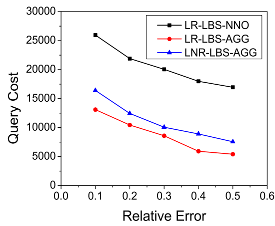


Figure 2.16: SUM(enrollment) in Schools

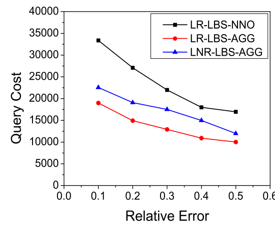


Figure 2.17: AVG(ratings) in Austin, TX Restaurants

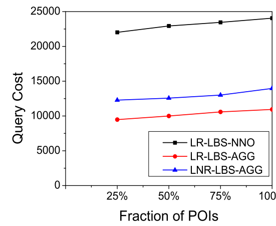


Figure 2.18: Varying Database Size

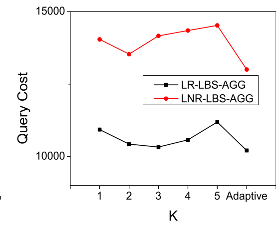


Figure 2.19: Varying k

second involves aggregates with selection condition that cannot be passed over (see §2.5 for discussion) such as COUNT of restaurants in Austin, Texas that are open on Sundays.

Table 2.1 shows the results of the experiments. We also verified the accuracy of our estimates for first aggregate (COUNT of Starbucks) through the public release of Starbucks Corp [22]. One can see from the table that, with just 5000 queries, LR-LBS-AGG achieves very accurate estimations ($< 5\%$ relative error) for the count.

To provide an intuitive illustration of the execution of our algorithm, we also continued the estimation of COUNT("Starbucks") until enumerating all Starbucks in the US. Figure 2.11 demonstrates the Voronoi diagram constructed by our algorithm at the end. One can see the vastly different sizes of Voronoi cells - spanning hundreds of thousands km^2 in rural areas and smaller than 1km^2 in urban cities, justifying the effectiveness of weighted sampling.

Table 2.1: Summary of Online Experiments

LBS	Aggregate	Estimate	Query Budget
Google Places	COUNT(Starbucks in US)	12023	5000
Google Places	COUNT(restaurants in Austin TX and open on Sundays)	2856	5000
WeChat	COUNT(WeChat users in China)	338.4 M	10000
WeChat	Gender Ratio of WeChat users in China	67.1:32.9	10000
Weibo	COUNT(Weibo users in China)	44.6 M	10000
Weibo	Gender Ratio of Weibo users in China	50.4:49.6	10000

WeChat and Sina Weibo: We estimated two aggregates, (1) total number of users and (2) gender ratio, over two LNR-LBS, WeChat and Sina Weibo, respectively. Table 2.1 shows the results of the experiments. One can observe from the table that our estimations quickly converge to a narrow range (+/- 5%) after issuing a small number of queries (10000). While we do not have access to the ground truth this time, we do note an interesting observation from our results: the percentage of male users is much higher on WeChat than on Sina Weibo - an observation verified by various surveys in China [23]. We would like to note that the COUNT aggregate measures the number of users who have enabled the location feature of WeChat and Weibo respectively and is different from the number of registered or active accounts.

Localization Accuracy: As a final set of experiments, we also evaluated the effectiveness of our Tuple position computation approaches in tracking real world users. Specifically, we sought to identify the precise location of *static* objects located across the region. We conducted this experiment over Google Places in US and WeChat in China. We treated Google

Places as LNR-LBS by ignoring the location provided its API. We sought to identify the location of 200 randomly chosen POIs after issuing at most 100 queries for each POI. For WeChat, we positioned our user at 200 diverse locations within China (typically in Urban places) and sought to identify the location. Since the precise location of the POI/user is known, we can compute the distance between actual and estimated positions. Figure 2.21 shows the result of the experiments. The results show that more than 80% of the POIs were located within 20m of the exact location and every POI was located within a distance of 75m. Due to the various location obfuscation strategies employed by WeChat, we achieved an accuracy of 50m or lower only 45% of the time. We still were able to locate user within 100m almost all the time. While our theoretical methods could precisely identify the location, the discrepancy in real-world occurs due to various external factors such as obfuscation, coverage/localization limits etc.

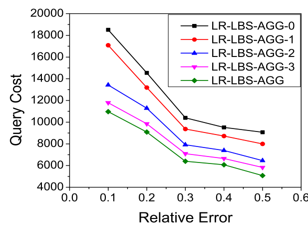


Figure 2.20: Query Savings of Error Reduction Strategies

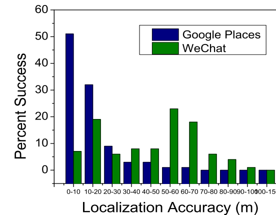


Figure 2.21: Localization Accuracy

2.7 Related Work

Analytics and Inference over LBS: Location based Services (LBS) such as map services (Google Maps) and location based social networks (such as FourSquare, WeChat, Sina Weibo) are becoming popular in recent years. The prior work on analytics over LBS focussed exclusively on the LR-LBS scenario. The closest prior work is [13] that seeks to

estimate COUNT and SUM aggregates over LR-LBS using a nearest neighbor oracle. It then corrects the bias by using the area of the Voronoi cell using an approach that is very expensive. Aggregate estimation over LBS such as FourSquare that does not provide nearest neighbor oracle interface could be done using [24, 25]. [25] proposed a random region sampling method with an unknown estimation bias that could be eliminated using techniques from [24]. However, none of them work for LNR-LBS. There has been work on inferring the location and other private information of users of LBS. [15] proposed trilateration based methods to infer the location of users even when the LBS only provided relative distances. There has been other extensive work [26–29] on inferring location information and re-identification of users although none of them are applicable for the LBS models studied in this paper.

Aggregate Estimations over Hidden Web Repositories: There has been a number of prior work in performing aggregate estimation over static hidden databases. [30] provided an unbiased estimator for COUNT and SUM aggregates for *static* databases with form based interfaces. [31–34] describe efficient techniques to obtain random samples from hidden web databases that can then be utilized to perform aggregate estimation. Recent works such as [35, 36] propose more sophisticated sampling techniques so as to reduce the variance of the aggregate estimation. For hidden databases with keyword interfaces, prior work have studied estimating the size of search engines [37–39] or a corpus [40, 41].

2.8 Final Remarks

In this paper, we explore the problem of aggregate estimation over location based services that are increasingly popular. We introduced a taxonomy of LBS with k-NN query interface based on whether location of the tuple is returned (LR-LBS) or not (LNR-LBS). For the former, we proposed an efficient algorithm and various error reduction strategies

that outperforms prior work. We initiate study into the latter by proposing effective algorithms for aggregation and inferring the position of tuple to arbitrary precision which might be of independent interest. We verified the effectiveness of our algorithms by using a comprehensive set of experiments on a large real-world geographic dataset and online demonstrations on high-profile real-world websites.

2.9 Binary Search Process

Design of Binary Search: Given the half-line ℓ from c_1 passing through c_2, we conduct the binary search as follows. First, we find c_{b}, the intersection of this half-line with the bounding box. Then, we perform a binary search between c_1 and c_{b} to find a segment of the half-line with length at most δ, say with two ends being c_3, c_4 (with the distance between c_3 and c_4 at most δ), such that while c_3 returns t, c_4 returns another tuple, say t'. This step takes at most $\log(b/\delta)$ queries, where b is the perimeter of the bounding box.

Then, we consider two half-lines ℓ_1 and ℓ_2, both of which start from c_1 and form an angle of $-\arcsin(\delta'/r)$ and $+\arcsin(\delta'/r)$ with ℓ, respectively, where δ' is a predetermined (small) threshold and r is the distance between c_1 and c_4. For each ℓ_i, we perform the above binary search process to find a (at most) δ-long segment that returns t on one end and t' on the other. Note that such a process might fail - e.g., there might no point on ℓ_i which returns t'. We set two rules to address this situation: First, we terminate the search for ℓ_i if we have reached a segment shorter than δ, with one end returning t and the other returning a tuple other than t'. Second, we move on to the next step as long as (at least) one of ℓ_1 and ℓ_2 gives us a satisfactory δ-long segment. If neither can produce the segment, we terminate the entire process and output the following (estimated) Voronoi edge: the perpendicular bisector of (c_3, c_4).

Now suppose that ℓ_1 produces a satisfactory segment of at most δ long. Let this segment be (c_5, c_6). We simply return our (estimated) Voronoi edge as the line that passes through: (1) the midpoint of (c_3, c_4), and (2) the midpoint of (c_5, c_6). One can see that the overall query cost of the binary search process is at most $3\log(b/\delta)$.

Algorithm 7 provides the pseudocode for Binary Search process.

Algorithm 7 Binary-Search

1: **Input:** Tuple t, Locations c_1, c_2 where query(c_1) returns t
2: **Output:** An edge of $V(t)$
3: c_b = Intersection of half-line c_1, c_2 with bounding box
4: Find c_3, c_4 s.t. $dist(c_3, c_4) < \delta$ and $query(c_3) \neq query(c_4)$
5: $r = dist(c_1, c_4)$
6: Construct lines ℓ_1, ℓ_2 from c_1 with angles $\pm \arcsin(\delta'/r)$
7: (c_5, c_6) = line segment on ℓ_1 or ℓ_2 with $dist(c_5, c_6) < \delta$ and query(c_5) $\neq$ query(c_6)
8: **if** none exists, return perpendicular bisector of (c_3, c_4)
9: **else** return line segment passing through midpoints of (c_3, c_4) and (c_5, c_6)

Error Bound on Edge Estimation: We have the following theorem on the error bound of this binary search process:

Theorem 3. *For a given tuple t and query location c_1 which returns t, for any other location c_2, the Voronoi cell of t must have an edge ℓ_{V} that intercepts half-line (c_1, c_2) such that the maximum edge error for estimating ℓ_{V} satisfies*

$$\epsilon \leq \max(2\delta', b \cdot \sin(\arctan(\delta/\delta'))). \tag{2.6}$$

In other words, for every point $p \in \ell_{\rm V}$*, there exists a point* p' *on our estimated Voronoi edge* $\ell'_{\rm V}$ *generated from* (c_1, c_2) *(i.e.,* $p' \in \ell'_{\rm V}$*), such that*

$$d(p, p') \leq \max(2\delta', b \cdot \sin(\arctan(\delta/\delta'))), \tag{2.7}$$

where $d(\cdot, \cdot)$ *is the Euclidean distance between two points. In addition, for every vertex* v *of* $\ell_{\rm V}$*, if line segment* (t, v) *intercepts* $\ell'_{\rm V}$*, then the interception point* v' *must satisfy*

$$d(t, v) - d(t, v') \leq \max(2\delta', b \cdot \sin(\arctan(\delta/\delta'))). \tag{2.8}$$

A simple observation from the theorem is that the binary search process can reach an arbitrary precision level - i.e., for any given upper bound on $d(p, p')$, say $d_{\rm U}$, we can set $\delta' = d_{\rm U}/2$ and

$$\delta \leq \tan\left(\arcsin\left(\frac{d_{\rm U}}{b}\right)\right) \cdot \frac{d_{\rm U}}{2} \tag{2.9}$$

to satisfy the bound. Since both $\tan$ and $\arcsin$ can be bounded from both sides by a polynomial of its input (through Taylor expansion), one can see that the corresponding query complexity is $O(\log(b/d_{\rm U}))$, leading to the following corollary on the maximum error edge defined in §2.3.

Corollary 1. *The query cost required for achieving a maximum edge error of* ϵ *is* $O(\log(b/\epsilon))$ *- i.e.,* $O(\log(1/\epsilon))$ *when we consider the bounding box size* b *to be constant.*

Error Bound on Voronoi Cell Volume Estimation: A direct corollary from Theorem 3 is an error bound on the estimated volume of a Voronoi cell. Note from our design of LNR-LBS-AGG that our estimated Voronoi cell is always a subregion of the real one. This, in combination with (2.8) in Theorem 3, leads to the following corollary.

Corollary 2. *For a given tuple* t*, the ratio between the volume of the estimated Voronoi cell* $V'(t)$ *and the real one* $V(t)$ *satisfies*

$$\left(\frac{d - \epsilon}{d}\right)^2 \leq \frac{|V'(t)|}{|V(t)|} \leq 1 \tag{2.10}$$

where d *is the nearest distance between* t *and another tuple in the database, and* ϵ *is the maximum edge error.*

CHAPTER 3

ANALOC: Efficient ANAlytics over LOCation Based Services

Location Based Services (LBS), including standalone ones such as Google Maps and embedded ones such as "users near me" in the WeChat instant-messaging platform, provide great utility to millions of users. Not only that, they also form an important data source for geospatial and commercial information such as Point-Of-Interest (POI) locations, review ratings, user geo-distributions, etc. Unfortunately, it is not easy to tap into these LBS for tasks such as data analytics and mining, because the only access interface they offer is a limited k-Nearest-Neighbor (kNN) search interface - i.e., for a given input location, return the k nearest tuples in the database, where k is a small constant such as 50 or 100. This limited interface essentially precludes the crawling of an LBS' underlying database, as the small k mandates an extremely large number of queries that no real-world LBS would allow from an IP address or API account.

We demonstrate ANALOC [42], a web based system that enables fast analytics over an LBS by issuing a small number of queries through its restricted kNN interface. ANALOC stands in sharp contrast with existing systems for analyzing geospatial data,

as those systems mostly assume complete access to the underlying data. Specifically, ANALOC supports the *approximate processing* of a wide variety of SUM, COUNT and AVG aggregates over user-specified selection conditions. In the demonstration, we shall not only illustrate the design and accuracy of our underlying aggregate estimation techniques, but also showcase how these estimated aggregates can be used to enable exciting applications such as hotspot detection, infographics, etc. Our demonstration system is designed to query real-world LBS (systems or modules) such as Google Maps, WeChat and Sina Weibo at real time, in order to provide the audience with a practical understanding of the performance of ANALOC.

3.1 Introduction

We propose to demonstrate ANALOC, a prototypical system for enabling the analytics of data underlying real-world location based services (LBS) by using nothing but the k-Nearest-Neighbor (kNN) search interface (often web-based) publicly provided by the LBS. Specifically, a core feature of ANALOC is its ability to quickly and accurately estimate aggregates such as COUNT, SUM and AVG (with user-specified selection conditions) by issuing only a small number of queries through the kNN interface. ANALOC now works with a suite of popular LBS systems and features such as Google Maps, the "Users Near Me" feature of WeChat mobile app (an instant-messaging platform with 600 million monthly active users), and the microblog search feature of Sina Weibo.

Location Based Services (LBS): LBS has become extremely popular in recent years. Millions of users make daily use of mapping services such as Google Maps, Bing Maps, Nokia HERE, etc. Besides these systems, features related to LBS are also integrated into numerous other systems. For example, many online social networks, e.g., Twitter, WeChat, Sina

Weibo and FourSquare, allow a user to search for other users, posts, POIs, etc., according to his/her own location.

Each LBS has a backend database with tuples representing points of interests (POIs) or users, and attributes capturing their geographical coordinates (e.g., latitude and longitude) along with other information such as POI name, review ratings, user posts, etc. What the LBS reveals to the public is a kNN interface which, upon given an arbitrary query point and a user-specified selection condition, returns k tuples in the database that, among those matching the selection condition, are closest (geographically) to the query location. The value of k is often small - e.g., 50 or 100.

The backend databases of LBS contain a gold mine of information for understanding POI quality, user behavior, etc. For example, the database community has produced a number of demo systems in recent years (e.g., [43, 44]) that leverage LBS data for diverse purposes such as analytics, visualization, etc. Unfortunately, almost all these systems assume complete access to the LBS data or, if the LBS data come from an online system, require such data to be downloaded first before being fed as input to the system. This requirement makes the existing work incompatible with most real-world LBS systems that enforce the so-called *query rate limitation*, i.e., they limit the number of requests from an IP address or API account for a certain time period[1]. Since the kNN interface reveals only a small number of tuples (at most k) per query, the query rate limits make it extremely difficult, if not impossible, to download the LBS data required by the existing systems.

Technical Novelty of ANALOC: Standing in sharp contrast with the existing work, ANALOC requires access to nothing but the public kNN interface of an LBS system. Given the interface, ANALOC uses a small number of kNN queries to accurately estimate COUNT, SUM and AVG aggregates with user-specified selection conditions, thereby enabling analytics over the "hidden" LBS data. Note that ANALOC works when such an

[1]Or, in the case of Google Maps API, the LBS starts charging per query once the free usage quota is exhausted.

interface is implemented on a webpage, through API, or even embedded within a mobile app.

A unique technical challenge faced by ANALOC is the *heterogeneity* of information included in kNN query answers returned by different LBS systems. Specifically, some services (e.g., Google maps and Sina Weibo) return the exact locations (i.e., latitude and longitude) of the k tuples. We refer to these services as Location-Returned LBS (LR-LBS). Others, especially online social networks such as WeChat, return a ranked list of k nearest tuples but *suppress* the location of each tuple, perhaps due to privacy concerns. We refer to such services as Location-Not-Returned LBS (LNR-LBS). Unsurprisingly, LNR-LBS imposes additional challenges on analytics because of the missing location information. Thanks to techniques developed in our recent work [45], ANALOC is able to handle both types and indeed make the end users *oblivious* of their differences.

Demo Plan: To demonstrate the range of applications that can be enabled by ANALOC, we organize our demonstration in two parts. The first part focuses on the core feature of ANALOC, i.e., its ability to support the fundamental yet versatile task of aggregate estimation and tracking. Real-world examples we shall demonstrate include the AVG review scores for Marriott vs Hilton hotels in Google Maps, the COUNT of Starbucks vs Peet's Coffee in the US, the gender distribution of WeChat users in various cities of China, etc. We plan to showcase both the outcome of estimations and also an animated, real-time, illustration of the queries issued and how they lead to the fast-converging, accurate, estimations. The second step of the demo focuses on using ANALOC to enable an end-to-end hotspot detection application which uses the aggregate estimations to detect unusual "hotspots" of social network users (e.g., Sina Weibo and WeChat users).

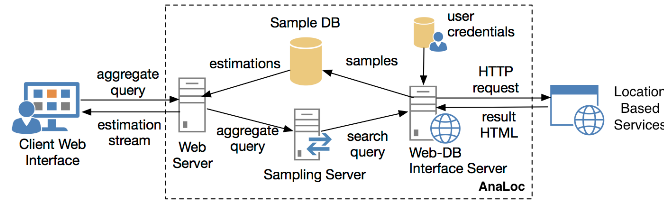


Figure 3.1: Architecture of ANALOC

3.2 System Architecture

We now discuss the architecture of ANALOC as depicted in Figure 3.1. One can see from the figure that the system contains five main components: web server, sampling server, sample database, user credential database, and web database interface server. We would like note that all these components are modular and extensible and could be used as part of a larger system. We describe a sample end-to-end application for hotspot detection in Section 3.3.3.

3.2.1 Web Server

This component provides users with a web interface that allows the specification of aggregate queries over LBS (as input) and visualizes the estimated aggregates (i.e., output) through charts and tables. In addition, the component also supports a RESTful API interface for the same input/output.

Input Interface: The graphical user interface (GUI) of ANALOC allows users to specify an aggregate query through an intuitive, step-by-step, process on a web interface. A user can specify (1) the *aggregate* of interest (SUM, COUNT and AVG are currently supported), (2) *attributes* over which the (SUM and AVG) aggregates will be computed over, (3) *bounding box* that provides the area of interest, (4) *filters* to specify one or more selection conditions on other structured or unstructured attributes featured in the LBS, and (5) *time limit* that limits the maximum overhead of the aggregate estimation process. Note

that this directly translates to a limit on the number of queries to be issued over the LBS. Figure 3.2 depicts an example aggregate query over Google Maps that seeks to compute average ratings of restaurants in Dallas-Forth Worth metroplex within a budget of 1000 queries.

Output Interface: The output interface displays the estimated aggregates in a 2D line chart, and visualizes the change of estimations using an animation. While this is the default output interface, it is possible to produce more sophisticated visualizations such as heatmaps (please see Section 3.3.3 for an example). Figure 3.3 provide a sample output for the aggregate query specified in the input interface. The X-axis corresponds to the query budget and Y-axis shows the current estimate. Note the fast convergence of the estimated aggregate.

3.2.2 Sampling Server

Due to the aforementioned query rate limit of LBS, it is often impossible to precisely compute the aggregate of interest. Hence, we adopt a *sampling-based* mechanism to obtain approximate aggregate estimates. The key task of the sampling server is to translate a user-specified aggregate query into a small number of kNN search queries supported by the LBS, and then submit these kNN queries through the web-DB interface server discussed next.

The sampling server uses a number of novel and efficient techniques for precisely computing or approximately estimating the Voronoi cell of the tuples. We refer the interested reader to [45] for details of our algorithms for aggregate estimation over both LR- and LNR-LBS (referred to as LR-LBS-AGG and LNR-LBS-AGG respectively). Here we only include a brief sketch of these techniques.

Algorithms LR-LBS-AGG and LNR-LBS-AGG: At a high level, the objective of the sampling server is to generate sample tuples in the LBS according to a given sampling

Figure 3.2: Input Interface

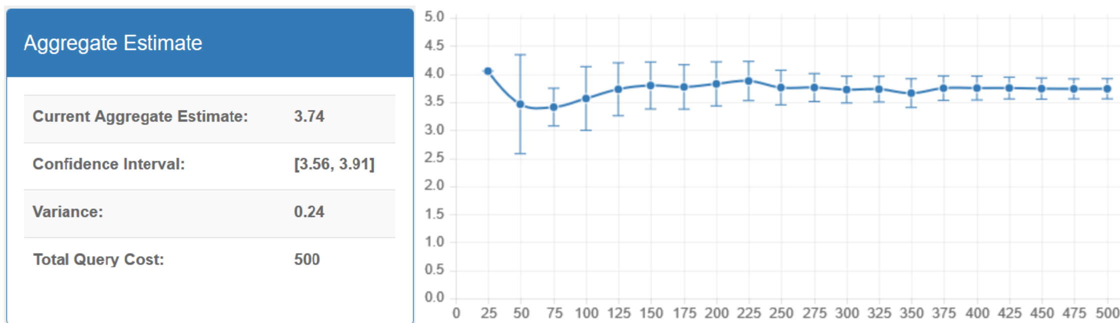


Figure 3.3: Output Interface

distribution. Consider the uniform distribution as an example. A simple idea might be to generate locations uniformly at random, send them as inputs to kNN queries, and retrieve the top ranked tuple as a sample. Unfortunately, this scheme samples some tuples at a higher rate than others. For example, a POI in a rural area is usually returned as the top result for a larger number of query points than a POI in a densely populated area. Hence, to produce sample tuples according to our desired distribution, we have to properly understand the probability for a tuple to be returned in the top ranked (e.g., top-k) results. Once we know the probability, we can apply techniques such as acceptance-rejection sampling or the Hansen-Hurwitz estimator to produce our desired sample and/or aggregate estimation.

In order to estimate the probability for a tuple to be returned, we leverage the concept of Voronoi cells from computational geometry. Given a database D of tuples, each occupying a point on the 2-dimensional space, and a bounding box B, the Voronoi cell of a tuple t is the set of points in B that are closer to t than any other tuple in D. Note a key observation here is that the ratio of area of the Voronoi cell of a tuple t to the area of the bounding box B is exactly the probability for t to be returned as the top-1 result (and sampled). In [45], we also extend the concept of Voronoi cells to capture the probability for t to be returned as one of the top-k results (when $k > 1$).

Given the observation, the key idea behind our algorithms in [45] is to compute the Voronoi cell of each sample t. If the location of the tuple is returned, then it is possible to precisely compute the Voronoi cell based on the observation that we only need the tuples of Voronoi cells adjacent to that of t. Please refer to [45] for further details. However, this approach does not work for LNR-LBS as the location is no longer returned. In this case, we only approximately compute the Voronoi cell of t by using a "binary-search" like algorithm (again, see [45] for details) that when invoked identifies one edge of the Voronoi cell of t to required precision.

3.2.3 Web-DB Interface Server

The task of the web-DB interface server is two-fold: (1) to translate each kNN query issued by the sampling server to a query over the LBS, and (2) to parse the returned results, transform them to structured tuples, and pass them to the sample database component. Each LBS may require a different wrapper design for the kNN input/output translations. For example, a kNN query may be translated to simple HTTP GET or POST requests, RESTful API calls, or simulated interactions in the Android emulator (in the case of a mobile LBS app lacking a web front-end, e.g., WeChat). The raw returned result from LBS are mostly in a structured format such as XML or JSON, which enables simple translations

to our sample database. Some LBS, however, return raw HTML code that has to go through DOM parsing and/or regular expressions before the structured tuples can be extracted.

ANALOC uses a modular architecture where the translation process (from search query to web request to structured tuples) is specified in a separate script file that is utilized by the Web-DB interface server. We demonstrate three typical yet diverse translation scripts in the demo, i.e., RESTful API calls for Google Maps API, HTTP requests for Weibo, and Android-emulator-based simulated interactions for WeChat. Most popular LBS can be supported using one of these approaches.

3.2.4 User Credentials Database

This component allows us to handle the query rate limit enforced by almost all LBS. For example, Sina Weibo allows only 150 queries per hour. Most LBS require logging in with user credentials for API and web access. In order to support multiple concurrent users, ANALOC uses user credential database to store the credentials of end users (such as API keys or username/password) and then issue queries on their behalf. Note that the design of sampling server allows us to "pool" the query quota of all users, allocate queries based on the confidence interval of the estimate so as to avoid overspending trying to improve already accurate estimates.

3.3 Demo Plan

3.3.1 Overview

Hardware Setup and Backup Plan: ANALOC is web based and supports access from multiple platforms. We shall use an iPad connected with a portable projector to demonstrate our system. Meanwhile, we shall also provide 3 laptops with access to ANALOC for visitors who are interested in more interactions. All demo tablets and laptops connect to our web server of ANALOC by default. While the default setting is to access the

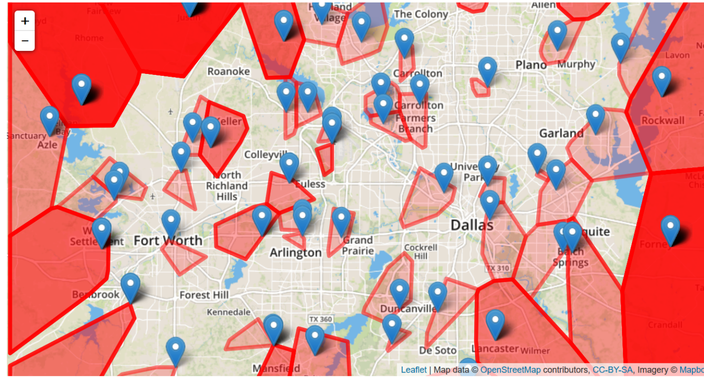


Figure 3.4: Voronoi Tessellation of Samples

target LBS (such as Sina Weibo, WeChat and Google Maps) at real time, we also include in the local deployment of ANALOC a lightweight web server simulating these LBS (using the real historic data we collected).

System Setup and Audience Interactions: While the design of ANALOC is generic to any LBS featuring a kNN search interface, the web-DB interface server component does require a pre-configured specification file for LBS. The demo system shall contain a large number of pre-configured specification files, including popular LBS ranging from online social networks (such as Sina Weibo, WeChat) to map services (such as Google Maps). Visitors to the demo can freely select the LBS of interest, specify the aggregate query of interest, and then observe the aggregate estimation over the selected LBS at real time. To demonstrate during the short demo session the effectiveness and utility of our ANALOC system, we also plan to use the historical data that we collected from various LBS that will then be queried by our Web-DB simulator.

3.3.2 Demo Scenario 1: Aggregate Estimation

Aggregate Estimation over Google Maps (LR-LBS): Google Places API has a kNN interface and returns at most $k = 60$ POIs nearby ordered by location. It also returns other relevant information about POIs such as review ratings, hours of operation etc. We

will demonstrate the estimation of number of aggregates of interest over Google Places. Sample queries will include AVG review scores for Marriott vs Hilton hotels, the COUNT of Starbucks vs Peets Coffee in the US etc. Visitors will be able to visualize how the estimate converges quickly along with the Voronoi cells of the samples visualized over the map. Please refer to Figures 3.3 and 3.4 for an example.

Aggregate Estimation over WeChat (LNR-LBS): WeChat is a popular Chinese social network that provide LBS functionality if the user has enabled it. It is a LNR-LBS as the location is not returned from their public kNN query interface ($k = 50$). We plan to demonstrate the aggregate estimation capability by estimating the gender distribution of WeChat users in various cities of China. In addition, we also will highlight the ability of ANALOC to *infer* the location of a user even though WeChat does not return it.

3.3.3 Demo Scenario 2: Hotspots and Explanations

We now describe a sample application that performs sophisticated analytics over LBS data and was built by leveraging functionality of ANALOC. In this application, we are interested in leveraging the location and timing of tweets from Sina Weibo to identify hotspots over major cities in China. Intuitively, hotspots are locations with unusual activity. Sina Weibo has a kNN query interface with $k = 100$.

By issuing kNN queries over randomly generated latitude/longitude pairs and by using techniques for aggregate estimation from [45] we estimate the count of users. This information is then visualized as a heatmap with darker colors corresponding to regions of high activity. We associate with each hotspot, a weight corresponding to how active it is. The weights of hotspots could also have additional applications such as identifying top-l hotspots with most activity and tracking the activity over a period of time. By performing repeated aggregate estimations over a given hotspot, we could identify trends over them and identify hotspots that are most frequently changing. We also seek to infer the reason

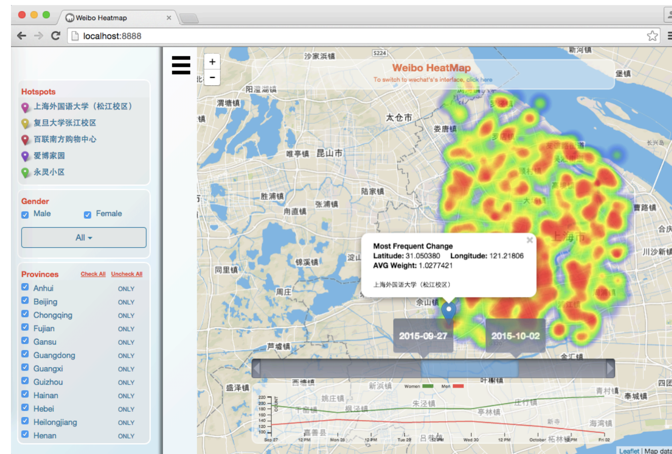


Figure 3.5: Weibo Hotspots

behind the hotsot by correlating the location data with other structured and unstructured information returned by Weibo. These could include demographics information such as gender and home state of users along with the posts made near the hotspots.

Figure 3.5 shows the result of the application over Weibo. It highlights the hotspot activity over Shanghai province of China between the given time interval. The left pane shows the list of identified hotspots that are also visualized over the map in the form of heatmap. The user could also filter the data by focussing on gender, province etc. In addition, it also displays the gender distribution over the period of interest.

3.4 Summary

We proposed to demonstrate ANALOC, a system for estimating aggregates over LBS with kNN interfaces. In contrast to prior demos, it does not assume complete access to location data and is one of the first systems to leverage kNN interface. ANALOC provides analysts with a valuable tool for performing aggregate estimates. Its extensible architecture could easily be leveraged by third parties for enabling multitude of applications such as mashups, visualizations, infographics and other sophisticated analytics over LBS data.

CHAPTER 4

Density based Clustering over Location Based Services

Location Based Services (LBS) have become extremely popular over the past decade, being used on a daily basis by millions of users. Instances of real-world LBS range from mapping services (e.g., Google Maps) to lifestyle recommendations (e.g., Yelp) to real-estate search (e.g., Redfin). In general, an LBS provides a public (often web-based) search interface over its backend database (of tuples with 2D geolocations), taking as input a 2D query point and returning k tuples in the database that are closest to the query point, where k is usually a small constant such as 20 or 50. Such a public interface is often called a k-Nearest-Neighbor, i.e., kNN, interface.

In this paper [46], we consider a novel problem of enabling density based clustering over the backend database of an LBS using nothing but limited access to the kNN interface provided by the LBS. Specifically, a key limit enforced by most real-world LBS is a maximum number of kNN queries allowed from a user over a given time period. Since such a limit is often orders of magnitude smaller than the number of tuples in the LBS database, our goal here is to mine from the LBS a *cluster assignment function* $f(\cdot)$, such that for any

tuple t in the database (which may or may not have been accessed), $f(\cdot)$ can produce the cluster assignment of t with high accuracy. We conduct a comprehensive set of experiments over benchmark datasets and popular real-world LBS such as Yahoo! Flickr, Zillow, Redfin and Google Maps and demonstrate the effectiveness of our proposed techniques.

4.1 Introduction

Location Based Services (LBS): Real-world LBS provide search and recommendation for numerous types of geospatial and commercial information such as Points-of-Interest (POIs), restaurants, real-estate properties, etc. Popular examples range from mapping services (e.g., Google Maps) to restaurants reviews (e.g., Yelp) to real-estate search (e.g., Redfin). Besides these dedicated LBS systems, LBS-related features have been widely integrated into other web based systems, e.g., social media platforms such as Twitter, WeChat, Sina Weibo, etc.

Generally speaking, each LBS has a backend database where each tuple represents a geotagged entity (e.g., a POI in mapping services or a user in social media). Attributes of a tuple often capture both geographical coordinates (e.g., latitude and longitude) as well as other structured information such as POI name, review ratings, etc. Public access to an LBS database is usually limited to a web (or API) based search interface. Such an interface often allows only k-Nearest-Neighbor (kNN) queries - i.e., upon given a geolocation p and, optionally, a selection condition s, the interface returns a small number (up to a pre-determined constant k such as 20 or 50) of tuples in the database that, among those matching the selection condition s, are closest (geographically) to p.

In addition to the limit to kNN queries, the interface might enforce other constraints as well, a popular one being the *query rate limitation* - i.e., the LBS might cap the maximum number of queries one (e.g., a website user, an API token, or an IP address) can issue within

a given time period. Due to the limit of returned tuples to k and the query rate limitation, it is usually impossible to crawl a reasonably-sized backend database of an LBS through its public-access interface.

Problem Definition and Motivation: The backend database of an LBS is often a gold mine of information for understanding the corresponding application domain. For example, data stored in real estate LBS such as redfin.com offer critical insights into the geographic spread of wealth, education quality, etc., while POI data such as those in Google Maps can support mining the spatial patterns of lifestyle choices such as bar themes, restaurant cuisines, etc.

Unfortunately, due to the aforementioned limitations, access to such invaluable data is restricted to the LBS provider itself, making it extremely difficult for unaffiliated *third parties*, e.g., social-science researchers, business analysts, etc., to take advantage of the data. We aim to enable the analytics and mining of such data by using nothing but the restrictive, public-access, interface of the LBS, making it possible for third parties to enjoy the value of LBS data without the lengthy and expensive negotiation process with the LBS provider.

More specifically, the objective of this paper is to study a novel problem of enabling *spatial clustering over an online LBS database* by issuing only a small number of kNN queries supported by the LBS interface. Clustering is one of the key problems in spatial data mining, with a wide range of applications. For example, by performing clustering over the geocoded tweets at Twitter, a third party may identify hotspots or popular events. Similarly, clustering over real-estate data such as Redfin can unveil the areas where citizens of different socioeconomic status live. While many spatial clustering algorithms have been studied in the literature, the objective of this paper is *not* to select the best-performing algorithm for LBS data, but to instead demonstrate the feasibility of enabling spatial clustering

using nothing but a few kNN query answers. For this purpose, we consider as a baseline a fundamental yet popular density-based clustering algorithm, DBSCAN [1], and develop a DBSCAN-like algorithm for LBS data with only a kNN interface for data access. While we focus on developing a DBSCAN-like algorithm in this paper, our overall objective is to enable the algorithmic design for a given cluster definition over a LBS with limited kNN interface and query cost limitations.

Technical Challenges: There are many challenges in enabling spatial clustering over LBS data with only a kNN query interface. The two most critical ones are on the *input* and *output* of the clustering algorithm, respectively. The foremost challenge is on the *input* side - i.e., there is no direct way for a LBS client to run a clustering algorithm like DBSCAN, since the user can only formulate kNN search queries, not the density-calculating queries required by DBSCAN. Any approach that works has to be based on executing a set of kNN queries via the restrictive query interface, and then "inferring" the underlying clusters structure from the results of the queries.

The second challenge is on the *output* side. To understand why, note that a faithful implementation of any spatial clustering algorithm (including DBSCAN) requires us to retrieve each tuple at least once, because the output is supposed to be a labeling of each tuple into the cluster it belongs to. This requires numerous (at least n/k where n is the total number of tuples) queries through the kNN interface, as each query returns at most k results. This query cost is often prohibitively expensive in practice, violating the rate limits or budgetary constraints imposed by the LBS interface (e.g., Google map API imposes a query rate limit of 10,000 per user per day).

To address these two challenges, we have to adjust both the input and the output of a clustering algorithm. Specifically, the output here can no longer be the identification of the cluster affiliation for each tuple in the database. Instead, we aim to construct a clustering

function $f(\cdot)$ by issuing a small number of kNN queries, such that for any given tuple t in the database which *may or may not* have accessed, $f(\cdot)$ outputs the cluster assignment of t with high accuracy. While this might seem a deviation from the traditional clustering definition, the two problems are essentially the same. Specifically, note that by having a "perfect" construction of $f(\cdot)$, we would have accurately identified the number of clusters and a few tuples in each cluster. The accuracy of this $f(\cdot)$ function can also be evaluated in the same way as traditional clustering - i.e., by applying $f(\cdot)$ over all tuples in the database, we can compare the distance between its outputs and the ground truth (or the output of a traditional clustering algorithm) to assess the accuracy of $f(\cdot)$.

A seemingly simple baseline would be to first sample the database (such as [47] that provides random samples from a LBS database using only kNN interface), and then run DBSCAN over the retrieved samples. To construct $f(t)$ for a new (i.e., not-sampled) point t, we can simply output the nearest cluster to t (e.g., based on the distance between t and the cluster centroid) as its cluster assignment. This approach, while conceptually simple, also suffers from a number of issues. For example, it might mishandle arbitrarily shaped clusters - a key strength of density based clustering - for tuples not included in the sample. Additionally, it does not have an effective way to identify outliers, given the sparsity of the samples. Finally, it requires a substantially large sample size to enable a reasonably accurate identification of the clusters.

Orthogonality with Traditional Clustering Research: Our problem is significantly different from density based clustering over traditional databases. Often, they have access to the entire database. Therefore, their key objective is to reduce the computational cost. In contrast, our algorithms only has limited access to the database (i.e., kNN queries only). As such, our objective is to perform a best-effort clustering (due to the limited data access) while minimizing the number of queries issued over the kNN interface. The computations that are performed locally at the client (i.e., query-issuer) side is not a major limiting factor.

In other words, our objective is not to challenge or improve prior cluster definitions - but to enable the algorithmic design according to a given cluster definition over an LBS with a limited kNN interface.

Outline of Technical Results: DBSCAN is based on two key parameters ϵ and *minPts*, and defines a cluster as a set of points (with cardinality at least *minPts*) that are within a radius of ϵ from at least one other point in the cluster. It considers a density measure equal to the number of points within a pre-determined radius ϵ. Thus, a requirement for implementing DBSCAN over LBS is the ability to compute the density at any given spatial point.

A baseline approach to simulate DBSCAN is as follows. If we partition the data space into grid cells of length ϵ on each dimension, then DBSCAN can then be almost faithfully executed as long as we can accurately estimate the number of points in each cell. Given that the key information required is whether the density of a grid cell exceeds the pre-determined density threshold *minPts*, our problem is reduced to estimating a Boolean indicator for each grid cell - whether the number of points falling within exceeds the given threshold *minPts*. This can be easily computed by issuing a kNN query q at the center of the cell. If all k returned points fall within the cell, then we return TRUE for the cell. Otherwise, we can return FALSE for not only the cell of q, but also all other cells completely covered within the range of the k returned points.

However, this approach can be extremely inefficient because it requires examining each cell in the grid - equivalent to visiting all points in a cluster. Hence, we need an efficient mechanism to find the *boundaries* of the cluster by "skipping over" intermediate points that are close to each other. Representations of these boundaries serve as our clustering function $f(\cdot)$, since any new point can be mapped into the appropriate cluster by checking whether it lies inside or outside the cluster boundary. The problem of discovering cluster boundaries is studied in literature [48, 49]. However, like traditional clustering algorithms they are only applicable when we have full access to the database. Instead,

our algorithms start by first discovering a point with high density, and then quickly traverse to the boundary of the cluster that contains that point, without having to examine all intermediate points.

We first develop algorithms for the special case of one dimensional data, and then extend them to two (and higher) dimensions. In the 1D case, each cluster is essentially a dense segment, and our goal is to discover the boundaries of each dense segment. Our algorithm starts from a dense point within a cluster and discovers the two boundary points by going to the left and right using a binary search-like process.

In contrast to 1D case where cluster boundaries of a dense segment can be discovered by going to the left and right side from a point inside the segment, the direction that we need to follow in 2D scenario is not very clear, especially when clusters can have any arbitrary shape. In this case, we use an innovative approach of mapping the points in 2D space to 1D using a *space filling curve* (SFC [2–4]), and then discover the clusters using the 1D clustering algorithm. A well designed SFC guarantees that two points close to each other in the mapped 1D space are also close together in the original 2D space. This property fits our purpose since we can skip points (by binary search) that are close to each other in mapped 1D space as they will also be potentially inside the same cluster in original 2D space. However, we must caution that points that are close to each other in 2D space might not be close in the mapped 1D space. Hence, a cluster in 2D space might split into many small clusters in 1D space. This complication can be addressed by a post-processing step of merging 1D clusters that are "close" to each other, eventually into 2D clusters.

Summary of Contributions: Our paper makes the following major contributions:

- We consider the novel problem of density-based clustering over location based services using only a kNN query interface.

- For 1D data, we develop HDBSCAN-1D, a binary-search based algorithm for quickly discovering the cluster boundaries without having to crawl all the data within each cluster.
- For 2D data, we develop HDBSCAN, a clustering algorithm that uses an adaptive space-filling curve algorithm as a subroutine to map 2D data to 1D, clusters the 1D data using the 1D clustering algorithm developed earlier, and then merges the resulting 1D clusters into eventual 2D clusters. This approach can be generalized to high dimensional data.
- We conduct a comprehensive set of experiments to demonstrate the effectiveness of our algorithms. Specifically, we conduct online tests over a number of real-world LBS, such as Yahoo! Flickr, real estate sites such as RedFin and Zillow and Google Places. In addition, we also experiment with well-known synthetic datasets used in prior density-based clustering research, such as Chameleon t7.10k, t4.8k and t8.8k.

The rest of the paper is organized as follows. Section 4.2 introduces the LBS data model and formalizes the problem of enabling DBSCAN over LBS. Section 4.3 develops algorithm HDBSCAN-1D for 1D data that highlights the basic ideas of our sampling based design. Section 4.4 considers the general case of clustering higher-dimensional data and develops algorithm HDBSCAN. Section 4.6 describes the experiments over popular benchmark datasets and real-world LBS. We describe the related work in Section 4.7 followed by final remarks in Section 4.8.

4.2 Background

4.2.1 Model of LBS

Consider a Location Based Service (LBS) over a database D of n tuples, each of which is labeled with a 2D location and possibly other relational attributes. Note that the

results of this paper can be directly extended to 3D (or higher dimensional) locations, as we shall discuss in Section 4.4. Examples of such an LBS include Google Maps, Redfin, etc., where each tuple is a point of interest such as restaurant or real estate property; as well as online social networks such as WeChat where each tuple is a user. In these examples, a tuple features not only its 2D location, but also other attributes such as restaurant rating, user gender, etc.

LBS usually supports kNN queries over the database. Such a query takes as input a 2D location q (e.g., $\langle$longitude, latitude$\rangle$), and returns the top-k nearest tuples in D to q as determined by a pre-defined distance function. We consider Euclidean distance as the distance function. For each returned tuple, the query answer includes both its location and other attributes. Note that while the value of k varies on different real-world LBS systems, it is generally at the range of 50 to 100 - e.g., Google Maps has $k = 60$, while $k = 50$ and 100 for WeChat and Sina Weibo respectively. Most LBS also impose additional restrictions such as *query rate limit* - i.e. the maximum number of kNN queries that can be issued per unit of time. For example, by default Google Maps allows 10,000 location queries per day while Sina Weibo allows only 150 queries per hour. Given these query limits, a key goal of our algorithms is to minimize the query cost.

4.2.2 Problem Definition

Objective of Clustering: As discussed in the introduction, we consider in this paper how to enable clustering over an LBS that exposes nothing but the above-described, limited, kNN interface. An important observation here is that, with the limitation imposed by real-world LBS systems, the definition of clustering will inevitably change in our problem setting. To understand why, note that the traditional definition of clustering is to assign a cluster ID for every tuple in the database (with the possibility of NULL ID for tuples deemed noise). If

we adopt the same goal, this means *accessing* each of the n tuple at least once through the interface, which requires no fewer than n/k queries (because each query returns at most k tuples). Given the large n, small k, and stringent query rate limit in real-world LBS systems, this query cost is often prohibitively expensive.

To address the challenge, the objective of clustering in our setting is to output a *cluster-assignment* function $f(\cdot)$ which, upon given a tuple $t \in D$ (which may have never been accessed by our algorithm), outputs the cluster ID of t. Note that this is indeed a *generalization* of the original clustering definition, and the traditional practice of producing a cluster ID for each tuple can also be considered as producing a function that maps each tuple to an ID.

Orthogonality with Traditional Clustering Research: Before discussing the performance measures for our clustering-over-LBS problem, it is important to make a proper distinction between the main objective of this paper and that of traditional clustering research. In both cases, the ultimate goal is to produce $f(\cdot)$ that perfectly resembles the ground-truth cluster assignment of all tuples, e.g., as determined by human experts. The key challenge, however, is completely different.

One can roughly partition the design of a clustering solution into two parts: the *definition* of clusters and the *algorithmic design* of efficiently clustering a given database according to the cluster definition. Many existing work on clustering contribute to both fronts - e.g., k-means clustering defines clusters according to the distance between a point and the k cluster centers, while density based clustering defines clusters according to how points are closely packed together. Our objective in this paper is *not* to challenge or improve the cluster definitions of prior work, but rather to enable the second part - i.e., algorithmic design according to a given cluster definition - over an LBS with a limited kNN interface. From this perspective, our goal here is largely orthogonal to traditional clustering research.

Performance measure: There are two important performance measures for clustering over LBS: (1) the efficiency of clustering, and (2) the quality of clustering output $f(\cdot)$. For efficiency, the key bottleneck in our problem setting is the query-rate limit imposed by real-world LBS. Thus, we focus on one efficiency measure in this paper: *query cost*, i.e., the number of queries the clustering algorithm has to issue in order to produce $f(\cdot)$. Note that traditional efficiency measures, e.g., the computational and storage overhead of the clustering algorithm, are all secondary concerns in our problem because of the limited input size - note that the number of "input" tuples, i.e., tuples that can be retrieved from the underlying database, is inherently bounded by k times the query cost, which is likely a small number in practice due to the query-rate limit.

In terms of clustering quality, we need to compare $f(D)$, i.e., the outputs of $f(\cdot)$ for all tuples in D, with a reference set of clustering IDs that can be either the ground truth or the output of a traditional clustering algorithm over the entire D. In either case, the difference between $f(D)$ and the reference set can be measured in a variety of metrics commonly used in clustering research [50]. In this paper, we consider three metrics in experimental analysis: *Rand index*, *Jaccard index* and *Folkes and Mallows index* [50], respectively.

Let $P_1, P_2, \ldots P_h$ be the clusters produced by $f(D)$. Assume $D = P_1 \cup \cdots \cup P_h$, as points deemed noise (i.e., with $f(t)$ being NULL) can be considered as all belonging to a "noise" cluster. Let $C_1, C_2, \ldots C_{h'}$ be the ground-truth clustering result, again with $D = C_1 \cup \cdots \cup C_{h'}$. The design of Rand, Jaccard, and Folkes and Mallows indices all consider the following four critical numbers:

- a, the COUNT of pairs of tuples in D, say t, t', that belong to same cluster according to both P and C, i.e., $\exists i \in [1, h]$ and $j \in [1, h']$, such that $\{t, t'\} \subseteq P_i$ and $\{t, t'\} \subseteq C_j$.

- b, the COUNT of pairs of tuples in D that belong to different clusters according to both P and C.
- c, the COUNT of pairs of tuples in D that belong to the same cluster according to P but different ones according to C.
- d, the COUNT of pairs of points that belong to different clusters according to C but the same one in P.

The Rand Index measure (R), Jaccard Index (J) and Fowlkes and Mallows index (FM) are defined as below:

$$R = \frac{a+b}{a+b+c+d}, J = \frac{a}{a+c+d}, FM = \frac{a}{\sqrt{(a+c)\cdot(a+d)}}$$

4.2.3 Density Based Clustering

Since most real-world LBS focus on 2D points, we consider density-based clustering, a popular class of techniques for low-dimensional data [51]. Before discussing how to enable density-based clustering over LBS in the technical sections, here we briefly review its basic design in the traditional setting of a database with full access.

Density based clustering algorithms use the density property (e.g., reachability) of points to partition them into separate clusters. Specifically, the output depicts dense clusters (of points) separated by low-density regions. DBSCAN [1] is the an example. It takes two parameters as input: ϵ, the radius of a region under consideration, and $minPts$, the minimum number of points inside a region for it to be dense. Specifically, a point $t \in D$ is considered *core* if there are at least $minPts$ points within distance ϵ of t. Two points $t_0, t_r \in D$ are *reachable* from each other if there is a sequence of core points $t_1, \ldots, t_{r-1}$, such that $\forall i \in [0, r-1]$, t_i and t_{i+1} are within distance of ϵ from each other.

Given ϵ and $minPts$, the definition of a cluster becomes straightforward. Specifically, a core point t is clustered together with all points in D that are reachable from t. If a point is not reachable from any other point, it becomes an outlier, i.e., noise. Other density-based clustering techniques follow similar principles, but measure density in different ways, leading to different *definitions* of clusters. For example, DENCLUE [52] defines the density of a point as a sum of the influence function of the other points in D. OPTICS [51] generalizes the density definition of DBSCAN by enabling different local densities with $\epsilon' \leq \epsilon$ [51].

As discussed earlier in this section, to enable clustering over LBS, we inevitably have to choose a cluster definition to follow. For the purpose of this paper, we consider the simple $(\epsilon, minPts)$-based definition of DBSCAN, and aim to produce a cluster assignment function $f(\cdot)$ with $f(D)$ being as close to the clusters defined by DBSCAN as possible (as measured by the aforementioned metrics R, J and FM). One special note here is that, since $minPts$ is often set to be a small value such as 20 in practice [1], we assume $minPts \leq k$ (as in the kNN interface offered by the LBS). This way, whether t is a core point can be determined by just one query (i.e., on t, by judging whether the $minPts$-th returned point is within ϵ from t). In case $minPts > k$, one can always call the kNN based crawling algorithm [53] to first crawl the $2\epsilon \times 2\epsilon$ square surrounding t and then make the determination.

4.3 1D Case

We now consider how to enable clustering over the kNN interface of an LBS. As discussed in the introduction, we start by developing HDBSCAN-1D for 1D data. The simplicity of the 1D setting allows us to highlight the basic idea of our sampling-based

design. Additionally, the general technique we develop in the next section leverages the 1D algorithm as a subroutine for solving the high-dimensional problem.

4.3.1 1D Baseline and Problem

Since we follow the cluster definition in DBSCAN, we start by considering a partitioning of the 1D space into cells of equal width, ϵ. We call such a cell "dense" if there are at least $minPts$ points in it, and "sparse" otherwise. Note that, so long as one can somehow determine (by issuing kNN queries) which cells are dense and which are not, the result of DBSCAN can be almost faithfully replicated by joining adjacent dense cells to form a cluster.

It is easy to determine the density of a single cell. One simply needs to issue a kNN query q at the center of the cell, and see whether the nearest $minPts$ points returned (recall from Section 4.2 that $minPts \leq k$) all fall within the cell. If so, the cell must be dense. Otherwise it must be sparse.

Extending the solution to determine the density of all cells, however, is not easy. A baseline solution here is to select cells uniformly at random, and issue queries at their centers to determine their density. The problem, however, is its high query cost. Note that any kNN query answer can "cover" at most $k/minPts$ dense cells. Thus, generally speaking, if there are D_c core points in D, the number of queries required is at least $D_c \cdot minPts/k$. Given the small k offered by real-world LBS systems, this query cost tends to be prohibitively expensive in practice.

4.3.2 Algorithm HDBSCAN-1D

Key Idea: To address the problem with the baseline solution, our main idea is to leverage the *locality* of cell densities - i.e., cells adjacent to each other are highly likely to have similar density values. This locality property, of course, is not new. It has been used in

the design of many density-based clustering algorithms - e.g., DENCLUE [52] reduces the problem of density based clustering to kernel density estimations, essentially assuming neighboring cells to have similar densities according to a Gaussian mixture model.

Specifically, our HDBSCAN-1D differs from the baseline on how to deal with a query q that returns $minPts$ points within a cell. In addition to marking the cell as dense, we also aim to (approximately) identify the *boundary* of the entire cluster surrounding the cell - i.e., the maximal sequence of cells, containing the one queried, that are all dense. The rationale here, of course, is that according to the locality property, the number of such sequences is much smaller than the number of dense cells in the space.

Consider a query q which returns at least *minPts* points within cell q - note that here we use the same notation to represent the query and the cell without ambiguity, because a query we issue is always right at the center of the 1D cell. Our idea is to first find a *sparse* cell to the left (resp. right) of q in the 1D space, and then identify the dense cell immediate to the right (resp. left) of the sparse region as a candidate for the left (resp. right) boundary of q's cluster. We discuss this process in detail as follows.

First, we set an initial range (a, b) based on the nearest sparse cells to q that we already know. That is, both a and b are already discovered sparse cells with $a \leq q \leq b$. If no sparse cell has been discovered, we can set a and b as the boundaries of the value domain. We start by finding the first dense cell to the right of a. To do so, we issue a query at $a+1$. If $a+1$ returns at least one point to its right, say at cell a', then we test the density of a' by issuing a' as a query. If a' is dense, we have accomplished our task. Otherwise, since we now know that a' is sparse, we can shrink the range to (a', b) and repeat this process.

Once we have identified the first dense cell to the right side of a, say d, our task now is to determine if $[d, q]$ consists solely of dense cells - i.e., if d is the left boundary for the cluster containing q. We sample c cells uniformly at random from $[d, q]$ to test if all of them are dense. If all c cells turn out to be dense, we consider the range to be all-dense, i.e.

continuous range of dense cells. Alternatively, we find a sparse cell $d' \in [d, q]$ - as soon as we discover d', we repeat the entire process with an updated initial range of (d', b).

One can see that this process eventually leads to a range $[d, e]$ where $q \leq e \leq b$ containing q that pass the c-sample test of being consecutive dense cells. What we do next is to sample uniformly at random a cell for which we cannot yet determine density, and query the cell to repeat the above-described process. Once again, this can be repeated until the density nature of all cells have been determined, or until all query budget has been exhausted.

Algorithm 8 HDBSCAN-1D

1: **while** query budget is not exhausted
2: Issue kNN query over randomly chosen unvisited cell q
3: **if** q is dense
4: (a, b) = range containing q where both a and b are already discovered sparse cells.
5: l, r = left and right boundary discovered using binary search inside range [a+1, q] and [q, b-1] respectively.
6: Add the cell range [l, r] to dense segment list
7: Output the dense segments (clusters) identified so far.

Query Cost Analysis: We start by considering an ideal case where all tuples in the database belong to one of the h clusters - i.e., there are no outlier points. During the initial search of the first dense cell to the right of a, there are only two possible outcomes for each query we issue: either it shows the cell to be dense and triggers a c-cell density test, or it returns no tuple to its right side at all.

For the first case (i.e. the c-cell density test), note that each failed test identifies a new cluster - i.e., we conduct the test at most h times, consuming $O(h \cdot c)$ queries in the worst-case scenario. When the query returns no point to its right, however, the situation can be more complex. Note that if a query a' returns no point to its right, then all the k nearest neighbors to $a + 1$ are on its left side. What this triggers is a process that we refer to as *exponential search*. Specifically, let ℓ be the maximum distance between $a' + 1$ and the k returned points. Since we are now certain that no tuple resides within $[a', a' + \ell - 1]$, the search space is shrunk to $[a' + \ell, b]$ - i.e., the next query we issue will be at $a' + \ell$. This query either returns a point to the right side of a, or proves that no point resides in $(a, a + 3\ell)$. If it returns a point to the right side of a, the c-sample test is triggered, with query cost falling within the $O(h \cdot c)$ queries in the above analysis. Otherwise, if $a + \ell$ still does not reveal any point to the right of a, i.e., our next queries to issue would be $a + 3\ell, a + 7\ell, a + 15\ell, \ldots$ - representing exponential growth of the query value. Note that each exponential search consumes $O(\log N/\epsilon)$ queries, where N is the total number of cells. The exponential search process can occur at most $h + 1$ times, corresponding to the $h + 1$ empty segments in between the h clusters. As such, the overall query cost becomes $O(h \cdot c + h \cdot O(\log N/\epsilon))$.

Handling Noisy Points: While the technique described above shows a significantly reduced query cost according to the above analysis, it actually has an important problem masked by an assumption made in the analysis - i.e., there is no outlier in D and every point belong to a cluster. Note that if a' returns a point to its right which nonetheless does not turn out to be a dense cell, then the next query issued would be $a' + 1$ (instead of much further to the right as in the case of exponential search). In other words, in the worst case scenario where every cell is filled with fewer than *minPts* but at least one point, the

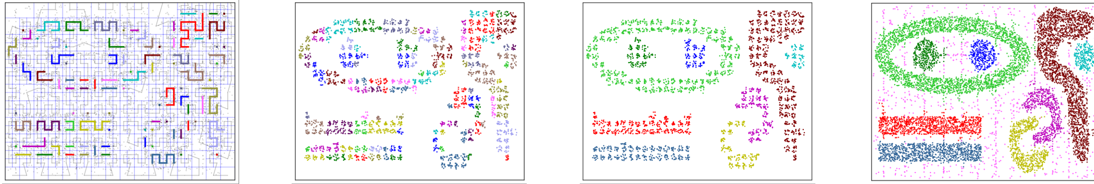


Figure 4.1: Mapping 2D space to 1D using Adaptive SFC

Figure 4.2: After clustering in 1D space

Figure 4.3: Merging nearby clusters in 2D space

Figure 4.4: Assigning new points to closest cluster

technique might have to enumerate all cells between a and the left boundary of the cluster containing q.

To address the problem, we introduce a binary search process for this scenario. Specifically, we start by querying $(a' + q)/2$. If it is dense, we move to the left (i.e., $(3a' + q)/4$, $(7a' + q)/8$, etc., in order). If it is sparse, we move to the right. One can see that this process always terminates when we discover a dense cell that has its immediate neighbor to the left being sparse. In other words, we have discovered the left boundary of a cluster - whether the cluster is the one surrounding q will be verified by the c-sample test.

One can see that each execution of this binary search process returns one of the two boundaries for a cluster. Thus, it is executed for $O(h)$ times in the worst case. In other words, the overall query cost remains $O(h \cdot c + h \cdot O(\log N/\epsilon))$.

4.4 HDBSCAN

In this section, we consider the general case of clustering higher-dimensional data through a restrictive kNN interface. While it will be clear from the discussions that our results can be directly extended to data of any dimension, the focus of this paper is on 2D data because of its prevalence among real-world LBS.

4.4.1 Overview

One can see from the design of HDBSCAN-1D that the "boundary" pursuit idea used there cannot be easily extended to higher dimensions, because instead of having just 2 bounds (left and right) as in 1D, there may be a large number of cells bounding a 2D (or higher-D) cluster of an arbitrary shape, making the exact discovery of them extremely expensive in query cost.

To address this challenge, our idea is to first map 2D data into a 1D space, and then call upon HDBSCAN-1D to perform clustering as described in §4.3. To enable the mapping, we use a specially designed *Space Filling Curve* (SFC) detailed later in the section. The general concept of SFC [2–4] is illustrated in Figure 4.5. More specifically, for each 2D point t, we records as its mapped 1D coordinate $S(t)$ the point on the SFC that is closest in distance to t (in the 2D space). With this design, each query HDBSCAN-1D decides to issue is mapped to its corresponding 2D coordinates and issued to the underlying 2D LBS, while every returned 2D tuple t is mapped back to its 1D coordinate $S(t)$ for HDBSCAN-1D to process. One can see that this two-way mapping enables the seamless execution of HDBSCAN-1D over the 2D space. In addition, it ensures that two tuples $t_1, t_2 \in D$ (indeed, their mappings $S(t_1)$ and $S(t_2)$) belonging to the same cluster produced by HDBSCAN-1D should also be clustered together in the 2D space. Figure 4.6 shows a mapping from 2D to 1D space using Hilbert curve.

Nonetheless, this SFC-based mapping also introduces a major challenge to the clustering design: Since no SFC can guarantee that two points close together in 2D are always close in 1D [2], we are left with the possibility that one 2D cluster may be partitioned into multiple 1D clusters after the mapping. To address the problem, we introduce an additional step of *merging* the "mini-clusters" produced by HDBSCAN-1D. Specifically, recall from §4.3 that HDBSCAN-1D outputs h mini-clusters as ranges $C_i : [a_i, b_i]$ ($i \in [1, h]$). We con-

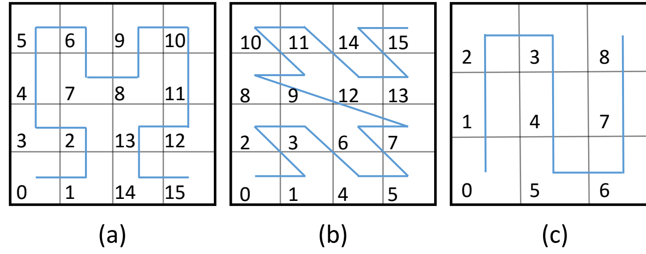


Figure 4.5: Illustration of popular SFC: (a) Hilbert curve (b) Z-curve (c) Peano curve

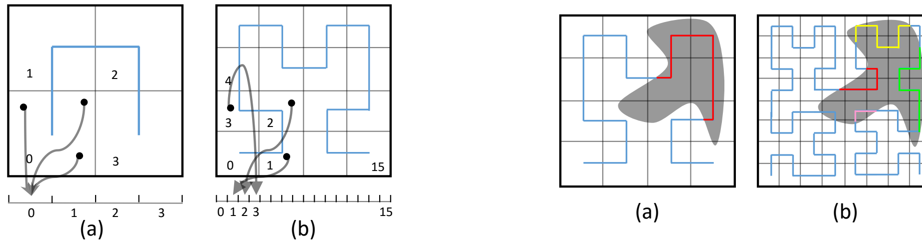


Figure 4.6: Mapping 2D points to 1D through Hilbert SFC

Figure 4.7: Impact of Minimum Grid Size on Clusters

sider for each pair[1] of mini-clusters whether they should be merged (with details discussed later in the section), according to the 2D points we have observed in each mini-cluster. This merging process essentially produces a many-to-one mapping $C_i \rightarrow C'_j$ ($i \in [1, h]$, $j \in [1, h']$), so each 1D range C_i is labeled with a final cluster ID from 1 to h' ($h' \leq h$).

In the following subsections, we shall discuss, respectively, the three critical steps for HDBSCAN: (1) the design of SFC for calling HDBSCAN-1D, (2) the merging of mini-clusters, and (3) the generation of clustering function $f(\cdot)$. Figures 4.1 - 4.4 shows the output of each steps described above for Chameleon t-7.10k dataset. Points that are detected as noise are colored in pink.

[1]While this pairwise-testing appears to be an expensive (quadratic) process, note that the cost incurred here is local computational overhead instead of query cost, the bottleneck in our problem setting.

4.4.2 Mapping 2D to 1D

Baseline Solution and Problem: To design the mapping from 2D to 1D, a baseline solution is to first partition the 2D space into $\epsilon \times \epsilon$ grid cells, and then apply standard SFCs such as Hilbert curve, Peano Curve, Z-curve, etc. This way, the mapped 1D space is naturally partitioned to cells of width ϵ, exactly matching the need of HDBSCAN-1D. Meanwhile, the mini-clusters produced by the algorithm, when mapped back to 2D, will consist of adjacent $\epsilon \times \epsilon$ cells, closely approximating the ϵ-radius neighborhood considered by the original DBSCAN.

A problem with this baseline solution, however, is the large number of grid cells defined by the mapping. Recall from the query-cost analysis in Section 4.3 that the number of queries required by HDBSCAN-1D is proportional to h, the number of clusters - in this case mini-clusters - identified by the algorithm. From the illustration in Figure 4.7, one can clearly see that, for a fixed Hilbert SFC, the more fine-grained the 2D grid cells are, the more mini-clusters it will partition a true 2D cluster into. When the granularity is down to $\epsilon \times \epsilon$ as in the baseline, the number of mini-clusters produced may far exceed the number of real clusters (i.e., $h \gg h'$), leading to a very large query cost.

On the other hand, it is also important to note that we cannot arbitrarily enlarge the grid cell size to reduce the count. To understand why, note that the mapping from 2D to 1D might project two points at the two opposite ends of a 2D cell to the same 1D coordinate. For example, consider the bottom left cell in Figure 4.6. Both the top left corner and the bottom right corner of the cell will be mapped to the 1D point at the center bottom of the 2D cell. This is usually not a problem for clustering when each cell is small (e.g., $\epsilon \times \epsilon$). However, if we make the cell size too large, then this mapping-induced error might cluster together two points far from each other, adversely impacting the quality of clustering results.

Adaptive Space Filling Curve (SFC): To address the problem, our key idea here is to introduce the concept of an *adaptive SFC*, which combine larger grid cell sizes on sparser regions, in order to reduce the overall query cost, with smaller grid sizes on denser regions, in order to gain enough resolution to separate clusters from each other and outlier points. Figure 4.8 depicts an example of such a space filling curve.

Note that the shape of the adaptive SFC depends on the underlying data distribution. Since we do not have prior knowledge of the data distribution, we can no longer pre-define this SFC and its corresponding 2D to 1D mapping. Instead, we have to construct the adaptive SFC on-the-fly and adjust the mapping as the SFC changes. Specifically, this online process can be described as follows.

- We start with the largest possible grid cells, i.e., by partitioning the entire space into four cells, as shown on the root node in Figure 4.8. Based on this initial SFC, we start the execution of HDBSCAN-1D. Note that the design of the adaptive SFC is transparent to HDBSCAN-1D - it simply takes a 1D space as input and is oblivious to how large the cell sizes are in the 2D space.
- For every query q issued by HDBSCAN-1D, we identify the 2D cell q falls into, say a cell of size $c \times c$, and issue a query q' at the center of the cell. If the $minPts$-th ranked point q' returns is farther than $\sqrt{2}c/2$ from the cell center, it means that there can be no more than $minPts$ points within the $c \times c$ cell, and we do not need to partition it. Otherwise, we partition the cell into four cells, as demonstrated in Figure 4.8.
- Note that once we decide to further partition a cell, the adaptive SFC changes, and so is the 2D to 1D mapping (as it "inserts" a number of 1D cells to the domain). Thus, starting from the next step in the execution of HDBSCAN-1D, we follow the new mapping for translating the 1D query and the 2D query answers.

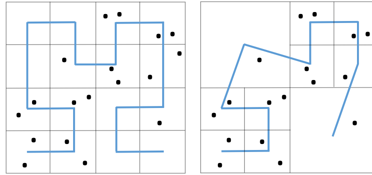


Figure 4.8: Standard vs Adaptive SFC

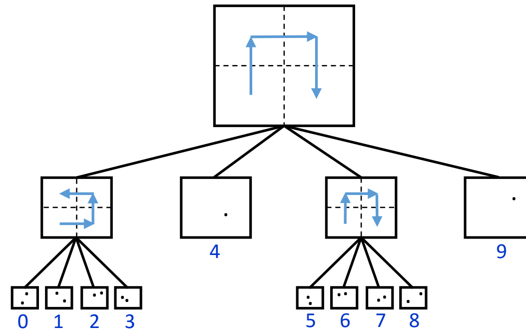


Figure 4.9: Adaptive SFC

4.4.3 Merging Mini-Clusters

We now consider how to merge the mini-clusters generated by HDBSCAN-1D to real 2D clusters. A simple approach here is to merge mini-clusters containing grids that neighbor each other in 2D (i.e. grids that share an edge). A problem with this solution, as we found through experiments, is that it is likely to merge two clusters into one if the clusters happen to be close to each other. Setting the ϵ value very small might solve this, but this will also increase the query cost. To overcome this problem, we select a subset of points from each clusters as representative to compute the inter cluster distance. This is similar in nature to the concept of using fixed set of representative points to measure cluster distance in CURE [51], a hierarchical clustering algorithm. Specifically, we compute the l-distance between two mini-clusters - i.e., we identify all points in the two mini-clusters that have been observed in previous query answers. Then, we select the top-l pairs of points with the minimum distance from each other, and compute their average distance. We merge

the two mini-clusters if their l-distance falls below a pre-determined threshold. Setting a small value of l would split larger clusters as they do not capture the shape of the clusters. Empirically, we found that setting l to *minPts/2* provided best results.

Algorithm 9 HDBSCAN-2D

1: **Input:** Tree t
2: **while** query budget is not exhausted
3: Issue kNN query over unvisited node n chosen randomly
4: **if** n is dense
5: Find dense segment containing n using Algorithm 8 such that none of t's leaf nodes get partitioned in that process.
6: Add the dense segment to mini-cluster list.
7: Merge min-clusters using top-l distance
8: Output the dense clusters identified so far

4.4.4 Clustering of New Points

Recall that our objective is to develop a clustering function $f(\cdot)$ that emulates the output of density-based clustering algorithms such as DBSCAN. Given a new point $f(\cdot)$ can then be used to identify its cluster affiliation. For each new point, we first map it to 1D space using the adaptive SFC and check whether it belongs to any of the "mini-clusters" previously discovered. The point is then assigned to the final cluster generated by the merging procedure from §4.4.3.

4.5 Discussion

4.5.1 Leveraging External Knowledge

Section 4.4 describes the clustering algorithm with adaptive SFC when we don't have any apriori knowledge about the data distribution. However, if some external knowledge about the data distribution is available (such as POI in a LBS such as Google Maps), then we can leverage that to reduce the query cost of building the tree. For example, while clustering the dense regions in an area, its reasonable to assume that the clusters will be positively correlated with the population distribution of that region. So instead of building the SFC mapping tree from the scratch, we can build the initial tree according to this information. We start with the tree built using the external knowledge and use this to map 2D space into 1D. As the clustering process progresses, we query the grids represented by the leaf nodes of the tree and partition them further if required. This way the final structure of the tree gradually changes from the initial structure to reflect the actual data distribution.

Note that, the use of external knowledge doesn't change the accuracy of the output clusters even when the external knowledge is inaccurate and does not align with the actual data distribution. This is because the density of a grid is still validated by querying it. This approach of optimization using external knowledge doesn't always guarantee query saving and but often serves as a handy heuristic. In the best case, no additional kNN query need to be issued for building the tree while in worst-case leveraging the external knowledge slightly increases the total query cost. For example, in such scenario, the initial tree built using external knowledge might partition the sparse regions into many smaller grids and leave the dense region with large grids. However, in practice, this approach is very effective.

4.5.2 Special LBS Constraints

Many real-world LBS apply additional constraints on their query interface. One such constrain is *maximum radius* on the returned results - the distance between the query

location q and the returned result is bounded by a predetermined threshold $d_{\max}$. If there are are no point within the $d_{\max}$ distance from query location q, an empty result set is returned. Google Maps only returns POIs that are at most 50KM away from the query point.

If $d_{\max}$ is larger then ϵ, we can still retrieve all the points inside the query grid. However, when ϵ is larger than $d_{\max}$, one kNN query might not be sufficient to check the density of a grid. There are several approaches to tackle this problem. One straight forward solution is to crawl the query grid. We divide the query grid into smaller grids of size $\epsilon' \times \epsilon'$, where $\epsilon' = \epsilon / d_{\max}$. The final density of a grid is then computed by aggregating the result of kNN queries issued at each of the smaller grids. This approach increases the query cost due to the high cost of crawling and also diminishes the advantage of adaptive SFC by issuing many queries in sparse region. A more efficient approach is to use sampling to estimate the count in a grid cell. For example, prior work such as [47] can be used to estimate the count of points in a region.

Another constraint found in real-world LBS application is a more complex ranking function that doesn't solely depend on distance between query point, q and POI, t. For example, Google Places API allows users to rank results by "prominence". Handling this is not straight forward as in extreme case it is possible that every query inside the grid cell returns prominent points outside of it making crawling impossible. However, for most of the real-world applications the value of k large (Google Places API returns upto 60 nearby POI for a given query point). As long as the nearest POI is returned anywhere in the top-k result, we can always post process the query result in order to obtain it. Then using the nearest result, count estimation techniques such as from [47] can be used.

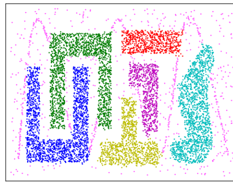


Figure 4.10: Chameleon t4.8k Dataset

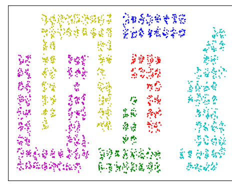


Figure 4.11: Output of HDBSCAN

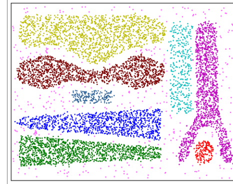


Figure 4.12: Chameleon t8.8k Dataset

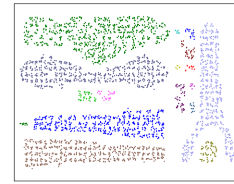


Figure 4.13: Output of HDBSCAN

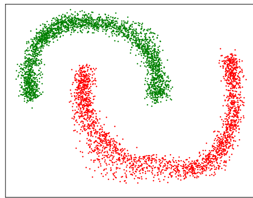


Figure 4.14: Banana Dataset

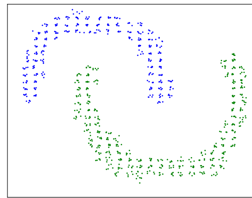


Figure 4.15: Output of HDBSCAN

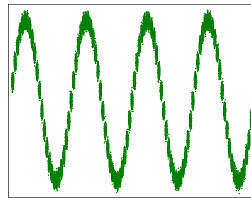


Figure 4.16: Birch R2 Dataset

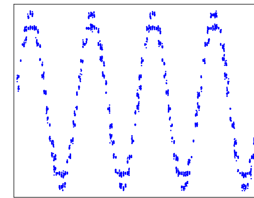


Figure 4.17: Output of HDBSCAN

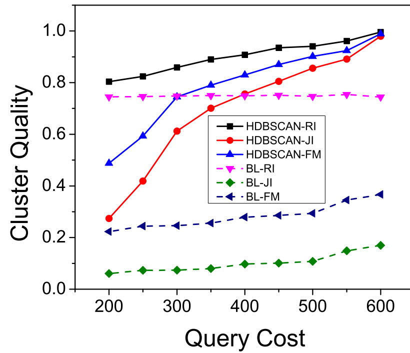


Figure 4.18: Comparison with baseline Algorithm

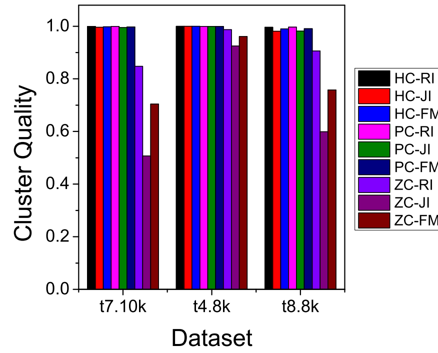


Figure 4.19: Cluster Quality for different SFCs

4.6 Experimental Results

4.6.1 Experimental Setup

Hardware and Platform: All our experiments were performed on a quad-core 2.5 GHz Intel i7 machine running Ubuntu 14.10 with 16 GB of RAM. The algorithms were implemented in Java.

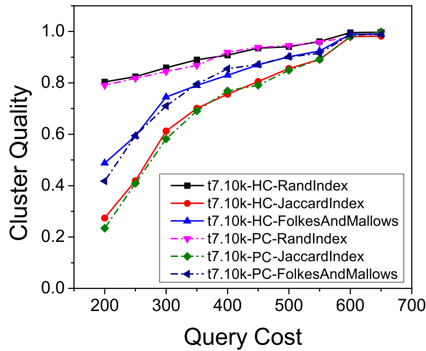


Figure 4.20: Cluster Quality Vs Query Cost

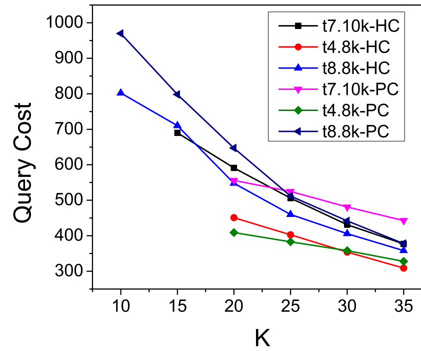


Figure 4.21: Varying K

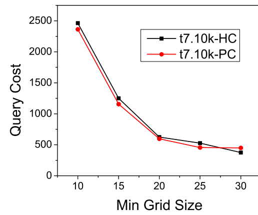


Figure 4.22: Varying min. grid size

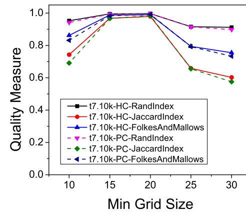


Figure 4.23: Varying min. grid size

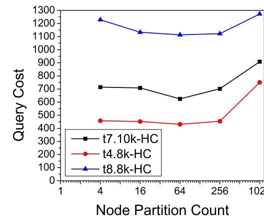


Figure 4.24: Varying number of Partitions per node

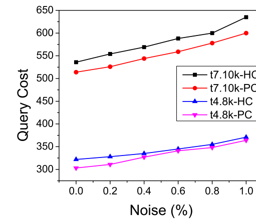


Figure 4.25: Varying percentage of Noise in dataset

Benchmark Datasets: We evaluated the clustering quality generated by HDBSCAN using the widely used Chameleon benchmark datasets [54]. They contain 2D points along with clusters of different shapes, densities, sizes and noises. We tested HDBSCAN on three Chameleon datasets: Chameleon-t7.10k, t4.8k and t8.8k. Note that for offline experiments we had full access to the dataset. To simulate the LBS access model, we implemented a kNN interface with Euclidean distance as the ranking function.

Real World Datasets: In order to highlight the practicality of HDBSCAN, we evaluated it against a number of popular real-world LBS such as Yahoo! Flickr, Zillow, Redfin and Capital BikeShare. Note that, unlike the experiments on benchmark datasets, we do not

know the ground truth cluster assignments. However, the cluster assignments produced by HDBSCAN had realistic clusters with valid real-world interpretations.

The Yahoo! Flickr dataset contained almost 100 million images of which about 49 million are geotagged. In addition to image metadata such as title, description and other user tags, it also contains location information. We considered the subset of almost 118K images from Washington DC with location granularity of street level. We implemented a kNN interface over the dataset that returns the images geotagged with locations near the query location. We can consider locations with large number of geo-tagged images as clusters (e.g. popular tourist hotspots).

Zillow is a popular online real state website that helps users to find houses and apartments for sale/rent. It has a kNN interface that allows users to search for houses in a given location and filter search results based on attributes like price, area, home type etc. We crawled approximately 12K houses listed for sale in Dallas Fort Worth (DFW) area along with metadata such as house location, price, number of beds and baths, area in square feet etc.

Redfin is an online real estate website similar to Zillow. The mobile application of Redfin provides a kNN interface where users can specify a location and get nearby houses sorted according to their distance from the query point. We executed HDBSCAN over the Redfin house listings for the DFW area.

Capital Bikeshare is a popular bicycle sharing system in Washington D.C. It has approximately 350 bike stations around D.C. They publish their rental data periodically every quarter. The dataset contains information about every rental such as start and end stations, rental date, duration etc. By combining this dataset with kNN interface provided by Google Places, we seek to cluster the stations based on the rental usage.

Performance Measures: We measured the efficiency of HDBSCAN through query cost, i.e. the number of queries issued to LBS. The clustering quality is measured based on Rand index, Jaccard index and FolkesAndMallows index as defined in §4.2.

Parameter values: Table 4.1 shows the parameter values of HDBSCAN for each dataset. The value of l was set to *minPts / 2*. The parameters are set to the values that provided the best accuracy for the offline execution of DBSCAN that had access to the entire dataset.

Table 4.1: Parameter values used in the experiments

Dataset	ϵ	$minPts$	# points	k
Chameleon t7.10k	20	14	10,000	14
Chameleon t4.8k	20	19	8,000	19
Chameleon t8.8k	15	6	8,000	6
Zillow	0.02	10	12,250	10
Yahoo Flickr DC	0.0025	600	117,875	600
Redfin	0.02	10	NA	200
Capital Bikeshare	0.005	3	NA	20

4.6.2 Experiments over Benchmark Datasets

Feasibility of HDBSCAN: In our first set of experiments, we show that it is indeed possible to identify underlying cluster structures (even for complex shapes with varying sizes) using a restricted kNN query interface and limited sample size. Figure 4.10 visualizes the benchmark dataset Chameleon 4.8k while Figure 4.11 visualizes the output of HDBSCAN over the dataset (using only the sample points) with a query budget of 400. To reduce clutter, we removed the noise points. Figures 4.12 and 4.13 show the results for Chameleon 8.8k. As the figures show, HDBSCAN followed by post-process merging of nearby clusters could identify the clusters using only the local view of the dataset. For the rest of the benchmark datasets experiments, we focus on the Chameleon 7.10k dataset (visualized in Figure 4.4). The results for other datasets were similar.

Comparison with Baseline Algorithm: Recall that a baseline approach for clustering over LBS data is to obtain a uniform sample using prior work such as [47] and run DBSCAN over the sample. Given a query point, we assign it to the nearest cluster. If there are no cluster within a distance threshold, it is categorized as a noise. We compared the clustering quality of baseline algorithm and HDBSCAN with same query budget while varying the budget from 200 to 600. Figure 4.18 shows the results for Chameleon t-7.10k dataset. Hilbert Curve is used for 2D to 1D mapping. As expected, HDBSCAN always outperforms baseline as the baseline often partitions the original clusters into many small clusters. Even the addition of a post processing step where we merge nearby small clusters does not improve the quality. Hence it is not a viable approach for tight query budgets.

Figure 4.19 shows how the cluster quality varies when the three most popular space filling curves - Hilbert (HC), Peano (PC) and Z-curves (ZC) were used with a query budget of 600. Hilbert and Peano curves provide best cluster qualities. Hence, for the rest of the experiments we only consider HC and PC. Since our work is a best-effort implementation of DBSCAN over LBS, we treat the output of DBSCAN as ground truth. A cluster quality of 1.0 is obtained when the cluster assignments of HDBSCAN and DBSCAN are identical.

Cluster Quality versus Query Cost: In this experiment, we evaluate how the quality of clusters discovered by HDBSCAN is impacted when the query budget is varied. We vary query budget from 200 to 600 and Figure 4.20 shows the result. As expected, the cluster quality improves with higher query budget. Nevertheless, even for a query budget as small as 200, the Rand index of HDBSCAN is 0.9. Recall that Rand index measures the percentage of cluster assignments that are correct which highlights a 90% accuracy of our algorithm.

Varying k: The value of k has a substantial impact on query cost. When the value of k is higher than *minPts*, the additional results retrieved could be used to infer the density of neighboring grid cells. Figure 4.21 shows the result of experiments when the value of k is

varied. As expected, the query cost is reduced when the value of k is increased. The impact of k on cluster quality is minimal.

Varying Minimum Grid Size: Next we demonstrate the effect of minimum grid size on query cost (Figure 4.22) and cluster quality (Figure 4.23). Intuitively, a large grid size reduces the total number of grids which in turn reduces the query cost. However, the grid size has a substantial impact on cluster quality. The relation between minimum grid size and cluster quality is not monotonic. If we set the minimum grid size too small the algorithm might partition the actual clusters into many smaller clusters. On the other hand, setting the grid size too large might merge the neighbor clusters in actual partitioning. In practice, our approach of adaptive grid sizes provides good results.

Query Cost versus Node Partition Count: At each level, standard Hilbert Curve partitions the existing cells into four equal size regions. However, in the adaptive space-filling curve approach we only partition a grid if required. We can divide the total query cost into two categories: i) Node splitting cost - queries that are issued on large grids that are partitioned later. ii) Leaf node query cost - queries executed at the leaf nodes of the tree. To reduce the node splitting cost, we can increase the number of children a node can be partitioned into. However, increasing this value may also increase the leaf node query cost. Figure 4.24 illustrates how varying the number of partitions of a node impacts the query cost. When the number of partitions per node is small, the total query cost is high due to the higher node splitting cost. When number of partitions is increased, the leaf node query cost becomes higher. The optimal node partition count depends on the data distribution. When the distance between the clusters are large, small node partition count is better, whereas the opposite is true when clusters are close to each other.

Query Cost versus Noise points (%): In the final experiment, we investigate the impact of noise points on query cost. We varied the noise point count by randomly selecting a portion of noise points from the dataset. As expected, query cost increases with increase in

the percentage of noise points. One of the major reasons for the increase in query cost is the impact the noise points have on the adaptive space filling curves. Without a well chosen threshold, the adaptive SFC algorithm could treat a sparse region (that would not have been explored if there is no noise) as a non-sparse region resulting in higher query cost.

Figure 4.26: Clustering of Places in DC using Yahoo Flickr geotagged photo

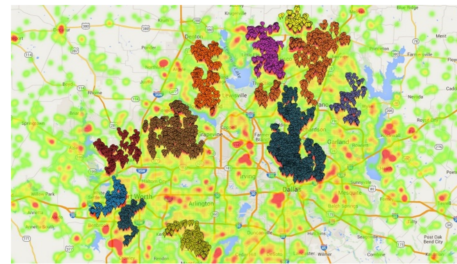


Figure 4.27: Clustering of houses in DFW area using Redfin

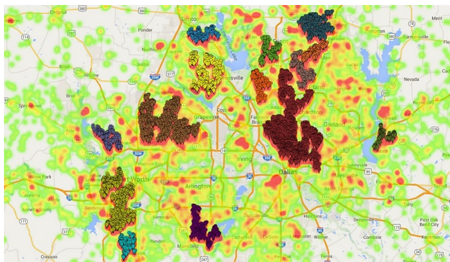


Figure 4.28: Clustering of houses in DFW area on Zillow dataset

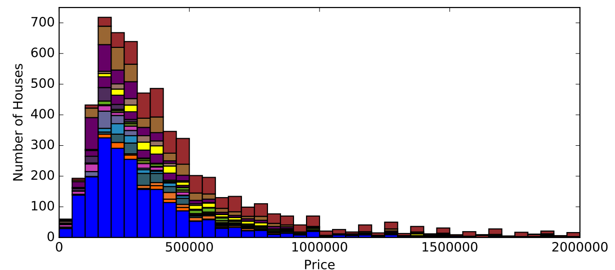


Figure 4.29: Price distribution of house for sales in DFW area

4.6.3 Experiments over Real-World LBS

Yahoo Flickr: For the Yahoo Flickr dataset, we run HDBSCAN on almost 118K images geotagged with Washington, DC area. The output clusters are plotted over the heat map of photos in Washington, DC. Figure 4.26 shows the comparison of discovered clusters with actual photo distribution. We can see that the clusters are located at the popular tourist

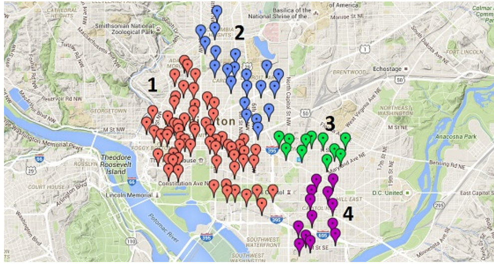


Figure 4.30: Clustering of Capital Bikeshare stations in DC

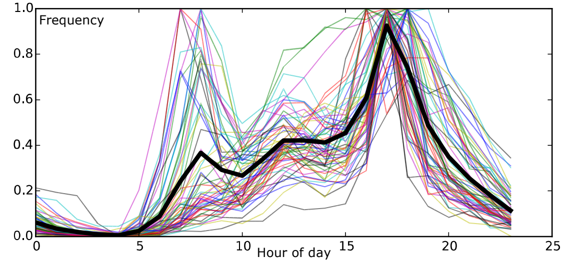


Figure 4.31: Average Rent per hour distribution for Cluster 1

spots of Washington, DC. This was also corroborated by the analysis of most frequent user tags from image metadata within each cluster. We used the population count information of US census data as external knowledge for this experiment.

Zillow: The results of running HDBSCAN over houses listed in DFW of Zillow website can be found in Figure 4.28. In order to compare the discovered clusters with actual distribution of "Houses for Sale" in DFW area, we plot the houses inside each cluster over the heat map of houses in Zillow dataset. Only clusters containing at-least 100 houses are shown. We can see that HDBSCAN discovers clusters of different shapes that matches the actual distribution of house in DFW. Since we have access to full dataset, we also performed basic statistical analysis of the house prices in discovered clusters. As expected, clusters near Dallas and Forth-Worth downtown have higher average price compared to others. The price distribution of houses in DFW area is shown as histogram in Figure 4.29. For each price bin, the corresponding count in each cluster is shown with different color. Count of houses in clusters of size smaller than 100 are combined altogether and shown with blue bar. We can see that clusters near urban areas are at the right side of the histogram and clusters in rural area are placed the left side. We used the "Housing Units" information of US Census data as external knowledge for optimization [17].

Redfin: We also performed clustering of houses in DFW area listed in Redfin website. This was an online experiment conducted live over Redfin. The output clusters are shown

in Figure 4.27. The list of identified clusters are similar to those identified from Zillow dataset. The minor differences are due to the fact that some houses were not listed in both services.

Capital Bikeshare: We used Google Places API to perform clustering on Capital Bikeshare (CB) stations in Washington D.C area. Since Google Maps does not have complete information about all the CB bike stations in D.C., we augmented the clustering results by adding the missing bike stations to its nearest cluster discovered using Google Place API. Figure 4.30 shows the clusters identified by HDBSCAN. There is one big cluster at the center of DC (shown in red marker) and three smaller clusters at the outer side area(shown in blue and green markers). We validated the identified clusters from the historical rental data for each bike stations[2]. Figure 4.31 displays the distribution of average rent count (normalized to 1) of each bike stations for Cluster 1 (red markers). We can see that stations in cluster 1 has a different hourly rental distribution from other stations. Specifically, we can see that there is a difference in the usage pattern among stations in CB. Bike stations close to the center of the city are more frequently rented at the evening time compared to other times of the day. Whereas stations located outside have two peaks in the morning and evening. Analysis of other clusters showed similar distinct usage pattern.

4.7 Related Work

There has been extensive work on spatial data mining of which clustering is a major technique. Our problem is substantially different from traditional clustering problems as they have access to entire database. To the best our knowledge, our work is the first to propose enabling DBSCAN (or any clustering algorithm) over a LBS. We do not challenge or improve the cluster definitions over traditional databases, but rather enable the algorithmic

[2]https://www.capitalbikeshare.com/trip-history-data

design according to a given cluster definition over an LBS with a limited kNN interface. Please refer to [51] for a detailed survey of clustering algorithms. Prior work on finding cluster boundaries [48, 49] do not apply here as they also require access to entire database.

Due to the increasing popularity of LBS, key problems such as sampling [13, 24, 25, 47], aggregate estimation [13, 47] and crawling [53] over a restricted query interface such as kNN have been recently studied.

SFC has been studied in the context of multi-dimensional indexing where the multi-dimensional data is mapped to one dimension, enabling simple and well understood one-dimensional indexing algorithms to be utilized [3, 4]. Theoretical analysis of the clustering properties of SFC has been studied in [2–4, 55].

4.8 Final Remarks

In this paper, we explore the problem of enabling DBSCAN over an LBS with only a limited, kNN query interface. We developed HDBSCAN that uses an adaptive space-filling curve algorithm as a subroutine to map 2D data to 1D, clusters the 1D data and then merges the resulting 1D clusters into eventual 2D clusters. We verified the effectiveness of our algorithms by conducting comprehensive experiments on benchmark datasets and multiple real-world LBS.

CHAPTER 5

Privacy Implications of Database Ranking

In recent years, there has been much research in the adoption of Ranked Retrieval model (in addition to the Boolean retrieval model) in structured databases, especially those in a client-server environment (e.g., web databases). With this model, a search query returns top-k tuples according to not just exact matches of selection conditions, but a suitable ranking function. While much research has gone into the design of ranking functions and the efficient processing of top-k queries, this paper studies a novel problem on the *privacy implications* of database ranking.

The motivation is a novel yet serious privacy leakage we found on real-world web databases which is caused by the ranking function design. Many such databases feature private attributes - e.g., a social network allows users to specify certain attributes as only visible to him/herself, but not to others. While these websites generally respect the privacy settings by not directly displaying private attribute values in search query answers, many of them nevertheless take into account such private attributes in the ranking function design. The conventional belief might be that tuple ranks alone are not enough to reveal the private attribute values. Our investigation, however, shows that this is not the case in reality.

To address the problem, we introduce a taxonomy of the problem space with two dimensions, (1) the type of query interface and (2) the capability of adversaries. For each

subspace, we develop a novel technique which either guarantees the successful inference of private attributes, or does so for a significant portion of real-world tuples. We demonstrate the effectiveness and efficiency of our techniques through theoretical analysis, extensive experiments over real-world datasets, as well as successful online attacks over websites with tens to hundreds of millions of users - e.g., Amazon Goodreads and Renren.com.

5.1 Introduction

5.1.1 Motivation

While traditional structured databases generally support the Boolean Retrieval model (i.e., return all tuples that exactly match the search query selection condition), in recent years there has been much research into exploring the applicability of an alternate Ranked Retrieval model (e.g., a kNN interface that returns top-k tuples according to a suitable ranking function). The ranked retrieval model has become an important component of many databases, especially in a client-server environment (e.g., web databases, where a client specifies and sends queries via a web interface to a backend database). Prior research has primarily focused on the effective design of ranking functions and the efficient processing of top-k queries for a given ranking function (e.g., [6, 56, 57]).

However, in this paper [58] we investigate a novel problem on the *privacy implications* of database ranking, which has not been studied before. We show how privacy leakage (through the top-k interface) can be caused by a seemingly innocent design of the ranking function in such ranked retrieval models.

To understand how the privacy leakage occurs, note that many databases in a client-server environment feature both public and *private* attributes. For example, social networking websites often allow users to specify privacy settings that hide certain attributes from the public's view, e.g., profile demographics such as race, gender, income; location; past

posts, etc. These websites honor the privacy settings by omitting the private attributes from being displayed in the returned query answers. Thus, the results include a ranked list of k tuples, but with only the public attributes displayed, and the private attributes hidden.

The problem here, however, is that many websites indeed include these private attributes as *input* to the ranking function. The purpose of doing so is, understandably, to make ranking more effective - e.g., the friend-search feature in a social network would preferably return users that have similar demographics or behavior patterns (e.g., posting with similar frequencies) as the user who executes the search, as common-sense indicates that they are more likely to be interested in each other. From the privacy perspective, this design might look harmless as well - after all, while a ranking function might take as input a large number of attributes, its output is merely the (relative) rank of a tuple among returned results - not even the actual ranking score! Naturally, the traditional belief here is that it is impossible to infer private attribute values from just the ranking of a returned tuple.

In our investigation of real-world client-server databases (including popular web databases), we found this traditional belief to be *wrong*. Specifically, in this paper, we develop a novel technique which, by asking a carefully constructed sequence of top-k queries and observing the corresponding change of tuple ranks in the query answers, may successfully infer the value of private attributes.

Before introducing our technical results, we would like to first illustrate the real-world impact of this privacy leakage by briefly demonstrating a very simple attack one can deploy using this technique on Renren.com, the equivalent of Facebook in China which has hundreds of millions of users. We chose this website as an example not only because of its large user base, but because it supports extensive privacy settings - allowing a user to specify as private any subset of profile attributes such as hometown, work affiliation, university attended, etc. It also respects these privacy settings in the display of search results - e.g., if a user specifies hometown as private and "only visible to friends", then

the user's hometown information will be hidden from all search and/or recommendation results unless the query is issued by a friend of the user.

Nevertheless, we also found that when ranking users in search or recommendation results, the ranking function used by Renren.com takes into account *all* attributes of a user's profile, regardless of whether a user has specified it to be private and/or who is issuing the query. For example, Figure 5.1a shows the screenshot[1] of the ranked list of tuples (i.e., users) returned for a friend-search query issued by a user LIONEL with hometown = Beijing, China and no other profile attribute specified. The query is formed using the only public attribute of our victim user TARGET (with a red target icon in the screenshot), name = Jia Ming. Since TARGET sets his hometown to be a private attribute "only visible to friends" and LIONEL is not a friend of TARGET, the hometown of TARGET is hidden from display in the query answer. Figure 5.1b shows the answer to the exact same query after LIONEL changes his hometown to Shanghai, China. The rank of TARGET now moves up from No. 3 to No. 1 in the new answer - and indeed ranks even higher than a few other users with the same name from Shanghai and studying in Fudan university (in Shanghai). The change of rank indicates a strong likelihood of TARGET having hometown = Shanghai (even though it does not form a proof). In this paper, we shall show how one can indeed prove that TARGET must come from Shanghai using just a few other query answers.

Figure 5.1: Demonstration of an attack over Renren

[1]Note that, since Renren.com does not have an English version, this screenshot is taken in Chrome with the automated webpage translation feature of Google Translations enabled.

5.1.2 Novel Problem: Rank-Based Inference

The above motivating example led us to identify an important and novel problem of *ranked-based inference of private attributes*. From a conceptual standpoint, this problem is interesting as, to the best of our knowledge, privacy compromise from *tuple ranks* has not been studied before. From a practical standpoint, this problem is important as many client-server databases, especially web databases that attract large amounts of user contributions, commonly offer top-k query interfaces yet contain sensitive data (e.g., profiles, demographics) that users would like to keep private.

We formalize the problem as follows. Consider a database D with n tuples and $m + m'$ attributes, m of which $A_1, \ldots, A_m$ are public while the other m', $B_1, \ldots, B_m$, are private. The database allows top-k queries where k is a small number ($k \ll n$). To specify a query q, one assigns a predicate on each of the $m + m'$ attributes. The predicate can be point[2] (i.e., $A_i = v$) or range (e.g., $B_i \in \{v_1, v_2\}$ or $*$, i.e., the entire domain).

Given a top-k query q, the database computes a predetermined ranking function $s(t|q)$ for each tuple t in the database, and returns the k tuples with the smallest $s(t|q)$. Of course, only the m public attributes are displayed on the return interface - not the private attributes or the ranking score. In most websites, the ranking function is a closely guarded secret - so we assume the adversary has no knowledge of the ranking function other than two very basic properties, *monotonicity* and *additivity*, which we shall define in §5.2 and demonstrate that they hold for almost all reasonable ranking functions used in the real-world.

The objective of an adversary is to compromise the privacy of a pre-determined victim tuple v. Of course, the adversary can readily acquire the public attributes of v. Nonetheless, it does not know the ranking function being used (and of course no knowledge what-

[2]For the purpose of this paper, we consider all attributes to be discrete, which can be categorical or ordinal, and assume the proper discretization of numeric attributes.

soever of the tuples' ranking scores). Thus, the technical challenge for the adversary is to unveil the private attribute values, e.g., $v[B_1]$, by issuing a small number of queries through the web interface and observing only the public attribute values of the returned tuples and the order in which they are returned. Note that an important goal for the adversary is to keep the number of queries as small as possible, because almost all websites enforce a limit on the number of queries one can issue through the web interface for a given time period (e.g., from one IP address or one user account each day), in order to prevent overburdening its backend database or to thwart third-party crawling of its contents.

To the best of our knowledge, the above problem of inferencing sensitive data from the *ranking* of a tuple is very novel. While top-k querying has been extensively studied by the database community [6, 11, 57], much of the efforts were focused on (1) developing techniques to answer such queries efficiently [6, 59–61], and (2) designing proper distance/ranking functions for various applications [62–64]. There have been prior work on data privacy in the general area of *query inferencing* [65–67], but most focus was on learning individual values from aggregates such as SUM, MIN, MAX, etc. We discuss related work in more detail in §6.5.

5.1.3 Overview of Technical Results

As one of our important contributions, we introduce a comprehensive taxonomy of the problem space according to two dimensions: (1) the type of query interfaces widely used in practice and (2) the capability of adversaries. Then, for each subspace of the problem, we develop a novel technique which either guarantees the successful inference of private attributes, or (when such an inference is provably infeasible in the worst-case scenario) accomplishes the attack for a significant portion of real-world tuples.

Consider the first dimension. We distinguish between interfaces which only support "point queries" (i.e., a single value must be specified for each attribute in the query), and

those that also support "IN queries" (i.e., where a subset/range of values can be specified for an attribute). For the second dimension, we distinguish between two types of adversaries: (1) those who are "query-only" (Q-only adversaries) - i.e., they are *passive* adversaries who only issue queries and observe query answers, but never tamper with (e.g., insert fake tuples into) the database; and (2) adversaries who "query-and-insert" (Q&I-adversaries), i.e., they only issue queries but also insert fake tuples into the database (e.g., by registering for fictitious user accounts on a social media website). As we shall further elaborate in §5.3, while many web databases have no restriction on the registration of new accounts, others makes it difficult for users to create fictitious accounts - e.g., Catch22Dating [68], a vulnerable website we shall study in the experiments, manually authenticates the real-world identity of each new account; while the aforementioned Renren.com also manually checks and verifies all user name changes. In these cases, most adversaries would be Q-only, while those who have adequate resources to acquire multiple real-world identities can become Q&I.

We have carefully investigated the four problem subspaces arising out of this taxonomy, and developed four novel attacks: Q&I-Point, Q-Point, Q&I-IN, and Q-IN. The fundamental ideas behind these attacks include two critical reductions: One reduces the problem of compromising a private attribute to finding so-called *differential queries* (defined in §5.4.1) which exclude all but one values in the domain. The second further reduces the problem to just finding a query which returns the victim tuple - nevertheless, this reduction holds only for Q&I-adversaries.

Table 5.1: Feasibility, Worst- and Practical Query Cost

	Q&I-Point	Q-Point	Q&I-IN	Q-IN
Feasibility	Yes	Maybe	Yes	Maybe
Worst-case	$\prod_{i=1}^{m'} \lvert V_i^{\mathrm{B}} \rvert$	N/A	$\prod_{i=1}^{m'} \lvert V_i^{\mathrm{B}} \rvert$	N/A
In Practice	High	Highest	Lowest	Low

Note: $|V_i^{\mathrm{B}}|$ is the domain size for private attribute B_i.

The differences on the applicability of these reductions lead to fundamentally different feasibilities of the attack, as illustrated in Table 1. Specifically, we find that while Q&I adversaries are always able to accomplish the attack, there are cases where Q-only ones will fail. In terms of query cost, while the worst-case cost for even Q&I adversaries can be exponential, the query cost in practice is very reasonable - and can be significantly reduced when IN queries are available, even though IN has no impact on the (theoretical) worst-case query cost.

In summary, we make the following contributions in this paper:

- We have identified a novel and important problem of rank-based inferencing over web databases.
- We introduce a comprehensive taxonomy of the problem space, and identify four important subspaces based on varying database interface limitations and adversarial capabilities.
- For each problem subspace, we developed nontrivial adversaries, and carried out a rigorous theoretical analysis of their performance. Our results show that in almost all cases, the adversaries can launch efficient and successful attacks.
- We performed extensive experiments over real-world datasets, with results corroborating well with our theoretical findings. We also conducted successful online experiments over real-world websites including the aforementioned social network Renren.com as well as other types of web databases such as Amazon Goodreads and Catch22Dating.

5.2 Preliminaries

5.2.1 Model of Web Databases

As discussed in the introduction, many web databases store both public and private attributes of a user. Consider an n-tuple (i.e., n-user) database D with a total of $m + m'$ attributes, including m public ones $A_1, \ldots, A_m$ and m' private ones $B_1, \ldots, B_{m'}$. Let V_i^{A} and V_j^{B} be the attribute domain (i.e., set of all attribute values) for A_i and and B_j, respectively. For the purpose of this paper, we consider V_i^{A} and V_j^{B} to be discrete and publicly known, and leave studies of numeric/infinite/unknown domains to future research.

We use $t[A_i]$ (resp. $t[B_j]$) to denote the value of a tuple $t \in D$ on attributes A_i (resp. B_j). For the purpose of this paper, we assume there is no duplicate tuple in the database (before an adversary makes any modification to the database) - i.e., every *bona fide* tuple has a unique value combination for the $m + m'$ attributes. While we assume that D does not change during the course of an attack, we include discussions in §5.4.2.1 to address the scenario where this assumption is violated.

Recall from the introduction that the database allows top-k queries where k is a small number such as 10 or 50. Given a *supported query* q defined below, the database computes the *ranking function* $s(t|q)$ for each tuple $t \in D$, and selects/returns k tuples in the *ascending* order of $s(t|q)$ (i.e., only the k tuples with minimum $s(t|q)$ will be returned). Of course, only the public attribute values, i.e., $t[A_1], \ldots, t[A_m]$, will be returned for each of the k tuples. Of course, since we allow duplicates on public attribute values - i.e., multiple tuples might share the same vaule combination on $A_1, \ldots, A_m$ - there must be a way to distinguish different returned tuples with the same public-attribute value-combination. For this purpose, we assume each tuple to be returned alongside a unique identifier (e.g., user ID) - and the adversary knows the unique identifier of the victim tuple as prior knowledge.

It is important to note that the ranking score $s(t|q)$ is *not* returned - in addition, the design of $s(\cdot|\cdot)$ itself is a secret kept by the database owner.

Supported Queries: For the purpose of this paper, we consider ranking functions/queries that take into account both public and private attribute information. In other words, the web database supports queries which specify values/conditions on some or all of the $m + m'$ (public and private) attributes. Consider friend recommendation on a social media website as an example - when the website uses private information of a user (say `education`) while generating the recommendations, it is indeed answering a query that contains a predicate on private attribute `education` - with the ranking function likely taking into account whether a tuple's value on `education` is equal to that specified in the query.

In this paper, we consider two types of predicates that can be specified on an attribute: *point* and *IN*. Let the predicate specified in a query q for attribute A_i (resp. B_i) be $q[A_i]$ (resp. $q[B_i]$). A point predicate assigns a single value in the domain, i.e., $q[A_i] \in V_i^{\mathrm{A}}$, while an IN predicate assigns a subset of values, i.e., $q[A_i] \subseteq V_i^{\mathrm{A}}$. Consider a dating website as an example. While gender is often specified as a point predicate (i.e., male or female), interests and age can be considered IN ones (i.e., find users who most closely match the interest set {reading, travel, cycling, cooking} or age range $[25, 30]$). A special example of IN predicate is $q[A_i] = V_i^{\mathrm{A}}$ - i.e., $q[A_i] = *$ - indicating "do-not-care" on an attribute.

Practical Constraints: Most, if not all, web databases enforce practical constraints on how one might interact with the web interface. The two most important constraints here are *query-rate limitation* and *tuple insertion constraint*.

Most web databases enforce certain query-rate limits, i.e., limits on the number of queries one can issue (e.g., from an IP address or a user account) per time period (e.g., each day), in order to prevent overburdening of the backend database and/or third-party

crawling of its contents. Hence an adversary must aim to minimize the query cost of a rank-based inference attack, as otherwise it would have to acquire more resources (e.g., more IP addresses, registering more accounts) in order to issue all queries required by the attack.

Tuple insertion constraint, on the other hand, refers to ones ability to *insert* tuples into the database. Some web databases, including many online social networks, do not enforce this constraint - i.e., one can freely insert new tuples (i.e., user accounts) to the database by registering for new accounts (e.g., using a new email address). Nonetheless, there are also others that require users' real identities and use offline authentication to check them. For example, catch22dating, a popular online dating website used in our real-world experiments, requires each user to have an authenticated identity as student of selected universities. For these databases, inserting new/fake tuples becomes extremely difficult, if not impossible. We say that the web database enforces a tuple insertion constraint which prevents an adversary from inserting arbitrary tuples.

5.2.2 Properties of Ranking Function

There has been significant research in database ranking (e.g., [6,60,69]) which studies the design of ranking function $s(t|q)$, including in cases where the query has IN predicates (e.g., [6, 70]). While this paper aims to study *generic* rank-based inferences that work for a broad class of ranking functions, it is important to note that *no* attack will work without assuming certain properties of the ranking function. To understand why, consider a simple example where $s(t|q)$ is generated uniformly at random from $[1, n]$. Since the rank of a tuple has nothing to do with the tuple's (private) attribute values, no adversary can compromise any private information from the returned ranks. Thus, it is the objective of this subsection to define a minimum set of conditions that are satisfied by most if not all

ranking functions used in practice. Specifically, we consider *monotonicity* and *additivity*, respectively as follows.

Monotonicity Condition: Intuitively, the monotonicity condition simply states that, for a given query, the relative rank between two tuples which differ only on one attribute should be determined by that attribute alone. Formally, for a point-query interface, if two tuples t and t' differ only on A_i and $t[A_i] = q[A_i]$, then t should have a smaller distance to q than t'. More generally, we have the following definition. Note that in this definition, we consider $q[A_i]$ (resp. $q[B_j]$) to be a set (in the case of point-query, containing just a single value) without introducing ambiguity.

Monotonicity: $\forall q, t \in D$, and $i \in [1, m]$ (resp. $j \in [1, m']$), if t and t' share the same value on all attributes except A_i (resp. B_j) and $t[A_i] \in q[A_i]$ while $t'[A_i] \notin q[A_i]$ (resp. $t[B_j] \in q[B_j]$, $t'[B_j] \notin q[B_j]$), there must be $s(t|q) < s(t'|q)$.

Additivity Condition: Intuitively, the additivity condition states that, for two tuples t and t', if t is already ranked higher than t' in query q, then further changing the predicate of q on A_i (resp. B_j) to exactly match t - i.e., making $q[A_i] = t[A_i]$ (resp. $q[B_j] = t[B_j]$) - should not change the relative rank between the two tuples. More formally, we have the following definition:

Additivity: $\forall q$ and $t, t' \in D$, if $s(t|q) < s(t'|q)$, then there must be $s(t|q') < s(t'|q')$, where q' is the same as q on all but one attribute A_i (resp. B_j), on which $q[A_i] = t[A_i]$ (resp. $q[B_j] = t[B_j]$).

One can see that both monotonicity and additivity are common-sense conditions that should be reasonably expected of a ranking function. Our studies of real-world web databases (in §5.7) verified this observation, as all websites considered satisfy both conditions.

5.3 Problem Space

In this section, we define the rank-based inference problem studied in the paper. Specifically, we start with defining the objectives of an adversary. Then, we partition the entire problem space into four quadrants along two dimensions: the type of queries supported, and the type of operations an adversary can perform.

5.3.1 Adversary Model

The objective of an adversary is two-fold: *compromising privacy* and *minimizing query cost*. Privacy-wise, an adversary aims to compromise private attributes of a victim tuple v. Without loss of generality, we assume that the adversary aims to compromise the value of $v[B_1]$ based on prior knowledge of all public attributes of v, i.e., $v[A_1], \ldots, v[A_m]$. In §5.4, we shall address cases where an adversary aims to compromise all private attributes of v.

To ensure the versatility of our algorithms, we make a conservative assumption that an adversary has *no* prior knowledge of the ranking function other than the fact that it satisfies the monotonicity and additivity conditions defined above. Clearly, all algorithms in the paper still work if an adversary does know the ranking function. While it is possible that prior knowledge of certain ranking functions can enable more efficient attacks than those in the paper, we leave such ranking-function-specific studies to future work.

Given the query-rate limitation discussed in §5.2, an important goal of the adversary is to minimize the query cost for the attack, as otherwise the website-enforced limit on the number of queries from each user (e.g., IP-address) may stop the attack from being completed. To this end, it is important to note that our key efficiency measure here is the number of requests issued to the web database (hereafter referring to as the *query cost*, including both search queries and requests to insert tuples, if the database does not enforce

the aforementioned tuple insertion constraint) - while other measures such as local (CPU or I/O) processing overhead are secondary.

5.3.2 Two Dimensions

The first dimension we use for partitioning the problem space is the type of queries supported. There are two different cases: (1) Point-Query Interface which requires a point predicate defined in §5.2 to be specified for *every* attribute. An example here is the friend recommendation offered by many social media websites - each user has to complete his/her own profile to enable the feature, essentially requiring the user to specify a point predicate on every public and private attribute. (2) IN-Query Interface which supports *IN queries* over all attributes. Clearly, here a user can choose "do not care" for an attribute by assigning its entire value domain to the IN condition. Since point queries are special cases of IN, all queries supported by the point-query interface are also supported here.

The second dimension for partitioning the problem space is the adversary power. Specifically, we consider the following two cases:

- *Query-and-Insert (Q&I) Adversary* can not only issue queries but also *insert* tuples to the database. It can also update or delete any tuple it inserted. These adversaries exist for websites which do not enforce the tuple insertion constraint.
- *Query-only (Q-only) Adversary* can query the web database but cannot change it. This is the case when the website enforces the tuple insertion constraint (see §5.2).

One can see from the definitions that Q&I adversaries are stronger - i.e., any attack launched by a Q-only adversary can also be launched by a Q&I-one, while the opposite is not true. We shall show later in the paper that the ability to *insert* leads to significant differences on the outcome of a rank-based inference attack. Specifically, while a Q&I adversary can *always* accomplish the attack even in the worst-case scenario, the same is not true for Q-only adversaries.

5.3.3 Problem Definition

Given the two dimensions, we partition the problem space into four quadrants: (1) point query interface with Q&I adversaries, (2) point query with Q-only, (3) IN with Q&I, and (4) IN with Q-only.

Problem Definition (Rank-Based Inference): Given a database D *and a victim tuple* $v \in D$*, find the shortest sequence of queries* $q_1, \ldots, q_c$ *supported by the interface and a corresponding sequence of tuple sets* $T_1, \ldots, T_c$*, such that*

$$\delta(q_1(D \cup T_1), q_2(D \cup T_2), \ldots, q_c(D \cup T_c)) = v[B_1]. \tag{5.1}$$

where $q_i(D \cup T_i)$ *is the answer to* q_i *over the* $D \cup T_i$ *and* $\delta(\cdot)$ *is a (deterministic) function for rank-based inferencing. For a Q-only adversary, there must be* $T_1 = \cdots = T_c = \emptyset$*.*

Naturally, the problem could be extended to infer multiple, if not all, private attributes of victim tuple v. In fact, as we shall describe in §5.4, our algorithm iteratively learns private attribute values one at a time till $v[B_1]$ is inferred. Extending it to infer all attributes is trivial. To better illustrate our ideas and to significantly simplify the notations, in the theoretical discussions in this paper, we focus on the case where $k = 1$ (note that $k = 1$ is actually a conservative worst-case assumption for the attack design), and discuss the straightforward extension to larger k in the experiments section.

Running example of ranking function: All algorithms developed in this paper work for any ranking function satisfying monotonicity and additivity - so does all complexity and lower bound analysis. Nonetheless, when studying the practical performance of attacks and illustrating how different ranking-function designs affect attack effectiveness, it is necessary to consider certain concrete ranking function designs - for this purpose only, we consider the following linear ranking function as a running example:

$$s(t|q) = \sum_{i=1}^{m} w_i^{\mathrm{A}} \cdot \rho(q[A_i], t[A_i]) + \sum_{i=1}^{m'} w_i^{\mathrm{B}} \cdot \rho(q[B_i], t[B_i]), \tag{5.2}$$

where $w_i^{\mathrm{A}}, w_i^{\mathrm{B}} \in (0, 1]$ are the *ranking weight* for attribute A_i and B_i, respectively. The distance measure for each attribute, i.e., $\rho(q[A_i], t[A_i])$, is a variation of the discrete metric: (1) $\rho(q[A_i], t[A_i]) = 0$ if $t[A_i] \in q[A_i]$ (note that for point queries, this means $t[A_i]$ being equal to the single value in $q[A_i]$), and (2) $\rho(q[A_i], t[A_i]) = 1$ if $t[A_i] \notin q[A_i]$.

Once again, we would like to note that the adversary has *no* knowledge of the ranking function design whatsoever (other than its monotonicity and additivity). This linear ranking function based running example merely provides a concrete basis for the analysis of attack performance in practice.

5.4 Point Query Interface

We start by considering a point query interface. Specifically, we shall start with reducing rank-based inference to the problem of finding pairs of *differential queries* based on the victim tuple v. Then, we discuss the design of Q&I-Point and Q-Point, our rank-based inference algorithms for Q&I and Q-only adversaries over a point query interface, respectively.

5.4.1 Goal: Finding Differential Queries

We start by showing that, for the worst-case scenario of $k = 1$, the problem of compromising the private attribute B_1 of victim tuple v can be reduced to finding for each possible value of B_1 except $v[B_1]$, i.e., $\forall\theta \in (V_1^{\mathrm{B}} \backslash v[B_1])$, a pair of *differential queries* q_θ and q'_θ which satisfy three properties: (1) they share the same predicate on all attributes but

B_1, (2) $q'_\theta[B_1] = \theta$ while $q_\theta[B_1] \neq \theta$, and (3) q_θ returns the victim tuple v while q'_θ does not - i.e.,

$$\forall t \in D \text{ where } t \neq v, s(t|q_\theta) > s(v|q_\theta),$$
$$\exists t \in D \text{ with } t \neq v \text{ such that } s(t|q'_\theta) < s(v|q'_\theta). \quad (5.3)$$

The proof of this reduction is straightforward: Due to the additivity condition, we can infer from (5.3) that the value of victim tuple v on B_1 must *not* be the same as $q'_\theta[B_1]$, i.e., $v[B_1] \neq \theta$. Since we found differential queries q_θ, q'_θ for all $\theta \in V_1^{\mathrm{B}} \backslash v[B_1]$, the only remaining possibility is the correct value of $v[B_1]$.

While this reduction is the basis of our following discussions, it is important to note that reduction in the opposite direction does not hold - i.e., an adversary does *not* have to find all $|V_1^{\mathrm{B}}| - 1$ pairs of differential queries in order to compromise $v[B_1]$. To understand why, consider an example where a Q&I-adversary inserts into the database a dummy tuple t with value 0 on all public and private attributes. Then, upon issuing a query with $q[A_i] = 0$ and $q[B_i] = 0$ on all attributes, the adversary receives v rather than t as the No. 1 result. One can see that the adversary can safely infer $v[B_1] = 0$ without issuing any additional query or identifying the differential queries for any value of B_1.

5.4.2 Q&I adversary

We develop Algorithm Q&I-Point in this subsection. Specifically, we start with a somewhat surprising finding - for a Q&I adversary, as long as it has the ability to find a query that returns the victim tuple v for a given database, then it can *always* successfully compromise $v[B_1]$ (by finding differential queries for all other values in V_1^{B}). Then, we present Algorithm Q&I-Point and analyze its worst- and average-case query costs.

5.4.2.1 Reduction to finding a query that returns v

Algorithm Q&I-Point: To construct the reduction to finding one query which returns the victim tuple, we start by assuming an oracle FIND-Q which, upon given input of a database D and the victim tuple v, returns a query q which returns v. We first develop Algorithm Q&I-Point which calls upon this oracle FIND-Q to compromise $v[B_1]$, and then introduce the design of FIND-Q afterwards. The pseudocode of Q&I-Point is shown in Algorithm 10.

For the ease of understanding, we introduce some simple notations: We represent the domain of every attribute as 0, 1, ..., $|V|-1$, where $|V|$ is the domain size of the attribute. Let there be $v[A_1] = \cdots = v[A_m] = 0$. Without loss of generality, we assume the output of FIND-Q to always have $q[A_1] = \cdots = q[A_m] = 0$. The reason here is simple - if q differs from v on any public attribute, we can always change the attribute to 0 - the new query will still return v due to the additivity condition of the ranking function.

We start by inserting into the database a tuple t that has all attributes equal to 0. Then, we call FIND-Q over the new database to discover q which returns v. Note that if FIND-Q fails to do so - i.e., no query over the database returns v - then we already succeed because, due to the no-duplicate assumption, the only scenario for this to happen is when $v = t$. Given the result q of FIND-Q, we note that q must differ from t on at least one private attribute - again, if $q = t$ and yet returns v, there must be $v = t$. Without loss of generality, suppose that q differs from t by having value 1 on private attributes $B'_1, \ldots, B'_h$ - i.e., $\forall i \in [1, h]$, $q[B'_i] = 1$.

We now construct $h + 1$ queries $q_0, \ldots, q_h$ as follows: all these $h + 1$ queries share the exact same value (i.e., 0) as t on all attributes but $B'_1, \ldots, B'_h$. For those h attributes, we assign to query q_i ($i \in [0, h]$) $q_i[B'_j] = q[B'_j] = 1$ if $j \leq h - i$ and $q_i[B'_j] = t[B'_j] = 0$ otherwise (i.e., if $j > h - i$). The following table shows an example. Note at the two extremes $q_0 = q$ and $q_h = t$.

	$A_1, \ldots, A_m$	B'_1	B'_2	B'_3	B_{others}
q	0	1	1	1	0
q_0	0	1	1	1	0
q_1	0	1	1	0	0
q_2	0	1	0	0	0
q_3	0	0	0	0	0
t	0	0	0	0	0

There are two important observations from the above query sequence: First, unless $v = t$, queries at the two ends must return different results - specifically, q_0 returns v while q_h returns t. The only exception here is when q_h also returns v - but this must mean $v = t$ because q_h exactly matches t - leading to an immediate compromise of $v[B_1]$. Second, every pair of adjacent queries in the sequence differ by exactly one attribute - i.e., query q_i and q_{i+1} differ on B'_{h-i}. Combining two observations, we know two things: (1) there must exist a pair of adjacent queries q_i and q_{i+1} such that q_i returns v while q_{i+1} does not - because otherwise all $h + 1$ queries would return v, contradicting Observation 1. (2) this pair of adjacent queries differ on exactly one attribute B'_{h-i}. In other words, they serve as a pair of *differential queries* for value 0 in the domain of B'_{h-i}, and prove $v[B'_{h-i}] \neq 0$. Note that the process of finding this pair of differential queries takes at most $h \leq m'$ queries.

Of course, this may not yet achieve the adversarial goal of compromising $v[B_1]$. Nonetheless, note that once we know $v[B'_{h-i}] \neq 0$, we can insert into the database a new t which replaces its value on B'_{h-i} with another value (other than 0) in its domain. We can then repeat the exact same process and get one of only two possible outcomes: either (1) we find another pair of differential queries and exclude from consideration a(other) value for one of the private attributes; or (2) an anomaly occurs - either FIND-Q cannot find q or q^h returns v instead of t - meaning $t = v$ and we have compromised $v[B_1]$ already.

One can see that, the worst-case scenario here is for us to repeat the process for $\sum_{i=1}^{m'}(|V_i^{\text{B}}| - 1)$ times - more repetitions is impossible because we would have already

excluded all wrong values for $B_1, \ldots, B_{m'}$. Throughout all repetitions, the number of queries issued by Algorithm Q&I-Point (excluding those required by FIND-Q) is $O(m' \cdot \sum_{i=1}^{m'} |V_i^{\mathrm{B}}|)$.

Practical Implications: We now discuss the practical implications of Algorithm Q&I-Point. First, while we shall address the design and theoretical bounds of FIND-Q in detail next, we would like to first point out here that, in practice, FIND-Q is usually a straightforward and efficient process, especially when there are many public attributes. The reason is simple: those public attributes alone are often sufficient to uniquely identify the victim tuple. Since FIND-Q knows $v[A_1], \ldots, v[A_m]$, it largely just needs to avoid hitting the few "fake" tuples Algorithm Q&I-Point inserts (by avoiding their private attribute values) in order to find a query that returns v.

The cost of FIND-Q aside, there are three interesting observations we can make regarding Algorithm Q&I-Point. First, its query cost depends on the SUM (not product) of domain size of private attributes. This works to the attacker's advantage in practice as real-world websites often feature only a few private attributes with small domains[3]. Nonetheless, this also means that large-domain attributes such as ZIP code can be very costly to attack. Intuitively, this is caused by nature of the point query interface - as each query here "covers" only one of the many domain values.

The second observation we would like to make is the *anytime* nature of the algorithm. While our problem definition focuses on compromising $v[B_1]$, one can see from the design of Q&I-Point that it indeed learns all private attributes of v. Specifically, every iteration (costing at most m' queries) excludes one value from consideration for one of the private attributes. Thus, even if we interrupt the algorithm at anytime (say running out of query allowance by the database), we would still have learned substantial information about many

[3]e.g., in the case of dating website discussed in Section 5.7.3, all private attributes are Boolean - e.g., "whether a user is willing to consider matches of a different race."

private attributes. This anytime feature makes the algorithm particularly difficult to thwart in practice.

Third, note from the design of Q&I-Point that all queries it issues (including those by FIND-Q) must have public attribute values equal to those of v. This makes the algorithm fairly resilient against changes to the database during the course of an attack - because the only changes that would affect the execution of Q&I-Point are those that feature tuples with the exact same public-attribute value-combination as v - an extremely unlikely event in practical databases.

Last, while a comprehensive discussion of defense strategies is out of the scope of this paper, one can see from the design of Q&I-Point a basic idea of thwarting rank-based inference: Note that during the attack, an adversary makes t closer and closer to the victim tuple v, in order to "isolate" the rank difference to a private attribute and infer its value. Thus, intuitively, the defense should try to detect and/or prevent the insertion of tuples that are too "close" to an existing tuple, as such insertions likely signal the Q&I-Point attack rather than a bona fide new tuple (which is highly unlikely to be almost identical to an existing tuple).

5.4.2.2 Query Cost Analysis

Algorithm FIND-Q: We now describe the algorithm for finding a query q that returns v for a given database D. Algorithm 11 shows the pseudocode for FIND-Q. The design is mostly straightforward - we randomly generate and issue a query q with $q[A_i] = v[A_i]$ for all $i \in [1, m]$ and each $q[B_j]$ ($j \in [1, m']$) drawn i.i.d. uniformly at random from V_j^{B} - and repeat this process until finding q that returns v. The only note of caution here is that the random generation is done *without replacement*, and *with memory* across different executions of FIND-Q. To understand why, note from the design of Algorithm Q&I-Point

Algorithm 10 Q&I-Point

1: **Input:** q, v **Output:** $v[B_1]$
2: $H_v = \varnothing; t[A_i] = v[A_i] \, \forall i \in [1, m]; t[B_j] = 0 \, \forall j \in [1, m']$
3: **while** $v[B_1]$ is not yet inferred **do**
4: Insert t into D
5: **if** q does not return v **then** $q \leftarrow$ FIND-Q(v, H_v)
6: **if** no such q, **then return** $t[B_1]$ // case: $t = v$
7: Let $B'_1 \dots B'_h$ be the attributes differing between t and q
8: $i = 0;$ $q_i = q$
9: **for** $i = 1$ to h **do**
10: $q_i = q_{i-1};$ $q_i[B'_{h-i+1}] = t[B'_{h-i+1}]$
11: **if** q_i and q_{i-1} return different tuples **then**
12: $q = q_{i-1};$ Set $t[B'_i]$ to an unexplored value
13: **break** for loop

that we only insert tuples into the database, and do not tamper with or delete the existing tuple values. Thus, any query which does not return v before cannot return v in the future - justifying the design.

One can see from the design of FIND-Q that it always succeeds. As such, our focus here is to consider its query cost. First, all calls of FIND-Q, altogether, consume a worst-case query cost of $O(\prod_{i=1}^{m'} |V_i^{\mathrm{B}}|)$. While this seems like an outrageously high cost, we make two interesting notes here: First, the worst-case scenario indeed requires these many queries - as proved by the following lower bound result which shows that the cost cannot be improved beyond a constant factor. Second, the real-world query cost for FIND-Q is likely much smaller, as demonstrated by an average-case example study at the end of this subsection.

Algorithm 11 FIND-Q

1: **Input:** Victim v, Query history H_v **Output:** q that returns v

2: Let $\mathcal{Q}$ be set of all possible point queries q with $q[A_i] = v[A_i] \quad \forall i \in [1, m]$

3: Find a query $q \in \mathcal{Q} \setminus H_v$ that returns v; Update H_v

4: **return** q if it exists else return failure

Lower Bound on Worst-Case Query Cost: The following theorem shows that, in the worst-case scenario, no algorithm can accomplish the attack without issuing $\Omega(\prod_{i=1}^{m'} |V_i^{\mathrm{B}}|)$ queries.

Theorem 4. *Given any ranking function and victim tuple v, there exists a database D such that no Q&I-adversary can compromise $v[B_1]$ without issuing $\Omega(\prod_{i=1}^{m'} |V_i^{\mathrm{B}}|)$ queries.*

Proof. First, we prove that, in order for an adversary to compromise $v[B_1]$, it must be able to find at least one query which returns v. Let the queries the adversary issues before inferring the value of $v[B_1]$ be $\epsilon_1, \ldots, \epsilon_x$. Note that in order for the inference to hold, the following condition must be true: if we change the value of $v[B_1]$ to $\theta \in V_1^{\mathrm{B}} \backslash v[B_1]$, then at least one of the queries $\epsilon_1, \ldots, \epsilon_x$ must have a different answer. To understand why, note that if all query answers the adversary received remain the same after the change, then there is no way for the adversary to always infer $v[B_1]$ correctly from the query answers, because any deterministic algorithm that takes the answers to $\epsilon_1, \ldots, \epsilon_x$ as input will output the exact same value when $v[B_1] = \theta$. Also, note that since the only change to the database is the value of v, the difference on query answer must be whether the query returns v or not. Thus, the adversary can always find at least one query which returns v.

We now construct a database which requires an expected number of $\Omega(\prod_{i=1}^{m'} |V_i^{\mathrm{B}}|)$ queries for finding one query which returns v. Let there be $\sum_{i=1}^{m'}(|V_i^{\mathrm{B}}| - 1)$ tuples in the database:

$$v_1^1, \ldots, v_1^{|V_1^{\mathrm{B}}|-1} \tag{5.4}$$

$$v_2^1, \ldots, v_2^{|V_2^{\mathrm{B}}|-1} \tag{5.5}$$

$$\ldots$$

$$v_{m'}^1, \ldots, v_{m'}^{|V_{m'}^{\mathrm{B}}|-1} \tag{5.6}$$

Specifically, for any $j \in [1, m']$, every v_j^i ($i \in [1, |V_j^{\mathrm{B}}| - 1]$) shares the same value as v on all attributes but B_j. In addition, each v_j^i takes a unique domain value in V_j^{B} that is different from $v[B_j]$. Due to the existence of these tuples, any query q which differs from v on at least one attribute will not return v - because, according to the monotonicity condition, there must exist at least one tuple in v_j^i with a smaller distance from q. Since the adversary was given prior knowledge of $v[A_1], \ldots, v[A_m]$ but no information about the private attribute values, the optimal adversarial strategy is to issue queries q with $q[A_i] = v[A_i]$ for all $i \in [1, m]$ and $q[B_i]$ chosen uniformly at random from V_i^{B}. One can see that, in this worst-case scenario, the expected query cost required for finding a query that returns v is $\Omega(\prod_{i=1}^{m'} |V_i^{\mathrm{B}}|)$. □

Running Example Query Cost: We now consider how Q&I-Point performs over the running example of a linear-combination ranking function in Equation 5.2 and a database where each tuple is generated i.i.d. randomly according to the uniform distribution, while the victim v is chosen uniformly from the database.

Theorem 5. *In the running example, the expected number of queries Q&I-Point issues to compromise $v[B_1]$ is at most $1/p + \sum_{i=1}^{m'}(|V_i^{\mathrm{B}}| - 1)$, where*

$$p = \prod_{t \in D, t \neq v} \left(\frac{1}{2} + \frac{1}{2} \cdot \mathrm{erf}\left(\frac{d^A(v,t)}{\sqrt{\sum_{i=1}^{m'} 2w_i'^2 \cdot \frac{|V_i^{\mathrm{B}}|-1}{|V_i^{\mathrm{B}}|^2}}} \right) \right) \tag{5.7}$$

where $\mathrm{erf}(\cdot)$ is the standard error function [71], and $d^A(v,t)$ is the distance between v and t on public attributes - i.e., $d^A(v,t) = w_1 \cdot \rho(v[A_1], t[A_1]) + \cdots + w_m \cdot \rho(v[A_m], t[A_m])$, where ρ is the distance function defined in the running example.

Proof. Note that q generated in the above-described random process has $d(q,v)$ following Multinomial distribution with mean μ and variance σ^2 as follows.

$$\mu = \sum_{i=1}^{m'} w_i' \cdot \frac{1}{|V_i^B|}; \;\; \sigma^2 = \sum_{i=1}^{m'} w_i'^2 \cdot \frac{|V_i^B - 1|}{|V_i^B|^2} \tag{5.8}$$

In addition, $\forall t \in D$, given $d^A(q,t)$, the overall distance $d(q,t)$ follows the Multinomial distribution with mean $\mu_0 = d^A(q,t) + \mu$, and the same variance σ^2 as above. Note that since q shares the same attribute values as v on all public attributes, we have $d^A(q,t) = d^A(v,t)$. As such, the probability for a query q with $q[A_i] = v[A_i]$ for all $i \in [1,m]$ and $q[B_i]$ chosen uniformly at random from V_i^B to return v is

$$p = \prod_{t \in D, t \neq v} \left(\frac{1}{2} + \frac{1}{2} \cdot \mathrm{erf}\left(\frac{d^A(v,t)}{\sqrt{\sum_{i=1}^{m'} 2w_i'^2 \cdot \frac{|V_i^{\mathrm{B}}|-1}{|V_i^{\mathrm{B}}|^2}}} \right) \right) \tag{5.9}$$

In other words, the expected number of queries the adversary needs to issue before finding a query that returns v is

$$\frac{1}{p} = \frac{1}{\prod_{t \in D, t \neq v} \left(\frac{1}{2} + \frac{1}{2} \cdot \mathrm{erf}\left(\frac{d^A(v,t)}{\sqrt{\sum_{i=1}^{m'} 2w_i'^2 \cdot \frac{|V_i^{\mathrm{B}}|-1}{|V_i^{\mathrm{B}}|^2}}} \right) \right)} \tag{5.10}$$

Of course, after finding a query that returns v, the adversary in the worst case has to issue an additional $\sum_{i=1}^{m'}(|V_i^{\mathrm{B}}|-1)$ queries (see proof of Theorem 4) for inferring the value of $v[B_1]$. $\square$

Note from (5.7) why the average-case query cost of FIND-Q (and thereby Q&I-Point) is likely much smaller than its worst-case bound: p is the probability for a query q randomly tested in FIND-Q to return v. One can see that, when there is a large number of public attributes or a small number of private ones - i.e., a larger $d^A(v,t)$ or a smaller w'_i, the probability for a tuple t ($t \neq v$) to "overcome" its difference with q on public attributes (with which v has zero difference) by private attribute values is fairly small - leading to a larger p and, ultimately, a smaller query cost.

The query cost required for Q&I-Point to compromise $v[B_1]$ actually *decreases* with a *smaller* weight on the private attributes. This observation seems counter-intuitive because when $w'_1 = 0$, no privacy disclosure occurs as the rank becomes independent of $v[B_1]$ - but the worst disclosure occurs when w'_1 takes the smallest positive value! To understand why, note that the smaller private ranking weights w'_i are, the easier it is for an adversary to pinpoint a query that returns v, as the adversary already has prior knowledge of all public attribute values of v. Given that, for a Q&I-adversary, finding a query returning v is (almost) equivalent with compromising $v[B_1]$, we have this seemingly counter-intuitive observation.

5.4.3 Q-only adversary

Design of Q-Point: For adversaries subject to the tuple-insertion constraint, the feasibility of compromising $v[B_1]$ is not of certainty as in the Q&I adversary case, as shown in the following theorem.

Theorem 6. *Given any victim tuple v, there exists a ranking function $s(\cdot|\cdot)$ and a database D such that no Q-only adversary can perform a rank-based inference of $v[B_1]$ over D.*

Proof. Consider the linear-combination ranking function defined in (5.2). We now show that there exists certain value combinations of w_i and w'_i that make it impossible for a Q-only adversary to compromise $v[B_1]$ as long as there does *not* exist another tuple v' in D which shares the same value with v on all attributes but B_1. Specifically, if $\forall S_1, S_2 \subseteq \{w_1, \ldots, w_m, w'_2, \ldots, w'_{m'}\}$ where $S_1 \neq S_2$, there is $|\sum S_1 - \sum S_2| > w'_1$, then one can see that no query answer will be changed for all values of $v[B_1]$, because, for any query q, the change of $d(v, q)$ caused by changing the value of $v[B_1]$ is smaller than even the smallest possible rank difference between any two tuples. In other words, it is infeasible for a Q-only adversary to compromise $v[B_1]$. □

As a simple example, consider the linear ranking function in the running example and a database with only two attributes, one public A_1 and one private B_1. If the weighting on A_1 is larger than B_1, and each tuple in the database takes a different value on A_1, then there is no way for a Q-only adversary to infer $v[B_1]$ because the results of every possible query is already determined without knowing the value of B_1 for any tuple. Specifically, a query will always return the tuple that shares its value on A_1, regardless of what values the tuples have on B_1. As such, the inference of $v[B_1]$ from tuple ranks becomes infeasible.

Despite of the worst-case infeasibility, however, in practice it is quite likely for a Q-only adversary to find enough queries to unveil $v[B_1]$, as we shall show in the experimental results. To address these cases, we now develop Algorithm Q-Point for a Q-only adversary to launch a rank-based inference attack over a point query interface. Once again, our goal here is to find a pair of *differential queries* q_θ and q'_θ for each value $\theta \in V_1^{\mathrm{B}} \backslash v[B_1]$. Like in the Q&I-case, without loss of generality, we denote the domain values in V_1^{B} as 0, 1, ..., $|V_1^{\mathrm{B}}| - 1$.

We start by calling Algorithm FIND-Q to find a query q which returns v. Then, we construct and issue $|V_1^{\mathrm{B}}|$ queries $f_0(q)$, $f_1(q)$, ..., $f_{|V_1^{\mathrm{B}}|-1}(q)$. While all these queries share

the exact same predicates as q on A_1, ..., A_m, B_2, ..., $B_{m'}$, there is $f_i(q)[B_1] = i$ for all $i \in [0, |V_1^{\mathrm{B}}| - 1]$. Due to the additivity property, at least one of these $|V_1^{\mathrm{B}}|$ queries must return v. If only one does, then our attack on $v[B_1]$ already succeeds - the one which returns v must have the same value as v on B_1. If more than one return v, we can do two things: First, we can exclude from consideration those values corresponding to the queries that do not return v - for those, we have already found their differential queries to support the exclusion. Second, we can proceed to revise q (and correspondingly $f_i(q)$) as follows to continue the exclusion process.

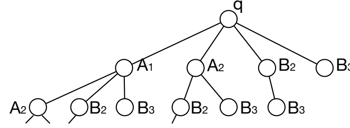


Figure 5.2: Enumeration Tree in Algorithm Q-Point

Specifically, our query-revision process can be considered performing a breadth-first search over the tree structure depicted in Figure 5.2, which demonstrates a special case where $m = 2$ and $m' = 3$. In the tree, each node consists of a class of revisions to q. Specifically, a node contains all queries that differ from q exactly on the attributes that appear on the path from the node to the root in the tree. For example, the bottom-left corner node in Figure 5.2 contains all queries that differ from q on A_1 and A_2.

During the search process, for each node encountered, we enumerate all queries q' in the node and repeat the value-exclusion process described above by issuing $f_i(q')$ for all $i \in [0, |V_1^{\mathrm{B}}| - 1]$. Note that the enumeration can be made more efficient with a pruning-based optimization: If for a query q', none of the $|V_1^{\mathrm{B}}|$ queries $f_i(q')$ returns v, then we can safely exclude from future consideration all queries in the subtree of the current node which

only differs from q' on public attribute values. Algorithm 12 summarizes the pseudocode for Algorithm Q-Point.

Performance Analysis: One can see from the design of Q-Point that, in the worst-case scenario, it issues enough queries to determine for every query specifiable through the point-query interface whether it returns v. Thus, Q-Point always accomplishes the attack as long as such an attack is at all feasible over the point-query interface. Nonetheless, the query complexity of Q-Point is $O(\prod_{i=1}^{m} |V_i^{\mathrm{A}}| \cdot \prod_{i=1}^{m'} |V_i^{\mathrm{B}}|)$ - much higher than Q&I-Point, given that real-world databases often feature more public attributes with large domains.

We consider again the linear ranking function in the running example and a database where each tuple is generated i.i.d. uniformly at random. We have the following result for Q-Point:

Theorem 7. *In the above scenario, given q produced by FIND-Q, the probability (taken over the randomness of database D) for Q-Point to infer $v[B_1]$ after issuing only $|V_1^{\mathrm{B}}|$ queries is at least*

$$\left(1 - \prod_{t \in D, t \neq v} \frac{1 + \mathrm{erf}\left(\frac{(d^A(v,t) - w'_1)}{\sqrt{2\sum_{i=2}^{m'}(w_i'^2 \cdot (|V_i^{\mathrm{B}}|-1)/|V_i^{\mathrm{B}}|^2)}}\right)}{1 + \mathrm{erf}\left(\frac{d^A(v,t)}{\sqrt{2\sum_{i=2}^{m'}(w_i'^2 \cdot (|V_i^{\mathrm{B}}|-1)/|V_i^{\mathrm{B}}|^2)}}\right)}\right)^{|V_1^{\mathrm{B}}|-1} \tag{5.11}$$

Proof. Following the results from the proof of Theorem 5, the expected ratio of point queries q which has $q[A_i] = v[A_i]$ for all $i \in [1, m]$ and returns v is

$$p = \prod_{t \in D, t \neq v} \left(\frac{1}{2} + \frac{1}{2} \cdot \mathrm{erf}\left(\frac{d^A(v,t)}{\sqrt{\sum_{i=1}^{m'} 2w_i'^2 \cdot \frac{|V_i^{\mathrm{B}}|-1}{|V_i^{\mathrm{B}}|^2}}}\right)\right) \tag{5.12}$$

because $d^A(v,t)$ is the extra distance a tuple $t \neq v$ has compared with v - and such a distance has to be "covered" by the private attributes in order for t to be returned. A key observation here is that, changing $q[B_1]$ between adjacent values in V_1^B changes the distance by at most w'_1. Thus, in order for a tuple t to be returned after the change, the

private attributes of t still have to "cover" a distance of at least $d^A(v,t) - w'_1$. In other words, the expected ratio of point queries q which (1) has $q[A_i] = v[A_i]$ for all $i \in [1,m]$, (2) returns v, and (3) still returns v after the flip of $q[B_1]$ is at most

$$p' \leq \prod_{t \in D, t \neq v} \left(\frac{1}{2} + \frac{1}{2} \cdot \text{erf} \left(\frac{(d^A(v,t) - w'_1)}{\sqrt{2 \sum_{i=2}^{m'} (w_i'^2 \cdot (|V_i^{\text{B}}| - 1)/|V_i^{\text{B}}|^2)}} \right) \right) \tag{5.13}$$

Thus, among all point queries q which have $q[A_i] = v[A_i]$ for all $i \in [1,m]$ and return v, the ratio that, upon changing the value of $q[B_1]$ to all values in $V_1^B \setminus v[B_1]$, return another tuple in the database is at least

$$\begin{aligned} &\left(1 - p'/p\right)^{|V_1^B|-1} = \\ &\left(1 - \prod_{t \in D, t \neq v} \frac{1 + \text{erf}\left(\frac{(d^A(v,t) - w'_1)}{\sqrt{2 \sum_{i=2}^{m'} (w_i'^2 \cdot (|V_i^{\text{B}}|-1)/|V_i^{\text{B}}|^2)}} \right)}{1 + \text{erf}\left(\frac{d^A(v,t)}{\sqrt{2 \sum_{i=2}^{m'} (w_i'^2 \cdot (|V_i^{\text{B}}|-1)/|V_i^{\text{B}}|^2)}} \right)} \right)^{|V_1^{\text{B}}|-1} \end{aligned} \tag{5.14}$$

One can see that if a point query q has $q[A_i] \neq v[A_i]$ for certain $i \in [1,m]$, then this ratio must be even higher because of the now shorter distance $d^A(q,t)$ a tuple $t \neq v$ needs to "cover" with the private attributes. Thus, (5.14) is indeed a lower bound on the expected ratio for all point queries which return v. □

Similar to the Q&I case, the attack is more (likely to be) efficient with a smaller $|V_1^{\text{B}}|$. Nonetheless, an interesting observation here is that, contrary to the Q&I case, now the larger w'_1 is, the more efficient the attack is likely to be. On the other hand, the efficiency also increases with a larger database size $|D|$ and a smaller weight on other private attributes w'_i (as $\text{erf}(x)$ has a larger derivative when x is close to 0).

Algorithm 12 Q-Point

1: **Input:** v **Output:** $v[B_1]$
2: **while** some query returns v **do**
3: $q \leftarrow$ FIND-Q(v, H_v); Construct enumeration tree T_q for q
4: **for** $i = 1$ to $m + m'$ **do**
5: **for** each query node q' in level i of T_q **do**
6: Construct queries $f_0(q') \ldots f_{|V_1^B|-1}(q')$
7: **if** none return v **then** prune subtree(q')
8: **if** only $f_j(q')$ returns v **then return** j as $v[B_1]$
9: Exclude query nodes $f_k(q')$ that does not return v
10: **return** failure

5.5 IN Query Interface

5.5.1 Q&I Adversary

For Q&I adversaries, the feasibility of rank-based inference attack is established for point-query interface in §5.4. Since point-query interface is a special case of IN, the attack feasibility here is already established. Thus, our focus here is to study how the additional power of IN queries further empowers Q&I adversaries.

Recall from §5.4 that, for Q&I adversaries, rank-based inference can be fairly efficiently reduced to the task of FIND-Q - i.e., identifying a (now IN) query returning victim v. We shall start by showing that, despite of the larger space of queries, the reduction still holds - leading to the design of Algorithm Q&I-IN. Then, we show that, while FIND-Q for IN has the same worst-case query cost as in Q&I-Point, the query cost in practice is likely much smaller.

5.5.1.1 Reduction to finding an IN query that returns v

We start by showing that, so long as a Q&I adversary can call upon FIND-Q to identify an *IN query* q that returns v, it can always infer $v[B_1]$ within $O(m' \cdot \sum_{i=1}^{m'} |V_i^{\mathrm{B}}|)$ queries. The reduction in §5.4 cannot be directly used here as it relies on the ability to find a *point* query returning v - which we do not want FIND-Q to do over an IN-query interface due to high query cost associated with it.

To enable the reduction to finding an IN query, the only difference from point-query case (§5.4.2.1) is that now the input q might have ranges like $\{0, 1, 2\}$ specified as predicates on B_i, instead of a single value as in the point-query case (which we denoted as 0). Fortunately, this change does not alter the key design of reduction construction. What we do now is to define $B'_1, \ldots, B'_h$ as those attributes on which the inserted tuple t has a value that differs from the set specified in q. This could be that $t[B'_i]$ falls outside of the range specified in $q[B'_i]$ (e.g., when $t[B'_i] = 3$ while $q[B'_i] = \{0, 1, 2\}$; or that $t[B'_i]$ is in the range but not the only element of $q[B'_i]$ (e.g., when $t[B'_i] = 0$ and $q[B'_i] = \{0, 1, 2\}$. Here is an example of the sequence of queries we construct:

	$A_1, \ldots, A_m$	B'_1	B'_2	B'_3	B_{others}
q	{0}	{0,1}	{1,2}	{1}	{0}
q_0	{0}	{0,1}	{1,2}	{1}	{0}
q_1	{0}	{0,1}	{1,2}	{0}	{0}
q_2	{0}	{0,1}	{0}	{0}	{0}
q_3	{0}	{0}	{0}	{0}	{0}
t	0	0	0	0	0

Once again, there must exist a pair of adjacent queries q_i and q_{i+1} such that q_i returns v while q_{i+1} does not. The remaining inference process follows §5.4.2.1. For example, if q_1 returns v but q_2 does not, then we can safely infer that $v[B'_2] \neq 0$ due to the additivity condition. Similarly, if q_2 returns v while q_3 does not, we can infer that $v[B'_1] \neq 0$. Thus,

just like in the Q&I-Point case, excluding the query cost of FIND-Q, a Q&I-adversary requires at most $O(m' \cdot \sum_{i=1}^{m'} |V_i^{\mathrm{B}}|)$ queries to compromise $v[B_1]$.

5.5.1.2 Efficiency Enhancement in Q&I-IN

Given that the reduction still holds, we are now ready to study how IN queries empower an adversary to quickly accomplish FIND-Q and find a query that returns v. In the following, we describe a concrete example which demonstrates the significant saving brought by IN queries, followed by the design of Algorithm Q&I-IN.

Example of significant query savings: To understand why IN queries significantly reduce the query cost, consider a simple example where: (1) the number of public attributes m is sufficiently large, so each tuple in the database has a unique value combination for the m public attributes; and (2) the number of private attributes m' is even larger, so the probability for a randomly generated point query to return v is extremely small.

The first observation from this example is that FIND-Q over a point-query interface actually requires an extremely large number of queries. Specifically, note from Theorems 4 and 5 that, for a given m, the query cost can be made arbitrarily large with an increasing m'. On the other hand, if IN queries are available, the attack query cost - more specifically, the number of queries required to find one query returning v - is exactly 1 because an IN query q with $A_i = v[A_i]$ for $i \in [1, m]$ and $B_j = V_j^{\mathrm{B}}$ (essentially "*", i.e., do-not-care) for $j \in [1, m']$ always returns v.

One can see from the example that the usage of IN queries significantly reduces the attack query cost because of a simple reason: the ability for an adversary to *eliminate* all private attributes from a query specification makes it much easier for FIND-Q to unveil the victim tuple from the database, so that the adversary can compromise the private attributes one at a time using the above-described reduction. In other words, with IN queries, an

adversary no longer has to get lucky and guess multiple private attributes correctly at the same time (e.g., in order to have v returned by a point query).

Design of Q&I-IN: Algorithm 13 depicts the pseudocode for Algorithm Q&I-IN, which enables a Q&I-adversary to launch our rank-based inference attack on $v[B_1]$ over an IN query interface. With the algorithm, we start with a query q which has $q[A_i] = v[A_i]$ for all $i \in [1, m]$ and $q[B_j] = V_j^{\mathrm{B}}$ for all $j \in [1, m']$. Then, if q does not return v, we gradually replace predicates on B_i with point predicates (i.e., $B_i = v$ where $v \in V_i^{\mathrm{B}}$). Specifically, we perform what is essentially a *breadth-first search* process which enumerates all value combinations for B_1, $\{B_1, B_2\}$, $\{B_1, B_2, B_3\}$, ..., $\{B_1, \ldots, B_{m'}\}$ in order. For example, when $V_1^{\mathrm{B}} = V_2^{\mathrm{B}} = \{0, 1\}$, the queries we issue are $B_1 = 0$, $B_1 = 1$, $B_1 = 0$ AND $B_2 = 0$, $B_1 = 0$ AND $B_2 = 1$, $B_1 = 1$ AND $B_2 = 0$, $B_1 = 1$ AND $B_2 = 1$, ..., where each query also includes $q[A_i] = v[A_i]$ for all $i \in [1, m]$ and $q[B_j] = V_j^{\mathrm{B}}$ for all unspecified B_j. When we find a query that returns v, we launch the above-described reduction process to complete the attack of $v[B_1]$.

One can see from the algorithm design that, just like in the point-query case, we guarantee a successful attack. But the worst-case query cost for Q&I-IN is also just like Q&I-Point - i.e., $O(\prod_{i=1}^{m'} |V_i^{\mathrm{B}}|)$. As we shall demonstrate in the following worst-case analysis, this query cost still cannot be improved beyond a constant factor.

Algorithm 13 Q&I-IN

1: **Input:** v **Output:** $v[B_1]$

2: Initialize starting query q: $q[A_i] = v[A_i] \, \forall i \in [1, m]$ and $q[B_j] = V_j^B \, \forall j \in [1, m']$

3: Iteratively convert q to a point query till it returns v

4: $v[B_1] \leftarrow$ Q&I-Point(q, v)

5.5.1.3 Query cost analysis

The main result here is that, while the availability of an IN query interface does *not* help a Q&I adversary at all in the worst-case scenario, it does have the potential to significantly reduce the query cost in practice - especially when the number of public attributes is large. To understand why the worst-case scenario remains unchanged, consider the construction in the proof of Theorem 4 which inserts to the database $\sum_{i=1}^{m'}(|V_i^{\mathrm{B}}| - 1)$ tuples described in §5.4. Given the worst-case assumption that, when there is a draw (i.e., $s(t_1|q) = s(t_2|q)$), any inserted tuple will be returned before the victim v, one can see that the adversary gets no help from IN queries - because as long as a query q contains an IN predicate, say on B_i, it is impossible for q to return the victim tuple v as there must exist an inserted tuple which matches q on B_i, has the exact same value combination as v on all other attributes, and therefore will be returned ahead of v in the answer to q. Thus, the worst-case query cost Q&I-IN remains $\Omega(\prod_{i=1}^{m'}|V_i^{\mathrm{B}}|)$ - same as Q&I-Point.

Theorem 8. *In the running example, the expected number of queries FIND-Q requires for finding a query that returns v is 1 if $\min_{t\in D, t\neq v} d^A(v,t) > 0$, and at most $\sum_{h=1}^{m'-1}(c_{h+1} \cdot (1-(1-p(h))^{c_h})$ otherwise, where $c_h = \sum_{i=1}^{h}\prod_{j=1}^{i}|V_i^{\mathrm{B}}|$ and*

$$p(h) = \prod_{t\in D, t\neq v}\left(\frac{1}{2} + \frac{1}{2}\cdot \mathrm{erf}\left(\frac{d^A(v,t)}{\sqrt{2\sum_{i=1}^{h} w_i'^2 \cdot \frac{|V_i^{\mathrm{B}}|-1}{|V_i^{\mathrm{B}}|^2}}}\right)\right) \tag{5.15}$$

Proof. First, when no other tuple $t \in D$ shares the same public-attribute value-combination as v (i.e., $\min_{t\in D, t\neq v} d^A(v,t) > 0$), then as we discussed in the design of Q&I-IN, only one query (with point-predicates and IN-predicates on all public and private attributes, respectively) is required. For other cases, in analogy to the proof of Theorem 5, one can see

that the probability for a randomly generated query q with $q[A_i] = v[A_i]$ for all $i \in [1, m]$ and h point-query predicates specified on private attributes to return v is

$$p(h) = \prod_{t \in D, t \neq v} \left(\frac{1}{2} + \frac{1}{2} \cdot \text{erf} \left(\frac{d^A(v,t)}{\sqrt{2 \sum_{i=1}^{h} w_i'^2 \cdot \frac{|V_i^{\mathrm{B}}|-1}{|V_i^{\mathrm{B}}|^2}}} \right) \right) \tag{5.16}$$

Since the total number of such queries is c^h, and the overall query cost after enumerating queries with h or fewer predicates is (i.e., on B_1, $\{B_1, B_2\}$, ..., $\{B_1, \ldots, B_h\}$, as specified in Q&I-IN) is c^{h+1}, the expected number of queries required by Q&I-IN is at most $\sum_{h=1}^{m'-1}(c_{h+1} \cdot (1 - (1 - p(h))^{c_h})$. □

One can see from the theorem the substantial promise for IN queries to significantly reduce the query cost - not only the query cost can be cut to 1 when no other tuple shares the same public-attribute value-combination as v, but the value of $p(h)$ - i.e., the probability for a query with h point-predicates on private attributes to return v - actually decreases with h. As such, the query cost is likely much smaller than Q&I-Point, especially when the number of public attributes m is large (which leads to a large $d^A(v, t)$).

5.5.2 Q-only adversary

Just like the availability of IN queries does not help reduce the worst-case query cost for Q&I-adversaries, it cannot change the (in)feasibility result for Q-only adversaries either. To understand why, consider a database with the aforementioned linear ranking function and all tuples sharing the same value on B_1. Clearly, the returned tuples will be of the same order regardless of what range the query specifies on B_1. Thus, there is no way for a Q-only adversary to infer which value in $v[B_1]$ all tuples take - proving that Q-only adversaries cannot guarantee the success of rank-based inference even for IN query interfaces. Nonetheless, as we shall show in this subsection and in the experimental results,

the availability of IN queries does help with reducing the query cost in practice, especially when the number of public attributes is large.

Algorithm 14 depicts the pseudocode for Algorithm Q-IN, which enables a Q-only adversary to launch our rank-based inference attack on $v[B_1]$ over an IN query interface. We start with calling Algorithm FIND-Q to find one query q which returns v. Note that, according to the design of FIND-Q, q always has $q[A_i] = v[A_i]$ for all $i \in [1, m]$. After obtaining q, Algorithm Q-IN issues $|V_1^{\mathrm{B}}|$ queries $f_0(q)$, ..., $f_{|V_1^{\mathrm{B}}|-1}(q)$ defined in the same way as in §5.4 - i.e., while all these queries are exactly the same as q on A_1, ..., A_m, B_2, ..., $B_{m'}$, there is $f_i(q)[B_1] = i$ for all $i \in [0, |V_1^{\mathrm{B}}| - 1]$. Similar to the discussion in Algorithm Q-Point, one can see that at least one of these queries must return v, and the attack is already successful if only one of them does. If more than one returns v, we can exclude from consideration those values corresponding to queries that do not return v, and then gradually revise q according to the following procedure.

Specifically, we start with revising q to $q_1, \ldots, q_m$ by changing the predicate of q_i on A_i to (A_i IN V_i^{A}). For each q_i which returns v, we repeat the above process and issue $f_j(q_i)$ for each $j \in [0, |V_1^{\mathrm{B}}| - 1]$ that is not yet excluded as a possible value of $v[B_1]$. Once again, this either directly reveals $v[B_1]$ or further excludes additional values from consideration. If we still cannot pin down $v[B_1]$ after enumerating $q_1, \ldots, q_m$, we consider the process by setting an additional public attribute to its entire domain. For example, if q_1 returns v, we construct $q_{1,x_1}, \ldots, q_{1,x_h}$, such that (1) $q_{x_1}, \ldots, q_{x_h}$ also return v, and (2) $q_{1,i}$ is the same as q_1 on all attributes but A_i, for which there is $q_{1,i}[A_i] = V_i^{\mathrm{A}}$. We repeat this value-exclusion process until finding the exact value of $v[B_1]$, or when we have exhausted all combinations of public attributes - at which time we move back to Algorithm FIND-Q, find another query q which returns v, and attempt the revision process again.

Algorithm 14 Q-IN

1: **Input:** v **Output:** $v[B_1]$
2: **while** some query returns v **do**
3: $q \leftarrow$ FIND-Q(v, H_v)
4: **for** $i = 0$ to m **do**
5: **for** each $\binom{m}{i}$ combination of C of $\{A_1, \ldots, A_m\}$ **do**
6: $q' \leftarrow q; \quad q[A_{i'}] = V_{i'}^{A} \quad \forall A_{i'} \in C$
7: Construct queries $f_0(q') \ldots f_{|V_1^B|-1}(q')$
8: **if** only $f_j(q')$ returned v then return j as $V[B_1]$
9: Exclude query nodes that did not return v

One can see from the design of Algorithm Q-IN that its worst-case query cost is the same as Q-Point, i.e., $O(\prod_{i=1}^{m} |V_i^{\mathrm{A}}| \cdot \prod_{i=1}^{m'} |V_i^{\mathrm{B}}|)$. For the running example and a database where each tuple is generated i.i.d. uniformly at random, we have the following results:

Corollary 3. *In the above scenario, given q from FIND-Q which (1) has point-predicates on $S \subseteq \{A_1, \ldots, A_m\}$, (2) has point-predicates on B_1 and $S' \subseteq \{B_2, \ldots, B_{m'}\}$, and (3) returns v, the probability (taken over the randomness of database D) for Q-IN to infer $v[B_1]$ after issuing only $|V_1^{\mathrm{B}}|$ queries is at least*

$$\left(1 - \prod_{t \in D, t \neq v} \frac{1 + \operatorname{erf}\left(\frac{(d^S(v,t) - w_1')}{\sqrt{2 \sum_{i:B_i \in S'} (w_i'^2 \cdot (|V_i^{\mathrm{B}}|-1)/|V_i^{\mathrm{B}}|^2)}}\right)}{1 + \operatorname{erf}\left(\frac{d^S(v,t)}{\sqrt{2 \sum_{i:B_i \in S'} (w_i'^2 \cdot (|V_i^{\mathrm{B}}|-1)/|V_i^{\mathrm{B}}|^2)}}\right)}\right)^{|V_1^{\mathrm{B}}|-1} \tag{5.17}$$

The corollary follows directly from Theorem 7. We can observe from the theorem the substantial promise for IN queries to significantly reduce the query cost - specifically, note that the smaller S or S' is, the higher this expected ratio will be. As such, the overall query cost is likely much smaller than Q-Point.

5.6 Discussions

5.6.1 Numeric Attributes

Attack Precision for Numeric Private Attribute: In the original problem definition discussed in §5.2, we consider an attack to succeed if and only if the adversary unveils the exact value of a (Boolean or categorical) private attribute. For a numeric (private) attribute, however, it becomes more complex to measure the success of an attack. Specifically, as we shall demonstrate as follows, while there are cases where an adversary can able infer a numeric attribute value to an arbitrary precision, there are also cases where the precision is limited to a (small) fixed range. Nonetheless, either case still represents serious compromise of user privacy.

Interestingly, whether an adversary can infer a numeric attribute B_i to arbitrary precision depends on the *ranking function*, specifically the definition of $s(q[B_i]|t[B_i])$, used by the query interface. If the query interface allows a range to be specified for each attribute, and the ranking function simply assigns $s(q[B_i]|t[B_i]) = 0$ if $t[B_i] \in q[B_i]$ and 1 otherwise, then any adversary which can successfully launch the attack (i.e., finding q_1 and q_2 which only differ on B_1 yet return t at different ranks) can always infer $t[B_i]$ to any precision level (by continuously shrinking $q[B_1]$) as long as the interface allows an arbitrarily small range to be specified in the query. On the other hand, if the interface is point-query only and the ranking function is $s(q[B_i]|t[B_i]) = |q[B_i] - t[B_i]|$ (or with range-query allowed and $s(q[B_i]|t[B_i])$ being the difference between $t[B_i]$ and the center point of $q[B_i]$) with precision set to two digits after decimal point, then clearly no adversary can infer $v[B_1]$ beyond a precision level of 0.01.

Given the wide variety of ranking functions a query interface might feature, and the fact that even a fairly wide interval on B_1 (as long as it is significantly narrower than the entire domain) is usually a significant threat to privacy in practice, discussing the achievable

precision for each type of interfaces is beyond the scope of this paper. Instead, we make an assumption that numeric attributes can be properly discretized (and treated as a categorical one) according to two principles: (1) the discretized range is narrow enough so each tuple has a unique value combination of all attributes, and (2) the range should be as wide as possible, so as to minimize $|V_i^{\mathrm{A}}|$ and $|V_i^{\mathrm{B}}|$, thereby minimizing the query cost of the attack.

5.6.2 Defense Against Rank-based Inference

Since our main objective here is to unveil a novel rank-based inference attack on web databases, a comprehensive discussion of defense methodologies is beyond the scope of this paper. Nonetheless, we would like to briefly describe a few simple defense strategies, and discuss how the analysis of various algorithms in the paper might shed lights on the design of defense.

An obvious defense methodology is to enforce more stringent *practical constraints* discussed in the paper - e.g., requiring a user to answer a CAPTCHA challenge [72] before issuing each query, performing rigid authentication for each tuple insertion/update operation, etc. Another possible strategy here is to *delay* any new tuples from appearing in query answers. As one can see from the design of Q&I-Point and Q&I-IN, this delay may significantly prolong the amount of time a Q&I-adversary needs to launch the attack. However, it is important to note that all defense strategies in this category are essentially making a tradeoff between privacy protection and the *convenience* of bona fide users, and therefore must be designed and implemented carefully (e.g., after user studies).

Another category of defense is to adjust the assignment of public/private attributes and/or the design of ranking function. Recall from the discussions of Q&I-Point and Q-Point that the more attributes the database owner assigns to be private, and the higher weights the ranking function assigns on private attributes, the more difficult it is for an adversary to launch the attack, as the prior knowledge held by an adversary on the victim

tuple (i.e., $v[A_1]$, $\ldots, v[A_m]$) now plays a lesser role on determining the rank, making it harder for the adversary to efficiently locate the victim tuple.

Nonetheless, this strategy does not work as effectively on an IN-query interface. To understand why, note from the design of Q&I-IN that, as long as the public attributes are sufficient for uniquely identifying the victim tuple, a Q&I-adversary can succeed with $O(\sum_{i=1}^{m\prime} |V_i^B|)$ queries no matter how much weight the ranking function places on the private attributes. In this case, the defender can choose to publicize fewer attributes (if doing so prevents an adversary from learning these attribute values for the victim tuple), or disabling IN-query predicates on certain attributes. As we discussed in §5.5, the reduction of IN-query predicates may significantly delay the attack in the average-case scenario.

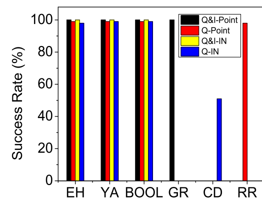


Figure 5.3: Attack Success Rate

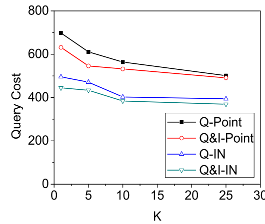


Figure 5.4: Varying k

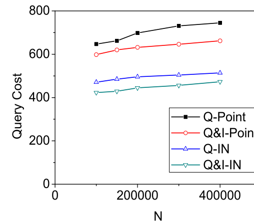


Figure 5.5: Varying n

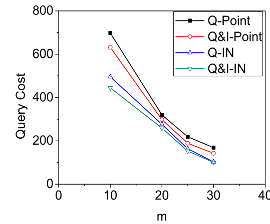


Figure 5.6: Varying m

5.7 Experimental Results

5.7.1 Experimental Setup

Hardware and Platform: All our experiments were performed on a quad-core 2 GHz AMD Phenom machine running Ubuntu 14.04 with 8 GB of RAM. The algorithms were implemented in Python.

Offline Datasets: To verify the correctness of our results, we started by testing our algorithms locally over two real-world and one synthetic dataset. We have full access to

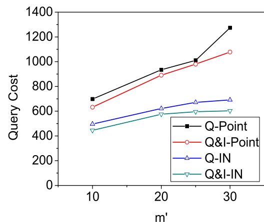


Figure 5.7: Varying m'

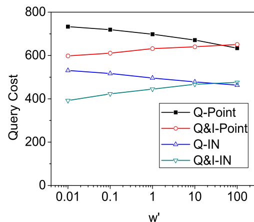


Figure 5.8: Varying w_1'

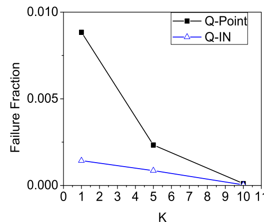


Figure 5.9: Fraction of Uncompromised Accounts

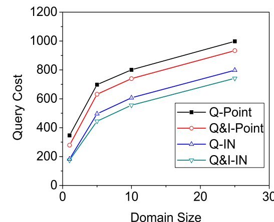


Figure 5.10: Varying Domain Size of Inferred Attribute

these datasets, along with full control of the ranking function used. One dataset is from *eHarmony* [73], a prominent online dating service [74] and consists of anonymized profile information of 500K users. Each user has 56 attributes, of which more than 30 are boolean. The second dataset is *Yahoo! Autos*, which contains 200K used cars for sale in the Dallas-Fort Worth area with 32 Boolean attributes and 6 categorical attributes, the domain cardinalities of which vary from 5 to 447. The third dataset is a synthetic Boolean i.i.d. dataset with 200K tuples and 40 attributes, each following the uniform distribution.

The public and private attributes were randomly chosen from the set of available attributes. By default, we randomly picked 20 attributes for testing, designated $m = 10$ of them as public and $m' = 10$ as private, while varying m and m' between 10 and 30 in various tests. Target attribute B_1 was chosen uniformly at random from all private attributes. Unless otherwise specified, we used the ranking function from (5.2) with all weights set to 1.

Online Demonstration: In order to demonstrate the success of our attacks over real-world websites, we selected three high-profile real-world websites - Renren.com, Amazon Goodreads, Catch22Dating - and conducted live experiments using our algorithms. We would like to note that, without a partnership with these websites, we do not possess/assume any knowledge of their ranking function (other than the monotonicity and additivity proper-

ties defined in §5.2, which we verified through the correctness of our experiment outputs). The results of the online experiments can be found in §5.7.3.

Performance Measures: As discussed earlier in §5.3, we measure efficiency through query cost, i.e., the number of queries required for each attack - consistent with prior work [75, 76].

5.7.2 Experiments over Real-World Datasets

Empirical Evaluation of Attack Success Rate: Figure 5.3 shows the attack success rate of all our algorithms over 3 offline datasets and the relevant algorithm (based on the problem subspace the website falls) over 3 online datasets. As expected, Q&I-adversary has 100% success rate for all datasets. For Q-only adversaries over real-world datasets, we were able to achieve a success rate of almost 100%. The same holds for online tests except CD - the main reason here is that CD allows NULL value on the private attribute we are targeting, leading to failed attacks.

In the following discussion of offline experiments, we focus on results over eHarmony. The results on Yahoo! Autos and the synthetic dataset were largely similar and detailed at the end of this section.

Query Cost versus k: We first investigated the performance of our algorithms for different values of k. Figure 5.4 shows that query cost decreases with higher values of k as expected. Extending our algorithms for $k > 1$ is straightforward. First, we seek to find a query that returns v in top-k (not just top-1). Second, we extend the notion of differential queries (see §5.4.1) such that the v has a higher rank for query q'_θ than for q_θ. The query cost of our algorithms can be broadly categorized into two parts - the query cost to identify a query q that returns the victim tuple and the query cost required to construct additional queries from q through which the private attribute is inferred. When the value of k increases, the

former query cost falls dramatically. Further, the figure also shows that when IN-queries are available (Q&I-IN and Q-IN), the query cost is lower than the cases where only point queries are allowed (Q&I-Point and Q-Point), consistent with our discussions in §5.5.

Query Cost versus Database Size, n: Figure 5.5 depicts the impact of database size on query cost when $k = 1$ (which is henceforth used as the default setting unless otherwise specified). As expected, the increase in database size do not have any major impact and only results in a slight increase in overall query cost. This is due to the fact that the number of queries needed to identify a randomly chosen tuple increases much more slowly than the database size.

Query Cost versus m, m': In our next experiments, we investigate how varying the number of public and private attributes affect the query cost. The results of these experiments are shown in Figures 5.6 and 5.7. As expected, when the number of public attributes increase, the query cost drops significantly. When the number of public attributes are limited, their values are not adequate to distinctly identify a random tuple. Hence, we need to resort to using randomly chosen values for the private attributes which increases query cost. However, when m increases, most tuples become uniquely identified based on their public attributes only. For a fixed m, the query cost increases with increasing m' - when the public attributes are inadequate for uniquely identifying the victim tuple, our algorithms resort to issuing queries where the private attributes are chosen randomly from their respective value domains. But the number of such possible queries increases with higher m' - hence the phenomenon.

Query Cost versus Ranking Weights: In this experiment, we fixed the weight of all public attributes to 1 and varied the weights of private attributes w'_i between 0.01 and 100. The results shown in Figure 5.8 are consistent with our theoretical results from Sections 5.4 and 5.5. When the weights over private attributes decrease, the query cost for Q&I adversaries

also decreases. This is due to the fact that identifying the query q that returns the victim tuple v becomes much easier for this case. The opposite holds for Q-only adversaries where increasing the weights decreases the query cost.

Other Experiments: In order to identify the fraction of tuples in a database that could be successfully compromised using our algorithms, we randomly chose 100K tuples and tried to compromise them. Recall that the Q&I adversary based algorithms are always guaranteed to succeed. Figure 5.9 shows that the Q-only algorithms are able to compromise almost all the tuples. Even with a highly restrictive interface of $k = 1$, Q-Point compromises more than 99% of the tuples. We then adapted our inference algorithms so that they seek to infer all $m' = 25$ private attributes. Figure 5.11 shows the result. While the overall query cost seems high, the amortized query per private attribute varies between 35 and 60. Figure 5.10 shows how varying the domain size of the private attribute affects the query cost. Consistent with our analysis query cost increases with larger domain size.

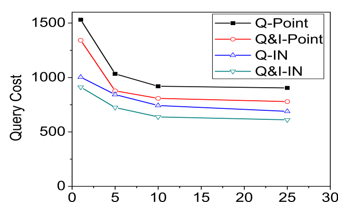


Figure 5.11: Query Cost to Infer all Private Attributes

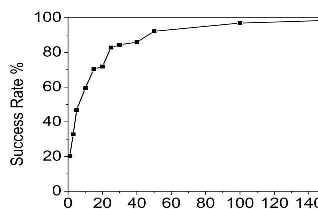


Figure 5.12: Renren: Success Rate vs. k

Experiments over Other Datasets: In addition to eHarmony dataset, we also conducted experiments over two other datasets, Yahoo Autos and BOOL-IID. The results of experiments over Yahoo Autos can be found in Figures 5.13-5.20 while the results for BOOL-IID can be found in Figures 5.21-5.28. We can see that the results are very similar to that of eHarmony demonstrating the wide spread applicability of our algorithms over diverse datasets.

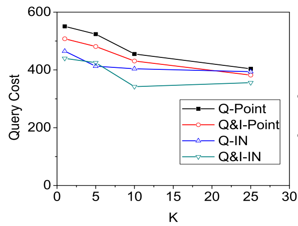


Figure 5.13: Varying k (YA)

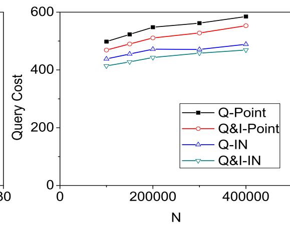


Figure 5.14: Varying n (YA)

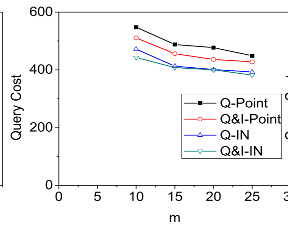


Figure 5.15: Varying m (YA)

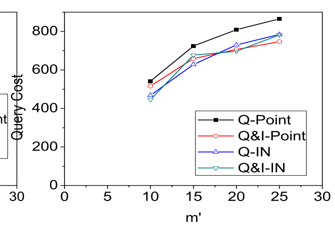


Figure 5.16: Varying m' (YA)

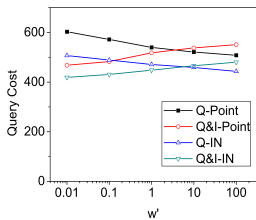


Figure 5.17: Varying w'_1 (YA)

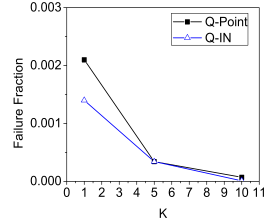


Figure 5.18: Fraction of Uncompromised Accounts (YA)

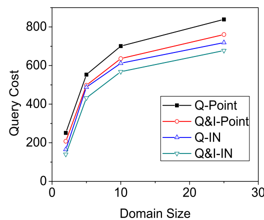


Figure 5.19: Varying Domain Size of Inferred Attribute (YA)

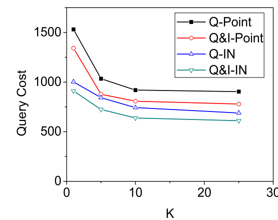


Figure 5.20: Query Cost to Infer all Private Attributes (YA)

5.7.3 Online Demonstration

In the online experiments, we sought to compromise private attributes of user profiles from Amazon Goodreads (GR), Catch22-Dating (CD) and Renren.com (RR) respectively. A detailed description of the procedure we used and its correctness can be found in §5.8. Note that since we have no connection with these websites and thus do not have access to the ground truth, we limit the scope to a small-scale proof-of-concept.

Renren: Renren (RR) [77] is a major Chinese social networking website (similar to Facebook) with more than 160 million users. The user profile consists of details like demographics, education and work affiliation. The website supports extensive privacy settings - allowing a user to specify any subset of profile attributes as public, private or only visible to friends. In our experiments, we focused on one attribute `hometown province` with a domain size of 34 - which is set to private by all users we target. Renren has a search

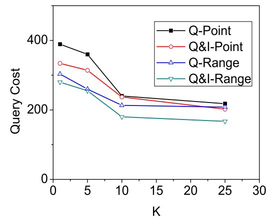


Figure 5.21: Varying k (BOOL)

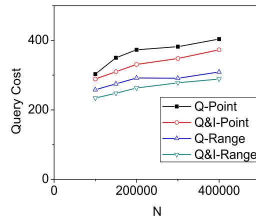


Figure 5.22: Varying n (BOOL)

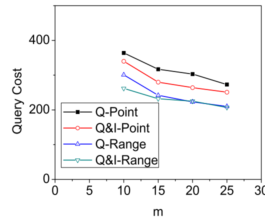


Figure 5.23: Varying m (BOOL)

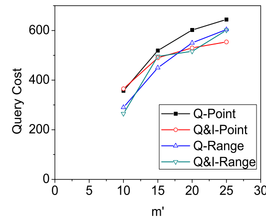


Figure 5.24: Varying m' (BOOL)

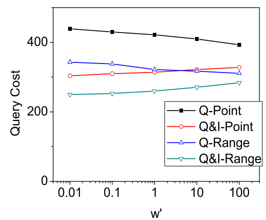


Figure 5.25: Varying w_1' (BOOL)

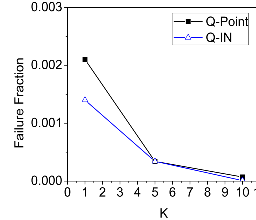


Figure 5.26: Fraction of Uncompromised Accounts (BOOL)

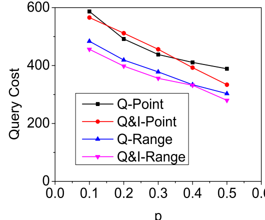


Figure 5.27: Varying p (BOOL)

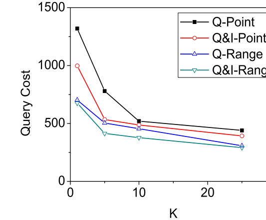


Figure 5.28: Query Cost to Infer all Private Attributes (BOOL)

interface that accepts keyword queries on a user's public attributes such as profile name. It displays appropriate information depending on who issued the query - i.e. everyone can see public attributes, only friends can see attributes marked as visible to friends and none can see the private attributes. The results are ordered based on a ranking function that takes into account the entire profile regardless of privacy settings (as shown in §5.1).

Renren enforces tuple insertion constraint and also allows NULL values. We conducted our attack using Q-Point algorithm. One can see from Table 5.2 that we were able to successfully infer the private attribute for 75 out of 76 profiles with an average query cost of 20 per profile. We also conducted an experiment to measure the success rate of our attacks by varying k. The search interface of Renren, has a large value of k (ranging in hundreds). In our experiment, we artificially truncated the results for different values of k

and verified if we can infer the private attribute. Figure 5.12 shows that for k as little as 50, we achieve a success rate of 92%.

Catch22Dating: Catch22Dating (CD) [78] is an online dating website where users create profiles that are then matched to other users. The public attributes here capture the demographic information of a user, whereas the private attributes specify a user's matching preferences - e.g., the one private attribute we focus on is Boolean "*Is it OK if your matches have been married before*" (henceforth referred to as `Married`). The search interface of Catch22Dating has an option called "Both Perspectives", which enables the ranking function to take into account both public and private attributes of all profiles on the website. It does enforce the tuple insertion constraint by requiring Student ID from selected universities during user registration. It also allows IN queries to be specified (e.g., one can set an attribute to be "do not care" in the query). Hence, we model the adversary as Q-only operating over an IN-interface.

The website allows NULL values on almost all attributes. As a result, our Q-IN attack might fail simply because the user specified NULL as the attribute value. One can see from Table 5.2 that out of the 120 users we attacked, we compromised the private attribute `Married` for 61 of them. For the other 60, either the user did not specify whether he/she would like to accept matches who have been married, or Q-IN attack fails on these users. The average query cost for the success and NULL/failure cases are 60 and 660, respectively, consistent with our prior discussions that failures generally consume many more queries than the successful cases.

Amazon Goodreads (GR) [79] a social cataloging site where the users can connect to each other and share their experience/opinions about books. The user profile consists of demographic information such as `user name` which is always public, and attributes such as `zipcode` which can be set as private. Regardless of a user's choice on location privacy,

the ranking function used in the website's "user search" interface ranks each user according to its (geographic) distance from the location of the user performing the search. Goodreads allows free and instant account registration - i.e., there is no tuple insertion constraint - but no range query. Hence we use the Q&I-Point algorithm.

We started with registering 10 fake accounts with randomly generated ZIP codes, and launched Q&I-Point over it to verify the correctness of our algorithm. Then, to enable verification on real accounts, we identified 53 "special" users at Goodreads who have their ZIP code hidden but chose to reveal their city/state (in US). We launched Q&I-Point successfully on all these users, and then verified that every ZIP code we compromised indeed belongs to its corresponding city/state revealed by the user. The average query cost, as shown in Table 5.2, is 455 per victim.

Table 5.2: Summary of Online Experiments

	#Accounts Attacked	#Success	Avg Cost (Success)	Avg Cost (Failure)
CD	120	61	60	660
GR	53	53	455	N/A
RR	76	75	20	34

5.8 Additional Details for Online Experiments

In this section, we provide some additional details for online experiments. We first describe a practical attack where we infer the private attribute of a user in Catch22Dating website and provide a general approach followed by a formal argument as to its correctness. We then provide the equivalent algorithm for Goodreads. The logic and correctness argument for Goodreads is similar.

Figure 5.29: Query q_1 where Anya is top ranked

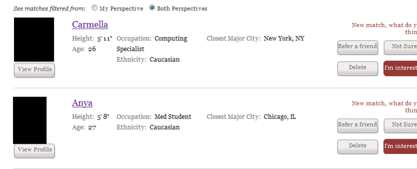


Figure 5.30: Query q_2 where Anya is not top ranked

Example Attack over Catch22Dating: *Catch22dating* (CD) is an online dating website with millions of users. CD allows users to create profiles containing public (such as demographics) and private (such as matching preferences). CD also has a search interface where users could specify a query (based on public information only) to search for other users. CD uses a ranking function that matches the profile using *both* public and private information. Suppose, we wish to infer a private information (*Is it ok if your matches have been married before*) of a user v (with screen name `Anya`). We first created a fake user profile u where we specified the marital status as 'Never married'. Under these circumstances, our results in `Anya` as the best matching user. Figures 5.29 shows the result. We then change u's profile to specify the marital status as 'Previously Married'. When we issue the *same* query (but for the modified profile), we can see that the rank of `Anya` has dropped. We can now plausibly infer that `Anya` has specified that she prefers her matches not to be married before.

Catch22Dating Inference: Using the notations from the technical sections, let v be the victim tuple whose private attribute value $v[B_1]$ we seek to infer. In the context of Catch22Dating, the private boolean attribute B_1 stores the user's response to question: *Is it ok if your matches have been married before*. It can take two values - No or No Preference. The public attribute most relevant to B_1 is A_1 which stores the user's response to the question: *Have you married before*. It takes two values - Yes and No. We construct a random point query by using the public attributes from v's profile and chose the values for private

attributes randomly. However, we set the value for the attribute *Have you married before* to No. If this randomly constructed query (say q_1) returned v, then we create an alternate query q_2. q_2 is identical to q_1 on all attributes except for the value of attribute A_1 - $q_1[A_1]$ = No (not married before) and $q_2[A_1]$ = Yes (had married before). Now if the rank of v is lower in q_2 than in q_1 (i.e., $d_l(q_2, v) > d_l(q_1, v)$), the attacker can infer that the target profile v has private attribute B_1 value set to No.

Correctness Argument: If the target profile v has B_1 value set to No Preference, then $d_l(q_1, v) = d_l(q_2, v)$. This is because by setting $v[B_1]$ to No Preference, the target profile is accepting any value of A_1 in the search query. On the other hand if $v[B_1]$ = No then $d_l(q_1, v) < d_l(q_2, v)$. When the attacker issues a query q_2 followed by q_1, one of the three scenarios can arise:

1. rank of v remains same as it was in q_1
2. rank of v increases
3. rank of v decreases

If $v[B_1]$ = No Preference, only (1) or (2) is possible. While scenario (1) is easy to understand, scenario (2) may appear if there exists a tuple t, such that $t[B_1]$ = No, $d_l(q_1, t) < d_l(q_1, v)$ and $d_l(q_2, t) > d_l(q_2, v)$. Scenario (3) is impossible when $v[B_1]$ = No Preference as it is not possible to find a tuple t that has $d_l(q_1, v) < d_l(q_1, t)$ and $d(q_2, v) > d_l(q_2, t)$. So when the attacker finds that the rank of the target profile v decreases after switching from q_1 to q_2, he/she can correctly infer that $v[B_1]$ = No, because the only assignment $v[B_1]$ can have other than No Preference is No.

Goodreads Inference: Goodreads has a single private attribute `zipcode`. The search interface to find other similar users allows only a single attribute - user name. When displaying the results of a search query it ranks the user profiles (who have the user name from the query) according to a proprietary distance function from the location of the user

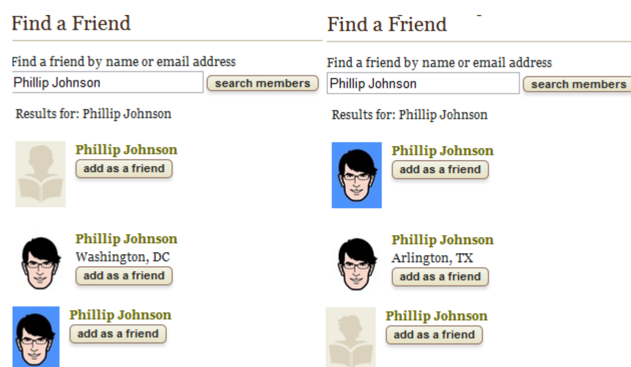


Figure 5.31: Demonstration of an attack over Goodreads

performing the search. Based on our observations, Goodreads seems to use some proprietary variant of zipcode-zipcode distance function. We used a publicly available distance function - but to address the uncertainty of Goodreads' ranking function we added an error margin.

Our attack proceeds in two stages. We start with the set of all zipcodes in USA. Since Goodreads allows an adversary to create multiple accounts, we create two accounts, say a_1, a_2. We set the zipcode of a_1, a_2 to two different randomly chosen zipcodes. We issue a search query based on victim v's user name. Suppose for a_1, v has a higher rank than a_2 (which has the same name as a_1), then remove all zipcodes that has distance higher than the distance between zipcodes of a_1 and a_2 (with an additional error margin) and vice versa. This process is repeated till the zipcode list cannot be pruned anymore. Let the set of all non-pruned zipcodes be Z. In the second stage, we set the zipcode of a_1 to be a random zipcode from Z. We set the zipcode of a_2 to each value in Z and search for v till v has a higher rank than a_1. We then use this information to narrow the zipcodes till we identify the user's zipcode. The results of our experiments can be found in Figure 5.31.

5.9 Related Work

Database Ranking: The area of ranking has been extensively studied in the context of deterministic [6, 70], probabilistic [60] and incomplete [80] data. Processing top-k query when the ranking score is a combination of scores of individual attributes was studied in [11,59]. A popular ranking function is nearest neighbor [69] where the tuples are ordered based on the distance between tuple t and the given query q. Other categorizations such as monotone, generic or no ranking (such as Skyline queries) has also been studied [6]. Recently, there have been studies on learning the rank of a tuple [81] or the ranking function design [82, 83] through a top-k static ranking interface.

Inference Control: Prior work on privacy inference [65] studied the problem of inferring individual tuple values [67, 84] and the existence of a tuple in a database [85] from aggregates such as SUM, MIN, MAX, etc. The field of inference control [65, 66, 86] seeks to prevent such attacks by through query auditing, controlling the number of tuples that match a query or modify query responses using perturbation, distortion etc [87]. Researchers have also proposed multiple privacy preserving aggregate query processing techniques [88, 89]. Recently, [90] has showed that it is possible to infer the location of a user in a Location based Social Network (LBSN) (which could be considered as a private attribute) if the ranking function returns the distance between the query and the victim tuple. However, we do not assume the availability of such information as most websites do not display the score of a tuple for a query.

5.10 Final Remarks

In this paper, we identified a novel problem of rank-based inferencing over databases that use ranked retrieval model. We introduced a taxonomy of the problem space into four important subspaces based on varying interface designs and adversarial capabilities.

For each problem subspace, we developed nontrivial attacking algorithms and conducted theoretical analysis of their feasibility and performance. We verified the effectiveness of the attacks using a comprehensive set of experiments on real-world datasets and online demonstrations on high-profile real-world websites.

It is our hope that the paper initiates a new topic of research on the privacy implications of database ranking; and future research will address the many open problems, e.g., how to design effective defensive strategies that thwart the rank-based inference of private attributes yet maintain the utility of ranking functions.

CHAPTER 6

Eicient Computation of Subspace Skyline over Categorical Domains

Platforms such as AirBnB, Zillow, Yelp, and related sites have transformed the way we search for accommodation, restaurants, etc. The underlying datasets in such applications have numerous attributes that are mostly Boolean or Categorical. Discovering the skyline of such datasets over a subset of attributes would identify entries that stand out while enabling numerous applications. There are only a few algorithms designed to compute the skyline over categorical attributes, yet are applicable only when the number of attributes is small.

In this paper [91], we place the problem of skyline discovery over categorical attributes into perspective and design efficient algorithms for two cases. (i) In the absence of indices, we propose two algorithms, ST-S and ST-P, that exploit the categorical characteristics of the datasets, organizing tuples in a tree data structure, supporting efficient dominance tests over the candidate set. (ii) We then consider the existence of widely used precomputed sorted lists. After discussing several approaches, and studying their limitations, we propose TA-SKY, a novel threshold style algorithm that utilizes sorted lists. Moreover, we further

optimize TA-SKY and explore its progressive nature, making it suitable for applications with strict interactive requirements. In addition to the extensive theoretical analysis of the proposed algorithms, we conduct a comprehensive experimental evaluation of the combination of real (including the entire AirBnB data collection) and synthetic datasets to study the practicality of the proposed algorithms. The results showcase the superior performance of our techniques, outperforming applicable approaches by orders of magnitude.

6.1 Introduction

6.1.1 Motivation

Skyline queries are widely used in applications involving multi-criteria decision making [5], and are further related to well-known problems such as top-k queries [6, 92], preference collection [7], and nearest neighbor search [8]. Given a set of tuples, skylines are computed by considering the dominance relationships among them. A tuple p *dominates* another tuple q, if q is not better than p in any dimension and p is better than q in at least one dimension. Moreover, a pair of tuples p and q are considered to be *incomparable* if neither p nor q dominates the other. The *Skyline* is the set of tuples that are not dominated by any other tuple in the dataset [9].

In recent years, several applications have gained popularity in assisting users in tasks ranging from selecting a hotel in an area to locating a nearby restaurant. AirBnB, TripAdvisor, hotels.com, Craigslist, and Zillow are a few such examples. The underlying datasets have numerous attributes that are mostly Boolean or categorical. They also include a few numeric attributes, but in most cases the numeric attributes are discretized and transformed into categorical attributes [10]. For example, in the popular accommodation rental service AirBnB, the typical attributes are type and number of rooms, types of amenities offered, the number of occupants, etc. Table 6.1 shows a toy example that contains a subset of

Table 6.1: A sample hosts dataset

Host Name	Breakfast	Pool	Cable TV	Internet	Ratings
Host 1	T	F	T	T	4.0
Host 2	T	T	F	T	4.5
Host 3	T	F	F	T	3.5
Host 4	T	F	F	F	3.0
Host 5	F	F	T	T	3.5

attributes present in AirBnB. Note that most of the attributes are amenities provided by the hosts (the temporary rental providers) and are primarily Boolean. The AirBnB dataset features more than 40 such attributes describing amenities users can choose. One way of identifying desirable hosts in such a dataset is to focus on the non-dominated hosts. This is because if a listing t dominates another listing t' (i.e., t is at least as good as t' on all the attributes while is better on at least one attribute), t should naturally be preferred over t'.

In the example shown in Table 6.1, ”Host 1” and ”Host 2” are in the skyline, while all the others are dominated by at least one of them. In real-world applications, especially when the number of attributes increases, users naturally tend to focus on a subset of attributes that is of interest to them. For example, during an AirBnB query, we typically consider a few attributes while searching for hosts that are in the skyline. For instance, in the dataset shown in Table 6.1, one user might be interested in *Breakfast* and *Internet*, while another user might focus on *Internet*, *Cable TV*, and *Pool* when searching for a host.

In this paper, we consider the problem of *subspace skyline discovery* over such datasets, in which given an ad-hoc subset of attributes as a query, the goal is to identify the tuples in the skyline involving only those attributes[1]. Such subspace skyline queries are an effective tool in assisting users in data exploration (e.g., an AirBnB customer can explore the returned skyline to narrow down to a preferred host).

[1] Naturally this definition includes skyline discovery over all attributes of a relation.

In accordance with common practice in traditional database query processing, we design solutions for two important practical instances of this problem, namely: (a) assuming that no indices exist on the underlying dataset, and (b) assuming that indices exist on each individual attribute of the dataset. The space devoted to indices is a practical concern; given that the number of possible subset queries is exponential we do not consider techniques that would construct indices for each possible subset as that would impose an exponential storage overhead (not to mention increased overhead for maintaining such indices under dynamic updates as it is typical in our scenario). Thus we explore a solution space in which index overhead ranges from zero to linear in the number of attributes, trading space for increased performance as numerous techniques in database query processing typically do [93–95].

To the best of our knowledge, LS [10] and Hexagon [96] are the only two algorithms designed to compute skylines over categorical attributes. Both of these algorithms operate by creating *a lattice* over the attributes in a skyline query, which is feasible only when the number of attributes is really small.

6.1.2 Technical Highlights

In this paper, we propose efficient algorithms to effectively identify the answer for any subspace skyline query. Our main focus is to overcome the limitations of previous works ([10, 96]), introducing efficient and scalable skyline algorithms for categorical datasets.

For the case when no indices are available, we design a tree structure to arrange the tuples in a "candidate skyline" set. The tree structure supports efficient dominance tests over the candidate set, thus reducing the overall cost of skyline computation. We then propose two novel algorithms called **ST-S** (Skyline using Tree Sorting-based) and **ST-P** (Skyline using Tree Partition-based) that incorporate the tree structure into existing sorting-

and partition-based algorithms. Both ST-S and ST-P work when no index is available on the underlying datasets and deliver superior performance for any subset skyline query.

Then, we utilize precomputed sorted lists [11] and design efficient algorithms for the index-based version of our problem. As one of the main results of our paper, we propose the Threshold Algorithm for Skyline (**TA-SKY**) capable of answering subspace skyline queries. In the context of **TA-SKY**, we first start with a brief discussion of a few approaches that operate by constructing a full/partial lattice over the query space. However, these algorithms have a complexity that is exponential in the number of attributes involved in the skyline query. To overcome this limitation, we propose **TA-SKY**, an interesting adaptation of the top-K threshold (TA) [11] style of processing for the subspace skyline problem. TA-SKY utilizes sorted lists and constructs the projection of the tuples in query space.

TA-SKY proceeds by accumulating information, utilizing sequential access over the indices that enable it to stop early while guaranteeing that all skyline tuples have been identified. The early stopping condition enables TA-SKY to answer skyline queries *without accessing all the tuples*, thus reducing the total number of dominance checks, resulting in greater efficiency. Consequently, as further discussed in §6.6, TA-SKY demonstrates an order of magnitude speedup during our experiments. In addition to TA-SKY, we subsequently propose novel optimizations to make the algorithm even more efficient. TA-SKY is an online algorithm - it can output a subset of skyline tuples without discovering the entire skyline set. The progressive characteristic of TA-SKY makes it suitable for web applications, with strict interactive requirements, where users want to get a subset of results very quickly. We study this property of TA-SKY in §6.6 on the *entire AirBnB* data collection for which TA-SKY discovered more than two-thirds of the skyline in less than 3 seconds while accessing around 2% of the tuples, demonstrating the practical utility of our proposal.

6.1.3 Summary of Contributions

We propose a comprehensive set of algorithms for the subspace skyline discovery problem over categorical domains. The summary of main contributions of this paper are as follows:

- We present a novel tree data structure that supports efficient dominance tests over relations with categorical attributes.
- We propose the ST-S and ST-P algorithms that utilize the tree data structure for the subspace skyline discovery problem, in the absence of indices.
- We propose TA-SKY, an efficient algorithm for answering subspace skyline queries with a linear worst case cost dependency to the number of attributes. The progressive characteristic of TA-SKY makes it suitable for interactive web-applications. This is a novel and the first (to our knowledge) adaptation of the TA style of processing to a skyline problem.
- We present a comprehensive theoretical analysis of the algorithms quantifying their performance analytically, and present the expected cost of each algorithm.
- We present the results of extensive experimental evaluations of the proposed algorithms over real-world and synthetic datasets at scale showing the benefits of our proposals. In particular, in all cases considered we demonstrate that the performance benefits of our approach are extremely large (in most cases by orders of magnitude) when compared to other applicable approaches.

6.1.4 Paper Organization

The rest of the paper is organized as follows. We discuss preliminaries, notations, and problem definition in §6.2. Then, in §6.3, we present the algorithm for identifying the subspace skyline over low-cardinality datasets, in the absence of precomputed indices. The algorithms for the case of considering the precomputed sorted lists are discussed in §6.4.

Following related work in §6.5, we present the experimental results in §6.6. §6.7 concludes the paper.

6.2 Preliminaries

Consider a relation D with n tuples and $m + 1$ attributes. One of the attributes is $tupleID$, which has a unique value for each tuple. Let the remaining m categorical attributes be $\mathcal{A} = \{A_1, \ldots, A_m\}$. Let $Dom(\cdot)$ be a function that returns the domain of one or more attributes. For example, $Dom(A_i)$ represents the domain of A_i, while $Dom(\mathcal{A})$ represents the Cartesian product of the domains of attributes in $\mathcal{A}$. $|Dom(A_i)|$ represents the cardinality of $Dom(A_i)$. We use $t[A_i]$ to denote the value of t on the attribute A_i. We also assume that for each attribute, the values in the domain have a total ordering by preference (we shall use overloaded notation such as $a > b$ to indicate that value a is preferred over value b).

6.2.1 Skyline

We now define the notions of *dominance* and *skyline* [9] formally.

Definition 2. *(Dominance). A tuple $t \in D$ dominates a tuple $t' \in D$, denoted by $t \succ t'$, iff $\forall A \in \mathcal{A}$, $t[A] \geq t'[A]$ and $\exists A \in \mathcal{A}$, $t[A] > t'[A]$. Moreover, a tuple $t \in D$ is not comparable with a tuple $t' \in D$, denoted by $t \operatorname{sim} t'$, iff $t \not\succ t'$ and $t' \not\succ t$.*

Definition 3. *(Skyline). Skyline, $\mathcal{S}$, is the set of tuples that are not dominated by any other tuples in D, i.e.: $\mathcal{S} = \{t \in D | \nexists t' \in D \text{ s.t. } t' \succ t\}$*

For each tuple $t \in D$, we shall also be interested in computing its $score$ value, denoted by $score(t)$, using a monotonic function $F(\cdot)$. A function $F(\cdot)$ satisfies the monotonicity condition if $F(t) \geq F(t') \Rightarrow t' \not\succ t$.

Subspace Skyline: Let $\mathcal{Q} \subseteq \mathcal{A}$ be a subset of attributes. The attributes in $\mathcal{Q}$ forms a $|\mathcal{Q}|$-dimensional subspace of $\mathcal{A}$. The projection of a tuple $t \in D$ in subspace $\mathcal{Q}$ is denoted by $t_{\mathcal{Q}}$ where $t_{\mathcal{Q}}[A] = t[A]$, $\forall A \in \mathcal{Q}$. Let $D_{\mathcal{Q}}$ be the projection of all tuples of D in subspace $\mathcal{Q}$. A tuple $t_{\mathcal{Q}} \in D_{\mathcal{Q}}$ dominates another tuple $t'_{\mathcal{Q}} \in D_{\mathcal{Q}}$ in subspace $\mathcal{Q}$ (denoted by $t_{\mathcal{Q}} \succ_{\mathcal{Q}} t'_{\mathcal{Q}}$) if $t'_{\mathcal{Q}}$ is not preferred to t on any attribute in $\mathcal{Q}$ while t is preferred to t' on least one attribute in $\mathcal{Q}$.

Definition 4. *(Subspace Skyline). Given a subspace $\mathcal{Q}$, the Subspace Skyline, $\mathcal{S}_{\mathcal{Q}}$, is the set of tuples in $D_{\mathcal{Q}}$ that are not dominated by any other tuples, i.e.: $\mathcal{S}_{\mathcal{Q}} = \{t_{\mathcal{Q}} \in D_{\mathcal{Q}} | \nexists t'_{\mathcal{Q}} \in D_{\mathcal{Q}}$ s.t. $t'_{\mathcal{Q}} \succ_{\mathcal{Q}} t_{\mathcal{Q}}\}$*

6.2.2 Sorted Lists

Sorted lists are popular data structures widely used by many access-based techniques in data management [11, 97]. Let $\mathcal{L} = \{L_1, L_2, \ldots, L_m\}$ be m sorted lists, where L_i corresponds to a (descending) sorted list for attribute A_i. All these lists have the same length, n (i.e., one entry for each tuple in the relation). Each entry of L_i is a pair of the form $(tupleID, t[A_i])$.

A sorted list supports two modes of access: (i) *sorted (or sequential) access*, and (ii) *random access*. Each call to *sorted access* returns an entry with the next highest attribute value. Performing *sorted access* k times on list L_i will return the first k entries in the list. In *random access* mode, we can retrieve the attribute value of a specific tuple. A *random access* on list L_i assumes $tupleID$ of a tuple t as input and returns the corresponding attribute value $t[A_i]$.

6.2.3 Problem Definition

In this paper, we address the efficient computation of *subspace skyline* queries over a relation with categorical attributes. Formally:

Table 6.2: Table of notations

Notation	**Semantics**
D	Relation
n	Number of tuples in the relation
m	Number of attributes
$t_1, \ldots, t_n$	Set of tuples in D
$\mathcal{A}$	Set of attributes in D
$Dom(\cdot)$	Domain of a set of attributes
$score(t)$	Score of the tuple t computed using a monotonic function $F(\cdot)$
$t \succ t'$	t dominates t'
$\mathcal{L}$	Set of m sorted lists
$\mathcal{Q}$	Subspace skyline query
m'	Number of attributes in $\mathcal{Q}$
$D_{\mathcal{Q}}$	Projection of D in query space $\mathcal{Q}$
$\mathcal{S}_{\mathcal{Q}}$	Set of skyline tuples in $D_{\mathcal{Q}}$
$t_{\mathcal{Q}}$	Projection of tuple t in $\mathcal{Q}$
$t_{\mathcal{Q}} \succ_{\mathcal{Q}} t'_{\mathcal{Q}}$	$t_{\mathcal{Q}}$ dominates $t'_{\mathcal{Q}}$ on query space $\mathcal{Q}$
$\mathcal{L}_{\mathcal{Q}}$	Set of sorted lists corresponds to attributes in $\mathcal{Q}$
cv_{ij}	Attribute value returned by i-th sorted access on list L_j
T	Tree for storing the candidate skyline tuples
p_i	the probability that the binary attribute A_i is 1

SUBSPACE SKYLINE DISCOVERY: Given a relation D with the set of categorical attributes $\mathcal{A}$ and a subset of attributes in the form of a subspace skyline query $\mathcal{Q} \subseteq \mathcal{A}$, find the skyline over $\mathcal{Q}$, denoted by $\mathcal{S}_{\mathcal{Q}}$.

In answering subspace skyline queries we consider two scenarios: (i) no precomputed indices are available, and (ii) existence of precomputed sorted lists.

Table 6.2 lists all the notations that are used throughout the paper (we shall introduce some of these later in the paper).

6.3 Skyline Computation Over Categorical Attributes

Without loss of generality, for ease of explanation, we consider a relation with Boolean attributes, i.e., categorical attributes with domain size 2. We shall discuss the

extensions of the algorithms for categorical attributes with larger domains later in this section.

Throughout this section, we consider the case in which precomputed indices are not available. First, we exploit the categorical characteristics of attributes by designing a tree data structure that can perform efficient *dominance* operations. Specifically, given a new tuple t, the tree supports three primitive operations – i) INSERT(t): inserts a new tuple t to the tree, ii) IS-DOMINATED(t): checks if tuple t is dominated by any tuple in the tree, and iii) PRUNE-DOMINATED-TUPLES(t): deletes the tuples dominated by t from the tree. In §6.8.1, we further improve the performance of these basic operations by proposing several optimization techniques. Finally, we propose two algorithms ST-S (Skyline using Tree Sorting-based) and ST-P (Skyline using Tree Partition-based) that incorporate the tree structure to state-of-art sorting- and partition-based algorithms.

6.3.1 Organizing Tuples Tree

Tree structure: We use a binary tree to store tuples in the candidate skyline set. Consider an ordering of all attributes in $\mathcal{Q} \subseteq \mathcal{A}$, e.g., $[A_1, A_2, \ldots, A_{m'}]$. In addition to tuple attributes, we enhance each tuple with a score, assessed using a function $F(\cdot)$. This score assists in improving performance during identification of the dominated tuples or while conducting the dominance check. The proposed algorithm is agnostic to the choice of $F(\cdot)$; the only requirement is that the function does not assign a higher score to a dominated tuple compared to its dominator. The structure of the tree for Example 1 is depicted in Figure 6.1. The tree has a total of 5 ($= m' + 1$) levels, where the i'th level ($1 \leq i \leq m'$) represents attribute A_i. The left (resp. right) edge of each internal node represents value 0 (resp. 1). Each path from the root to a leaf represents a specific assignment of attribute values. The leaf nodes of the tree store two pieces of information: i) $score$: the score of the tuple mapped to that node, and ii) ***tupleID List***: list of ids of the tuples mapped to that

Table 6.3: Example 1 relation

$tupleID$	A_1	A_2	A_3	A_4	$Score$
t_1	1	1	0	0	12
t_2	0	0	1	1	3
t_3	0	1	1	0	6
t_4	1	0	0	1	9
t_5	1	0	1	0	10

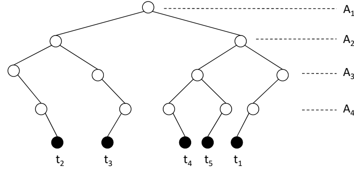


Figure 6.1: Tree structure for relation in Example 1

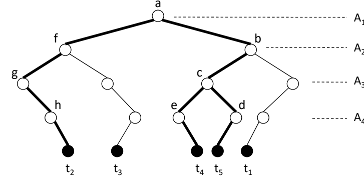


Figure 6.2: Prune dominated tuples

node. Note that all the tuples that are mapped to the same leaf node in the tree have the same attribute value assignment, i.e. have the same score. Moreover, the attribute values of a tuple t can be identified by inspecting the path from the root to a leaf node containing t. Thus, there is no requirement to store the attribute values of the tuples in the leaf nodes. Only the leaf nodes that correspond to an actual tuple are present in the tree.

Example 1. *As a running example through out this section, consider the relation D with $n = 5$ non-dominated tuples where its projection on $\mathcal{Q} = \{A_1, A_2, A_3, A_4\}$ is depicted in Table 6.3. The last column of the table presents the score of each tuple, utilizing the function $F(\cdot)$ provided in Equation 6.1.*

$$F(t_{\mathcal{Q}}) = \sum_{i=1}^{m'} 2^{m'-i} \cdot t[A_i] \tag{6.1}$$

INSERT(t): In order to insert a tuple t into the tree, we start from the root. At level i ($1 \leq i \leq m'$), we check the corresponding attribute value, $t[A_i]$. If $t[A_i] = 0$ (resp. $t[A_i] = 1$) and the left (resp. right) child of current node already exists in the tree, we simply follow the left (resp. right) child. Otherwise, we first have to create a new tree

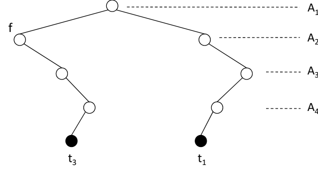


Figure 6.3: Tree after removing dominated tuples

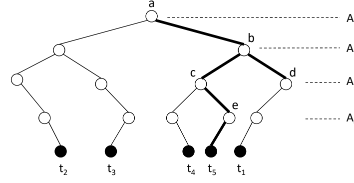


Figure 6.4: Check if tuple t is dominated

node as left (resp. right) child before traversing it. After reaching the leaf node at level $m' + 1$, the $tupleID$ of t is appended to ***tupleID List*** and the $score$ value is assigned to newly constructed leaf.

Algorithm 15 INSERT

1: **Input:** Tuple t, Node n, Level l, Query $\mathcal{Q}$;
2: **if** $l == |\mathcal{Q}| + 1$:
3:     **if** $n.score$ is None: $n.score = score(t)$
4:     Append $t[tupleID]$ to $n.tupleIDList$
5: **else**:
6:     **if** $t[A_l] == 0$:
7:         **if** $n.left$ is $None$:
8:             temp = *New* Node();
9:             $t.left$ = temp;
10:         INSERT(t, $n.left$, $l + 1$)
11:     **if** $t[A_l] == 1$:
12:         **if** $n.right$ is $None$:
13:             temp = *New* Node();
14:             $t.right$ = temp;
15:         INSERT(t, $n.right$, $l + 1$)

PRUNE-DOMINATED-TUPLES(t): The pruning algorithm to delete from the tree, tuples dominated by t, is recursively developed as follows: We start from the root node of the tree. If $t[A_1] = 1$, we search both the left and right subtree. Otherwise, only the left child is selected. This is because if $t[A_1] = 1$, a tuple t' dominated by t can assume value 0 or 1 on attribute A_1. On the other hand, t cannot dominate a tuple t' if $t[A_1] = 0$ and $t'[A_1] = 1$. We follow the same approach at each internal node visited by the algorithm - at level i $(1 \leq i \leq m')$, value of $t[A_i]$ is used to select the appropriate subtree. After reaching a leaf node, we compare $score(t_Q)$ with the $score$ value of leaf node. If both values are equal, no action is required, since, all the tuples mapped into the current leaf node have the same attribute value as t_Q. Else, the leaf node is deleted from the tree. Upon return from the recursion, we check if both the left and right child of the current (internal) node are empty. In that case, the current node is also deleted from the tree.

Figure 6.2 demonstrates the pruning algorithm for $t = \langle 1, 0, 1, 1 \rangle$. Tuples in the tree that are dominated by t are: t_2, t_4, and t_5. The bold edges represent paths followed by the pruning algorithm. Both the left and right children of node a are visited since $t[A_1] = 1$, whereas, at nodes f and b only the left subtree is selected for searching. The final structure of the tree after deleting the dominated tuples is shown in Figure 6.3.

IS-DOMINATED(t): The algorithm starts traversing the tree from the root. For each node visited by the algorithm at level i $(1 \leq i \leq m')$, we check the corresponding attribute value $t[A_i]$. If $t[A_i] = 0$, we search both the left and right subtree; otherwise, we only need to search in the right subtree. This is because when $t[A_i] = 0$, all the tuples dominating t can be either 0 or 1 on attribute A_i. If we reach a leaf node that has an attribute value assignment which is different than that of t (i.e., $score \neq score(t)$), t is dominated. Note that, when $t[A_i] = 0$ both the left and right subtree of the current node can have tuples dominating t, while the cost of identifying a dominating tuple (i.e., the number of nodes visited) may

Algorithm 16 PRUNE-DOMINATED-TUPLES

1: **Input:** Tuple t, Node n, Level l, Score s, Query $\mathcal{Q}$;
2: **if** n is $None$ or $n.minScore > s$ **return**
3: **if** $l == |\mathcal{Q}| + 1$ and $score(t_{\mathcal{Q}}) \neq n.score$:
4: Delete n from tree
5: **return**
6: **if** $t[A_l] == 1$:
7: PRUNE-DOMINATED-TUPLES(t, $n.right$, $l + 1$, s)
8: $s' = s - weight(A_l) \cdot t[A_l]$
9: PRUNE-DOMINATED-TUPLES(t, $n.left$, $l + 1$, s')
10: **else**:
11: PRUNE-DOMINATED-TUPLES(t, $n.left$, $l + 1$, s)
12: **if** Both $left$ and $right$ children of n is $None$
13: Delete n from tree

vary depending on whether the left or right subtree is visited first. For simplicity, we always search in the right subtree first. If there exists a tuple in the subtree of a node that dominates tuple t, we do not need to search in the left subtree anymore.

Figure 6.4 presents the nodes visited by the algorithm in order to check if the new tuple $t = \langle 0, 0, 1, 0 \rangle$ is dominated. We start from the root node a and check the value of t in attribute A_1. Since $t[A_1] = 0$, we first search in the right subtree of a. After reaching to node d, the algorithm back-tracks to b (parent of d). This is because $t[A_3] = 1$ and d has no actual tuple mapped under it's right child. Since $t[A_2] = 0$ and we could not identify any dominating tuple in the right subtree of b, the algorithm starts searching in the left subtree and moves to node c. At node c, only the right child is selected, since $t[A_3] = 1$. Applying the same approach at node f, we reach the leaf node e that contains the tupleID t_5. Since

the value of the $score$ variable at leaf node e is different from $score(t)$, we conclude that tuples mapped into e (i.e., t_5) dominate t.

Please refer to §6.8.1 for further optimizations on the tree data structure.

Algorithm 17 IS-DOMINATED

1: **Input:** Tuple t, Node n, Level l, Score s, Query $\mathcal{Q}$; **Output:** True if t is dominated else False.
2: **if** n is $None$ or $s > n.maxScore$: **return**
3: **if** $l == |\mathcal{Q}|$ and $score(t_{\mathcal{Q}}) \neq n.score$: **return** True
4: **if** $l == |\mathcal{Q}|$ and $score(t_{\mathcal{Q}}) = n.score$: **return** False
5: **if** $t[A_l] == 0$:
6: $\quad s' = s + weight(A_l) \cdot t[A_l]$
7: $\quad$ $dominated$ = IS-DOMINATED(t, $n.right$, $l+1$, s')
8: $\quad$ **if** $dominated$ == True: **return** True
9: $\quad$ **return** IS-DOMINATED(t, $n.left$, $l+1$, s)
10: **else**:
11: $\quad$ **return** IS-DOMINATED(t, $n.right$, $l+1$, s)

6.3.2 Skyline using Tree

Existing works on skyline computation mainly focus on two optimization criteria: reducing the number of dominance checks (CPU cost), limiting communication cost with the backend database (I/O cost). Sorting-based algorithms reduce the number of dominance check by ensuring that only the skyline tuples are inserted in the candidate skyline list. Whereas, partition-based algorithms achieve this by skipping dominance tests among tuples inside incomparable regions generated from the partition. However, given a list of tuples

$\mathcal{T}$ and a new tuple t, in order to discard tuples from $\mathcal{T}$ that are dominated by t, both the sorting- and partition-based algorithms need to compare t against all the tuples in $\mathcal{T}$. This is also the case when we need to check whether t is dominated by T. The tree structure defined in §6.3.1 allows us to perform these operations effectively for categorical attributes. Since the performance gain achieved by the tree structure is independent of the optimization approaches of previous algorithms, it is possible to combine the tree structure with existing skyline algorithms. We now present two algorithms ST-S (Skyline using Tree Sorting-based) and ST-P (Skyline using Tree Partition-based) that incorporates the tree structure into existing algorithm.

ST-S: ST-S combines the tree structure with a sorting-based algorithm. Specifically, we have selected the SaLSa [98] algorithms that exhibits better performance compared to other sorting-based algorithms. The final algorithm is presented in Algorithm 18. The tuples are first sorted according to "maximum coordinate", maxC, criterion[2]. Specifically, Given a skyline query $\mathcal{Q}$, $maxC(t_{\mathcal{Q}}) = (max_{A_i \in \mathcal{Q}}\{t[A_i]\}, sum(t_{\mathcal{Q}}))$, where $sum(t_{\mathcal{Q}}) = \sum_{A_i \in \mathcal{Q}} t[A_i]$. A tree structure T is used to store the skyline tuples. Note that the monotonic property of the scoring function $maxC(\cdot)$ ensures that all the tuples inserted in T are skyline tuples. The algorithm then iterates over the sorted list one by one, and for each new tuple t, if t is not dominated by any tuple in tree T, it is inserted in the tree (lines 7-8). For each new skyline tuple, the "stop point" t_{stop} is updated if required (line 10-12). The algorithm stops if all the tuples are accessed or t_{stop} dominates the remaining tuple. Detailed description of the "stop point" can be found in the original SaLSa paper [98]. We denote with $t^+ = min_{A_i \in \mathcal{Q}}\{t[A_i]\}$ the minimum attribute of tuple t.

ST-P: We have selected the state-of-art partition-based algorithm BSkyTree [99] for designing ST-P. The final algorithm is presented in Algorithm 19. Given a tuple list $\mathcal{T}$, the

[2] Assuming larger values are preferred for each attribute.

Algorithm 18 ST-S

1: **Input:** Tuple list $\mathcal{T}$, Query $\mathcal{Q}$ and Tree T;
   Output: $\mathcal{S}_{\mathcal{Q}}$
2: Sort tuples in D using a monotonic function $maxC(\cdot)$
3: **if** T is $None$: $T \leftarrow$ *New* Tree()
4: $t_{stop} \leftarrow$ undefined
5: **for** each tuple $t \in D$
6:   **if** $t^{+}_{stop} \geq maxC(t_{\mathcal{Q}})$ and $t_{stop} \neq t$: **return**
7:   **if not** IS-DOMINATED($t_{\mathcal{Q}}$, $T.rootNode$, 1, $score(t)$)
8:     INSERT($t_{\mathcal{Q}}$, $T.rootNode$, 1)
9:     Output $t_{\mathcal{Q}}$ as skyline tuple.
10:     $t^{+} \leftarrow min_{A \in \mathcal{Q}}\{t[A]\}$
11:     **if** $t^{+} > t^{+}_{stop}$: $t_{stop} \leftarrow t_{\mathcal{Q}}$

SELECT-PIVOT-POINT method returns a pivot tuple p^V such that it belongs to the skyline of $\mathcal{Q}$ (i.e., $\mathcal{S}_{\mathcal{Q}}$). Moreover, p^V partitions the tuples in $\mathcal{T}$ in a way such that the number of dominance test is minimized (details in [99]). Tuples in $\mathcal{T}$ are then split into $2^{|\mathcal{Q}|}$ lists, each corresponding to one of the $2^{|\mathcal{Q}|}$ regions generated by p^V (lines 7-9). Tuples in $\mathcal{L}[0]$ are dominated by p^V, hence can be pruned safely. For each pair of lists $\mathcal{L}[i]$ and $\mathcal{L}[j]$ ($max \geq j > i \geq 1$), if $\mathcal{L}[j]$ partially dominates $\mathcal{L}[i]$, tuples in $\mathcal{L}[i]$ that are dominated by any tuple in $\mathcal{L}[j]$ are eliminated. Finally, skylines in $\mathcal{L}[i]$ are then discovered in recursive manner (lines 10-15).

Performance Analysis: We now provide a theoretical analysis of the performance of primitive operations utilized by ST-S and ST-P. To make the theoretical analysis tractable, we assume that the underlying data is i.i.d., where p_i is the probability of having value 1 on attribute A_i.

Algorithm 19 ST-P

1: **Input:** Tuple list $\mathcal{T}$ and query $\mathcal{Q}$;
Output: $\mathcal{S}_{\mathcal{Q}}$
2: **if** $|\mathcal{T}| \leq 1$: **return** $\mathcal{T}$
3: $max \leftarrow 2^{|\mathcal{Q}|} - 2$ *//Size of the lattice*
4: $\mathcal{L}[1, max] \leftarrow \{\}$
5: $p^V \leftarrow$ SELECT-PIVOT-POINT($\mathcal{T}$)
6: $\mathcal{S}_{\mathcal{Q}} \leftarrow \mathcal{S}_{\mathcal{Q}} \cup p^V$ *//p^V is a skyline tuple*
7: **for** each tuple $t \in \mathcal{T}$
8: $B^i \leftarrow |\mathcal{Q}|$-bit binary vector corresponds to t wrt p^V
9: **if** $i \neq 0$: $\mathcal{L}[i] \leftarrow \mathcal{L}[i] \cup t$
10: **for** $i \leftarrow$ max to 1
11: $T \leftarrow$ *New* Tree()
12: Insert tuples in $\mathcal{L}[i]$ in T
13: **for** $\forall j \in [max, i) : B^j \succeq B^i$
14: **for** $\forall t \in \mathcal{L}[j]$: PRUNE-DOMINATED-TUPLES($t_{\mathcal{Q}}, T.rootNode$, 1, $score(t_{\mathcal{Q}})$)
15: $\mathcal{S}_{\mathcal{Q}} \leftarrow \mathcal{S}_{\mathcal{Q}} \cup$ ST-P(tuples in T)
16: **return** $\mathcal{S}_{\mathcal{Q}}$

The cost of INSERT-TUPLE($t_{\mathcal{Q}}$) operation is $O(m')$, since to insert a new tuple in the tree one only needs to follow a single path from the root to leaf. For IS-DOMINATED($t_{\mathcal{Q}}$) and PRUNE-DOMINATED-TUPLES($t_{\mathcal{Q}}$), we utilize the number of nodes visited in the tree as the performance measure of these operations.

Consider a tree T with s tuples; Let $Cost(l, s)$ be the expected number of nodes visited by the primitive operations.

Theorem 9. *Considering a relation with n binary attributes where p_i is the probability that a tuple has value 1 on attribute A_i, the expected cost of IS-DOMINATED($t_\mathcal{Q}$) operation on a tree T, containing s tuples is:*

$$
\begin{aligned}
C(m', s) &= 1 \\
C(l, 0) &= 1 \\
C(l, s) &= 1 + \sum_{i=0}^{s} \binom{s}{i} (1 - p_l)^i p_l^{s-i} C'(l, i, s - i)
\end{aligned}
\tag{6.2}
$$

where $S(l, s-i) = 1 - (1 - \prod_{i=1}^{|\mathcal{A}_{ones(t[l+1:m'])}|} p_i)^{s-i}$ *and*[3] $C'(l, i, s-i) = C(l+1, s-i) + (1 - p_l)(1 - S(l, s-i))C(l+1, i)$

Please refer to §6.8.3 for the proof.

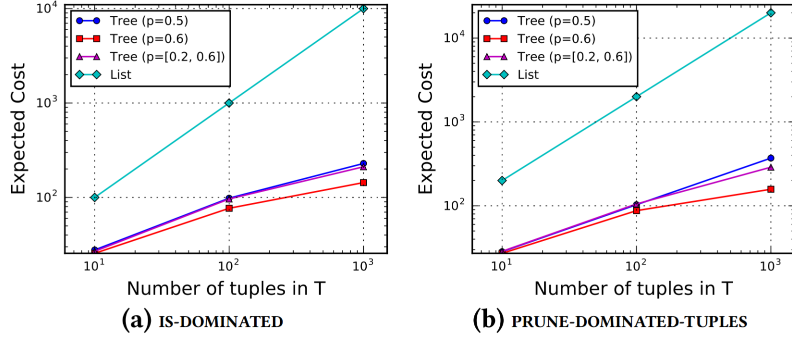
Theorem 10. *Given a boolean relation D with n tuple and the probability of having value 1 on attribute A_i being p_i, the expected cost of PRUNE-DOMINATED-TUPLES($t_\mathcal{Q}$) operation on a tree T, containing s tuples is*

$$
\begin{aligned}
C(m', s) &= 1 \\
C(l, 0) &= 1 \\
C(l, s) &= 1 + \sum_{i=0}^{s} \binom{s}{i} (1 - p_l)^i p_l^{s-i} (C(l + 1, i) + p_l C(l + 1, s - i))
\end{aligned}
\tag{6.3}
$$

The proof is available in §6.8.3

Figure 6.5 uses Equations 6.2 and 6.3 to provide an expected cost for the IS-DOMINATE and PRUNE-DOMINATED-TUPLES operations, for varying numbers of tuples in T (s) where $m' = 20$. We compare its performance with the appraoch, where candidate skyline tuples are organized in a list. Suppose there are s tuples in the list; the best case for the domination test occurs when the first tuple in the list dominates the input tuple ($O(1 \times m')$), while in the worst case, none or only the very last tuple dominates it ($O(s \times m')$) [9]. Thus, on average the dominance test iterates over half of its candidate

[3] $\mathcal{A}_{ones(t[l+1:m'])} = \{A_i | t[A_i] = 1, l + 1 \leq i \leq m'\}$ is the set of remaining attributes of t that has value equals 1.



(a) IS-DOMINATED **(b)** PRUNE-DOMINATED-TUPLES

Figure 6.5: Expected cost of IS-DOMINATED and PRUNE-DOMINATED-TUPLES operations as a function of s

list (i.e., $\frac{s}{2} \times m'$ comparisons). On the other hand, in order to prune tuples in the list that are dominated by t_Q, existing algorithms need to compare t_Q with all the entries in the list. Hence, expected cost of PRUNE-DOMINATED-TUPLES is $s \times m'$. From the figure, we can see that the expected number of comparisons required by the two primitive operations are significantly less when instead of a list, tuples are organized in a tree. Moreover, as p_i increases, the cost of the primitive operations decreases. This is because, when the value of p_i is large, the probability of following left edge (edges corresponds value 0) of a tree node decreases.

The above simulations show that the tree structure can reduce the cost of dominance test effectively thus improving the overall performance of ST algorithms. Although the analysis has been carried out for i.i.d. data, our experimental results in §6.6 show similar behavior for other types of datasets.

6.3.3 Extension for Categorical Attributes

We now discuss how to modify ST algorithm for relations having categorical attributes. We need to make the following two changes:

- The tree structure designed in §6.3.1 needs to be modified for categorical attribute.

- We also need to change the tree traversal algorithms used in each of the three primitive operations.

Tree structure: The tree structure will not be binary anymore. In order to incorporate categorical attributes, each node u at level l ($1 \leq l \leq m$) of the tree now should have $|Dom(A_l)|$ children, one for each attribute value $v \in Dom(A_l)$. We shall index the edges from left to right, where the left most edge corresponds to the lowest attribute value and the attribute value corresponding to each edge increases as we move from left most edge to right most edge.

INSERT(t): After reaching a node u at level l, select the $t[A_l]$-th child of u for moving to the next level of the tree.

IS-DOMINATED(t): We need to follow all the edges that has index value grater or equal to $t[A_l]$.

PRUNE-DOMINATED-TUPLES(t): Search in all the subtrees reachable by following edges with index value less than or equal to $t[A_l]$.

6.4 Subspace Skyline using Sorted Lists

In this section, we consider the availability of sorted lists $L_1, L_2, \ldots L_m$, as per §6.2 and utilize them to design efficient algorithms for subspace skyline discovery. We first briefly discuss a baseline approach that is an extension of LS [10]. Then in §6.4.1, we overcome the barriers of the baseline approach proposing an algorithm named **TOP-DOWN**. The algorithm applies a top-down on-the-fly parsing of the subspace lattice and prunes the dominated branches. However, the expected cost of TOP-DOWN *exponentially* depends on the value of m (§6.8.2). We then propose **TA-SKY** (Threshold Algorithm for Skyline)

Table 6.4: Example: Input Table

	A_1	A_2	A_3	A_4	A_5
t_1	0	1	0	1	1
t_2	0	0	1	1	0
t_3	0	0	1	0	1
t_4	0	0	0	1	1
t_5	1	0	1	1	1
t_6	1	1	1	0	0

in §6.4.2 that does not have such a dependency. In addition to the sorted lists, TA-SKY also utilizes the ST algorithm proposed in §6.3 for computing skylines.

L_1	L_2	L_3	L_4
$(t_5, 1)$	$(t_1, 1)$	$(t_2, 1)$	$(t_1, 1)$
$(t_6, 1)$	$(t_6, 1)$	$(t_3, 1)$	$(t_2, 1)$
$(t_1, 0)$	$(t_2, 0)$	$(t_5, 1)$	$(t_4, 1)$
$(t_2, 0)$	$(t_3, 0)$	$(t_6, 1)$	$(t_5, 1)$
$(t_3, 0)$	$(t_4, 0)$	$(t_1, 0)$	$(t_3, 0)$
$(t_4, 0)$	$(t_5, 0)$	$(t_4, 0)$	$(t_6, 0)$

Figure 6.6: Example: Sorted Lists, Organization 1

L_1	L_2	L_3	L_4
$(t_5, 1)$	$(t_6, 1)$	$(t_5, 1)$	$(t_5, 1)$
$(t_6, 1)$	$(t_1, 1)$	$(t_6, 1)$	$(t_1, 1)$
$(t_1, 0)$	$(t_2, 0)$	$(t_2, 1)$	$(t_2, 1)$
$(t_2, 0)$	$(t_3, 0)$	$(t_3, 1)$	$(t_4, 1)$
$(t_3, 0)$	$(t_4, 0)$	$(t_1, 0)$	$(t_3, 0)$
$(t_4, 0)$	$(t_5, 0)$	$(t_4, 0)$	$(t_6, 0)$

Figure 6.7: Example: Sorted Lists, Organization 2

Example 2. *Let $\mathcal{Q} \subseteq \mathcal{A}$ denotes the set of attributes in a subspace skyline query and $D_{\mathcal{Q}}$ be the projection of D in $\mathcal{Q}$. We denote the set of sorted lists corresponding to a query (one for each attribute involved in the query) as $\mathcal{L}_{\mathcal{Q}}$, $\mathcal{L}_{\mathcal{Q}} = \{L_i | A_i \in \mathcal{Q}\}$. Also, let $m' \leq m$ be $|\mathcal{Q}|$. Our running example uses the relation shown in Table 6.4 through out this section. There are a total of $n = 6$ tuples, each having $m = 5$ attributes. Consider a subspace skyline query $\mathcal{Q} = \{A_1, A_2, A_3, A_4\}$, thus, $m' = 4$. Figure 6.6 shows the corresponding sorted lists $\mathcal{L}_{\mathcal{Q}} = \{L_1, L_2, L_3, L_4\}$.*

BASELINE: We use sorted lists in $\mathcal{L}_{\mathcal{Q}}$ to construct the projection of each tuple $t \in D$ in the query space. For this, we shall perform n sequential accesses on sorted list $L_1 \in \mathcal{L}_{\mathcal{Q}}$. For each $(tupleID, value)$ pair returned by sequential access, we create a new tuple t_{new}.

t_{new} has $tupleID$ as its id and $t_{new}[A_1] = value$. The remaining attribute values of t_{new} are set by performing random access on sorted list L_j ($\forall j \in [2, m']$). After computing the projections of all tuples in query space, we create a lattice over $\mathcal{Q}$ and run the LS algorithm to discover the subspace skyline.

We identify the following problems with BASELINE:

- It makes two passes over all the tuples in the relation.
- It requires the construction of the complete lattice of size $|Dom(\mathcal{Q})|$. For example, when $Dom(A_i) = 4$ and $m' = 15$, the lattice has more than *one billion* nodes; yet the algorithm needs to map the tuples into the lattice.

One observation is that for relations with categorical attributes, especially when m' is relatively small, skyline tuples are more likely to be discovered at the upper levels of the lattice. This motivated us to seek alternate approaches. Unlike BASELINE, TOP-DOWN and the TA-SKY algorithm are designed in a way that they are capable of answering subspace skyline queries by traversing a small portion of the lattice, and more importantly *without the need to access the entire relation.*

6.4.1 TOP-DOWN

Key Idea: Given a subspace skyline query $\mathcal{Q}$, we create a lattice capturing the dominance relationships among the tuples in $D_{\mathcal{Q}}$. Each node in the lattice represents a specific attribute value combination in query space, hence, corresponds to a potential tuple in $D_{\mathcal{Q}}$. For a given lattice node u, if there exist tuples in $D_{\mathcal{Q}}$ with attribute value combination same as u, then all tuples in $D_{\mathcal{Q}}$ corresponding to nodes dominated by u in the lattice are also dominated. TOP-DOWN utilizes this observation to compute skylines for a given subspace skyline query. Instead of iterating over the tuples, TOP-DOWN traverses the lattice nodes from top to bottom; it utilizes sorted lists $\mathcal{L}_{\mathcal{Q}}$ to search for tuples with specific attribute

value combinations. When $|\mathcal{Q}|$ is relatively small, it is likely one will discover all the skyline tuples just by checking few attribute value combinations, without considering the rest of the lattice. However, the expected cost of TOP-DOWN increases exponentially as we increase the query length. Please refer to §6.8.2 for the details and the limitations of TOP-DOWN.

6.4.2 TA-SKY

We now propose our second algorithm, Threshold Algorithm for Skyline (TA-SKY) in order to answer subspace skyline queries. Unlike TOP-DOWN that exponentially depends on m, as we shall show in §6.4.2.1, TA-SKY has a worst case time complexity of $O(m'n^2)$; in addition, we shall also study the expected cost of TA-SKY. The main innovation in TA-SKY is that it follows the style of the well-known Threshold Algorithm (TA) [11] for Top-k query processing, except that it is used for solving a skyline problem rather than a Top-k problem.

TA-SKY iterates over the sorted lists $\mathcal{L}_\mathcal{Q}$ until a stopping condition is satisfied. At each iteration, we perform m' parallel sorted access, one for each sorted list in $\mathcal{L}_\mathcal{Q}$. Let cv_{ij} denote the current value returned from sorted access on list $L_j \in \mathcal{L}_\mathcal{Q}$ $(1 \leq j \leq m')$ at iteration i. Consider τ_i be the set of values returned at iteration i, $\tau_i = \{cv_{i1}, cv_{i1}, \ldots, cv_{im'}\}$. We create a synthetic tuple t_{syn} as the *threshold value* to establish a stopping condition for TA-SKY. The attribute values of synthetic tuple t_{syn} are set according to the current values returned by each sorted list. Specifically, at iteration i, $t_{syn}[A_j] = cv_{ij}, \forall j \in [1, m']$. In other words, t_{syn} corresponds to a potential tuple with the highest possible attribute values that has not been seen by TA-SKY yet.

In addition, TA-SKY also maintains a candidate skyline set. The candidate skyline set materializes the skylines among the tuples seen till the last stopping condition check. We use the tree structure described in §6.3.2 to organize the candidate skyline set. Note

that instead of checking the stopping condition at each iteration, TA-SKY considers the stopping condition at iteration i only when $\tau_i \neq \tau_{i-1}$ $(2 \leq i \leq n)$. $\tau_i \neq \tau_{i-1}$ if and only if $cv_{(i-1)j} \neq cv_{ij}$ $(1 \leq j \leq m')$ for at least one of the m' sequential accesses. This is because the stopping condition does not change among iterations that have the same τ value. Let us assume the value of τ changes at the current iteration i and the stopping condition was last checked at iteration i' $(i' < i)$. Let $\mathcal{T}$ be the set of tuples that are returned in, at least one of the sequential accesses between iteration i' and i. For each tuple $t \in \mathcal{T}$, we perform random access in order to retrieve the values of missing attributes (i.e., attributes of $t_{\mathcal{Q}}$ for which we do not know the values yet). Once the tuples in $\mathcal{T}$ are fully constructed, TA-SKY compares them against the tuples in the candidate skyline set. For each tuple $t \in \mathcal{T}$ three scenarios can arise:

1. t dominates a tuple t' in the tree (i.e., candidate skyline set), t' is deleted from the tree.
2. t is dominated by a tuple t' in the tree, it is discarded since it cannot be skyline.
3. t is not dominated by any tuple t' in the tree, it is inserted in the tree.

Once the candidate skyline set is updated with tuples in $\mathcal{T}$, we compare t_{syn} with the tuples in the candidate skyline set. The algorithm stops when t_{syn} is dominated by any tuple in the candidate skyline set.

We shall now explain TA-SKY for the subspace skyline query $\mathcal{Q}$ of Example 2. Sorted lists $\mathcal{L}_{\mathcal{Q}}$ corresponding to query $\mathcal{Q}$ are shown in Figure 6.6. At iteration 1, TA-SKY retrieves the tuples t_1, t_2 and t_5 by sequential access. For t_1 we know its value on attributes A_2 and A_4 whereas for t_2 and t_5 we know their value on A_3 and A_1 respectively. At this position we have $\mathcal{T} = \{t_1, t_2, t_5\}$ and $\tau_1 = \{1, 1, 1, 1\}$. Note that in addition to storing the tupleIDs that we have seen so far, we also keep track of the attribute values that are known from sequential access. After iteration 2, $\mathcal{T} = \{t_1, t_2, t_3, t_5, t_6\}$ and $\tau_2 = \{1, 1, 1, 1\}$. At iteration 3 we retrieve the values of t_1, t_2, t_5 and t_4 on attributes A_1, A_2, A_3, and A_4

respectively and update the corresponding entries $\mathcal{T}$. Since $\tau_3 = \{0, 0, 1, 1\}$ is different from τ_2, TA-SKY checks the stopping condition. First, we get the missing attribute values (attribute values which are not known from sequential access) of each tuple $t \in \mathcal{T}$. This is done performing random access on the appropriate sorted list in $\mathcal{L}_Q$. After all the tuples in $\mathcal{T}$ are fully constructed, we update the candidate skyline set using them. The final candidate skyline set is constructed after considering all the tuples in $\mathcal{T}$ is $\{t_1, t_5, t_6\}$. Since the synthetic tuple $t_{syn} = \langle 0, 0, 1, 1\rangle$ corresponds to τ_3 is dominated by the candidate skyline set, we stop scanning the sorted lists and output the tuples in the candidate skyline set as the skyline answer set.

The number of tuples inserted into $\mathcal{T}$ (i.e., partially retrieved by sequential accesses) before the stopping condition is satisfied, impacts the performance of TA-SKY. This is because for each tuple $t \in \mathcal{T}$, we have to first perform random accesses in order to get the missing attribute values of t and then compare t with the tuples in the candidate skyline set in order to check if t is skyline. Both the number of random accesses and number of dominance tests increase the execution time of TA-SKY. Hence, it is desirable to have a small number of entries in $\mathcal{T}$. We noticed that the number of tuples inserted in $\mathcal{T}$ by TA-SKY depends on the organization of *(tupleID, value)* pairs (i.e., ordering of pairs having same $value$) in sorted lists. Figure 6.7 displays sorted lists $\mathcal{L}'_Q$ for the same relation in Example 2 but with different organization. Both with $\mathcal{L}_Q$ and $\mathcal{L}'_Q$ TA-SKY stops at iteration 3. However, For $\mathcal{L}_Q$ after iteration 3, $\mathcal{T} = \{t_1, t_2, t_3, t_4, t_5, t_6\}$ and we need to make a total of 12 random accesses and 12 dominance tests[4]. On the other hand, with $\mathcal{L}'_Q$, after iteration 3 we have $\mathcal{T} = \{t_1, t_2, t_5, t_6\}$, requiring only 4 random accesses and 8 dominance tests.

One possible approach to improve the performance of TA-SKY is to re-organize the sorted lists before running the algorithm for a given subspace skyline query. Specifically,

[4]For each tuple $t \in \mathcal{T}$, we need to perform two dominance checks: i) if t is dominating any tuple in the candidate skyline set and ii) if t is dominated by tuples in the candidate skyline set.

$\forall t, t' \in D$ that $t[A_i] = t'[A_i]$, position t before t' in the sorted list L_i $(1 \leq i \leq m')$ if t has better value than t' on the remaining attributes. However, re-arranging the sorted lists for each subspace skyline query will be costly.

We now propose several optimization techniques that enable TA-SKY to compute skylines without considering all the entries in $\mathcal{T}$.

Selecting appropriate entries in $\mathcal{T}$: Our goal is to only perform random access and dominance checks for tuples in $\mathcal{T}$ that are likely to be skyline for a given subspace skyline query. Consider a scenario where TA-SKY needs to check the stopping condition at iteration k, i.e, $\tau_k \neq \tau_{(k-1)}$. Let $\mathcal{Q}'$ be the set of attributes for which the value returned by sequential access at iteration k is different from $(k-1)$-th iteration, $\mathcal{Q}' = \{A_i | A_i \in \mathcal{Q}, cv_{ki} < cv_{(k-1)i}\}$. In order for the tuple t_{syn} to be dominated, there must exist a tuple $t' \in \mathcal{T}$ that has $t'[A_i] \geq t_{syn}[A_i]$, $\forall A_i \in \mathcal{Q}$ and $\exists A_i \in \mathcal{Q}\ s.t.\ t'[A_i] > t_{syn}[A_i]$. Note that each tuple $t \in \mathcal{T}$ has $t[A_i] = t_{syn}[A_i], \forall A_i \in \mathcal{Q} \setminus \mathcal{Q}'$. This is because for all $A_i \in \mathcal{Q} \setminus \mathcal{Q}'$ sorted access returns same value on both $(k-1)$-th and k-th iteration (i.e., $cv_{(k-1)i} = cv_{ki}$). Hence, the only way a tuple $t' \in \mathcal{T}$ can dominate t_{syn} is to have a larger value on any of the attributes in $\mathcal{Q}'$. Therefore, we only need to consider a subset of tuples $\mathcal{T}' = \{t | t \in \mathcal{T}, \exists A_i \in \mathcal{Q} \setminus \mathcal{Q}\ \text{s.t.}\ t[A_i] = cv_{(k-1)i}\}$. Note that it is still possible that $\exists t, t' \in \mathcal{T}'$ s.t. $t \succ_{\mathcal{Q}} t'$. Thus, we need to only consider the tuples that are skylines among $\mathcal{T}'$ and the candidate skyline set. To summarize, before checking the stopping condition at iteration k, we have to perform the following operations: (i) Select a subset of tuples $\mathcal{T}'$ from $\mathcal{T}$ that are likely to dominate t_{syn}, (ii) For each tuple $t \in \mathcal{T}$ get the missing attribute values of t performing random access on appropriate sorted lists, (iii) Update the candidate skyline set using the skylines in $\mathcal{T}'$, and (iv) Check if t_{syn} is dominated by the updated candidate skyline set.

Note that in addition to reducing the number of random access and dominance test, the above optimization technique makes the TA-SKY algorithm *progressive*, i.e, tuples that are inserted into the candidate skyline set will always be skyline in the query space $\mathcal{Q}$. This characteristic of TA-SKY makes it suitable for real-world web applications where instead of waiting for all the results to be returned users want a subset of the results very quickly.

Utilizing the ST algorithms: We can utilize the ST algorithms for discovering the skyline tuples from $\mathcal{T}'$. This way we can take advantages of the optimization approaches proposed in §6.3. For example, we can call ST-S algorithm with parameter: tree T (stores all the tuples discovered so far) and tuple list $\mathcal{T}'$. The output skyline tuples in $\mathcal{T}'$ that are not dominated by T. Moreover, after sorting the tuples in ST-S, if we identify that $score(t_i) = score(t_{i-1})$ $(2 \leq i \leq |\mathcal{T}'|)$ and t_{i-1} is dominated, we can safely mark t_i as dominated. This is because $score(t_i) = score(t_{i-1})$ implies that both t_i and t_{i-1} have same attribute value assignment. When the number of attributes in a subspace skyline query is small, this approach allows us to skip a large number of dominance tests.

The pseudocode of TA-SKY, after applying the optimizations above, is presented in Algorithm 20.

6.4.2.1 Performance Analysis

Worst Case Analysis: In the worst case, TA-SKY will exhaust all the m' sorted lists. Hence, will perform $O(m'n)$ sorted and $O(m'n)$ random accesses. After all the tuples are fully constructed, for each tuple t, we need to check whether any other tuple in T dominates t. The cost of each dominance check operation is $O(m'n)$. Hence, cost of n dominance checks is $O(m'n^2)$. Therefore, the worst case time complexity of TA-SKY is $O(m'n^2)$

Expected Cost Analysis:

Algorithm 20 TA-SKY

1: **Input:** Query $\mathcal{Q}$, Sorted lists $\mathcal{L}_{\mathcal{Q}}$;
   Output: $\mathcal{S}_{\mathcal{Q}}$.
2: $T =$ *New* Tree(); $\mathcal{T} = \emptyset$
3: **repeat**
4: $\tau = \emptyset$
5: **for** each sorted list $L_i \in \mathcal{L}_{\mathcal{Q}}$
6: A_i = Attribute corresponds to L_i
7: $(tupleID, value) = SortedAccess(L)$
8: $\mathcal{T}[tupleID][A_i] = value$
9: $\tau[A_i] = value$
10: **if** τ remains unchanged from prev. iteration:
11: **continue;**
12: $\mathcal{Q}' = \{A_i | A_i \in \mathcal{Q}, \tau[A_i]$ changed from prev.iteration$\}$
13: $\mathcal{T}' = \{t | t \in \mathcal{T}, \exists A_i \in \mathcal{Q}', \mathcal{T}[t][A_i]$ is set$\}$
14: Delete entries from $\mathcal{T}$ that are inserted in $\mathcal{T}'$
15: **for** each $t \in \mathcal{T}'$
16: **for** each attribute $A_i \in Q \setminus Q'$
17: **if** $t[A_i]$ is missing:
18: $t[A_i] = RandomAccess(L, A_i)$
19: Update $score$ of t
20: ST-S($\mathcal{T}$, $\mathcal{Q}$, T)
21: $t_{syn} =$ Synthetic tuple with values of τ
22: **until** IS-DOMINATED(t_{syn}, $T.root$, 1, $score(t_{syn})$)

Lemma 2. *Considering p_i as the probability that a tuple has value 1 on the binary attribute A_i, the expected number of tuples discovered by TA-SKY after i iterations is:*

$$nP_{seen}(t, i) \tag{6.4}$$

where $P_{seen}(t, i)$ is computed using Equation 6.5.

$$P_{seen}(t, i) = 1 - \prod_{j=1}^{m'} \Big((1 - p_j)\big(\sum_{k=0}^{i-1} P_{L_j}(k)\frac{n-i}{n-k} + \sum_{k=i}^{n} P_{L_j}\big) + p_j \sum_{k=i+1}^{n} P_{L_j}(k)\Big) \tag{6.5}$$

Refer to §6.8.3 for the proof.

Theorem 11. *Given a subspace skyline query $\mathcal{Q}$, the expected number of sorted accesses performed by TA-SKY on an n tuple boolean relation with probability of having value 1 on attribute A_j being p_j is,*

$$m' \sum_{i=1}^{n} i \times P_{stop}(i) \tag{6.6}$$

where $P_{stop}(i)$ is computed using Equations 6.7, 6.8, and 6.9.

$$P_{stop}(i) = \sum_{k=1}^{m} P_0(i, k) \times \binom{m'}{k} \times (1 - (1 - P_{stop}(t, \mathcal{Q}_k))^{i'}) \tag{6.7}$$

$$P_0(i, k) = \binom{m'}{k} \prod_{A_j \in \mathcal{Q}_k} (1 - p_j)^{n-i} \prod_{A_j \in \mathcal{Q} \setminus \mathcal{Q}_k} \big(1 - (1 - p_j)^{n-i}\big) \tag{6.8}$$

$$P_{stop}(t, \mathcal{Q}_k) = \Pi_{\forall A_j \in \mathcal{Q} \setminus \mathcal{Q}_k} p_j (1 - \Pi_{\forall A_j \in \mathcal{Q}_k} (1 - p_j)) \tag{6.9}$$

The proof is available in §6.8.3

6.5 Related Work

In the database context, the skyline operator was first introduced in [9]. Since then much work aims to improve the performance of skyline computation in different scenarios. In this paper, we consider skyline algorithms designed for centralized database systems.

To the best of our knowledge, LS [10] and Hexagon [96] are the only two algorithms designed to compute skylines over categorical attributes. Both algorithms operate by first creating the complete lattice of possible attribute-value combinations. Using the lattice structure, non-skyline tuples are then discarded. Even though LS and Hexagon can discover the skylines in linear time, the requirement to construct the entire lattice for each skyline is strict and not scalable. The size of the lattice is exponential in the number of attributes in a skyline query. Moreover, in order to discover the skylines, the algorithms have to scan the entire dataset twice, which is not ideal for online applications.

Most of the existing work on skyline computation concerns relations with *numeric attributes*. Broadly speaking, skyline algorithms for numerical attributes can be categorized as follows. *Sorting-based Algorithms* utilize sorting to improve the performance of skyline computation aiming to discard nonskyline objects using a small number of dominance checks [100, 101]. For any subspace skyline query, such approaches will require sorting the dataset. SaLSa [98] is the best in this category and we demonstrated how our adaptation on categorical domains, namely ST-S outperforms SaLSa.

Partition-based Algorithms recursively partition the dataset into a set of disjoint regions, compute local skylines for each region and merge the results [9, 102]. Among these, BSkyTree [99] has been shown to be the best performer. We demonstrated that our adaptation of this algorithm, namely ST-P, for categorical domains outperforms the vanilla BSkyTree when applied to our application scenario. Other partitioning algorithms, such as NN [8], BBS [103] and ZSearch [104] utilize indexing structures such as R-tree, ZB-tree for efficient region level dominance tests. However, adaptations of such algorithms in the subspace skyline problem would incur exponential space overhead which is not in line with the scope of our work (at most linear to the number of attributes overhead).

A body of work is also devoted to *Subspace Skyline Algorithms* [105–107] which utilize pre-computation to compute skylines for each subspace skyline query. These algo-

rithms impose exponential space overhead, however. Further improvements to reduce the storage overhead in numeric settings [108–111] are highly data dependent and offer no performance guarantee.

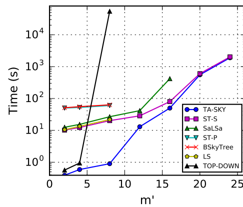


Figure 6.8: Varying query size

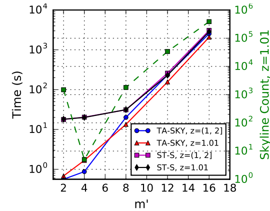


Figure 6.9: Varying query size

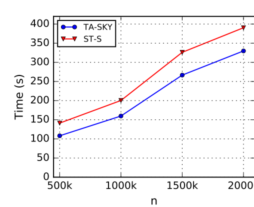


Figure 6.10: Varying number of tuples

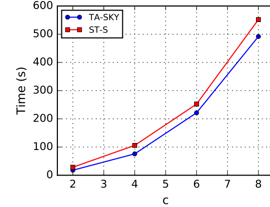


Figure 6.11: Varying cardinality

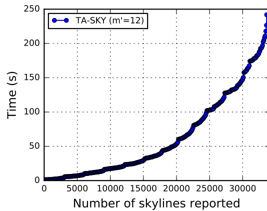


Figure 6.12: Time vs number of skylines returned

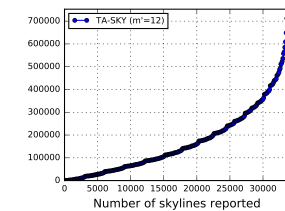


Figure 6.13: Tuples accessed vs number of skylines returned

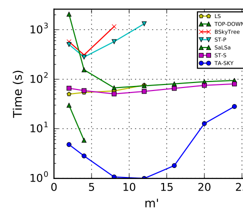


Figure 6.14: AirBnB: Varying query size

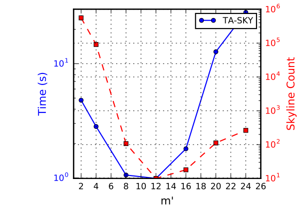


Figure 6.15: AirBnB: TA-SKY performance v.s. Skyline size

6.6 Experimental Evaluation

6.6.1 Experimental Setup

In this section, we describe our experimental results. In addition to the theoretical analysis presented in §6.3 and §6.4, we compared our algorithms experimentally against

existing state-of-the-art algorithms. Our experiments were run over synthetic data, as well as two real-world datasets collected from *AirBnB*[5] and *Zillow*[6].

Synthetic Datasets: In order to study the performance of the proposed algorithms in different scenarios, we generated a number of **Zipfian datasets**, each containing 2M tuples and 30 attributes. Specifically, we created datasets with attribute cardinality ranging from $2 - 8$. In this environment, the frequency of an attribute value is inversely proportional to its rank. Therefore, the number of tuples having a higher (i.e., better) attribute value is less than then number of tuples with a comparatively lower attribute value. We used a Python package for generating these datasets. For each attribute, we specify its distribution over the corresponding domain by controlling the z value. Two attributes having the same cardinality but different z values will have different distributions. Specifically, the attribute with lower z value will have a higher number of tuples having higher attribute value. Unless otherwise specified, we set the z values of the attributes evenly distributed in the range $(1, 2]$ for generating synthetic datasets.

Choice of dataset: we used Zipfian datasets as they reflect more precisely situation with real categorical datasets. Specifically, in real-world applications, for a specific attribute, the number of objects having higher attribute values (i.e., better) is likely to be less than the number of objects with lower attribute values. For example, in AirBnB, *3 bed room* hosts are less frequent than hosts having a *single bed room*. Similarly, in Craigslist, *sedans* are more prevalent than *sports cars*. Moreover, in real-world applications, the distributions of attributes are different from one another. For example, in our AirBnB dataset, approximately 600k out of the 2M hosts have amenity *Cable TV*. Whereas, the approximate number of hosts with amenity *Hot Tub* is only 200k.

[5]http://www.airbnb.com/

[6]http://zillow.com/

AirBnB Dataset: Probably one of the best fits for the application of this paper is AirBnB. It is a peer-to-peer location-based marketplace in which people can rent their properties or look for an abode for a temporary stay. We collected the information of approximately 2 million *real* properties around the globe, shared on this website. AirBnB has a total number of 41 attributes for each host that captures the features and amenities provided by the hosts. Among all the attributes, 36 of them are boolean (categorical with domain size 2) attributes, such as *Breakfast*, *Cable TV*, *Gym*, and *Internet*, while 5 are categorical attributes, such as *Number of Bedrooms*, and *Number of Beds* etc. We tested our proposed algorithms against this dataset to see their performance on real-world applications.

Zillow Dataset: Zillow is a popular online real estate website that helps users to find houses and apartments for sale/rent. We crawled approximately 240k houses listed for sale in Texas and Florida state. For each listing, we collected 9 attributes that are present in all the houses. Out of 9 attributes, 7 of them are categorical, such as *House Type*, *Number of Beds*, *Number of Baths*, *Parking Space* etc., and two are numeric - *House size* (in sqft), and *Price*. The domain cardinalities of the categorical attributes varies from 3 to 30. Using discretization we mapped the numeric attributes into the categorical domain, each of cardinality 20.

Algorithms Evaluated: We tested the proposed algorithms, namely ST-S, ST-P, TOP-DOWN, and TA-SKY as well as the state-of-art algorithms LS [10], SaLSa [98] and BSkyTree [99] that are applicable to our problem settings.

Performance Measures: We consider running time as the main performance measure of the algorithms proposedIn this paper. In addition, we also investigate the key features of ST-S, ST-P and TA-SKY algorithm and demonstrate how they behave under a variety of settings. Each data point is obtained as the average of 25 runs.

Hardware and Platform: All our experiments were performed on a quad-core 3.5 GHz Intel i7 machine running Ubuntu 14.04 with 16 GB of RAM. The algorithms were implemented in Python.

6.6.2 Experiments over Synthetic Datasets

Effect of Query Size m' : We start by comparing the performance of our algorithms with existing state-of-art algorithms that exhibit the best performance in their respective domain. Note that, unlike TA-SKY, the rest of the algorithms do not leverage any indexing structure. The goal of this experiment is to demonstrate how utilizing a small amount of precomputation (compared to the inordinate amount of space required by Skycube algorithms) can improve the performance of subspace skyline computation. Moreover, the precomputation cost is independent of the skyline query. This is because we only need to build the sorted lists once at the beginning. For this experiment, we set $n = 500$k and vary m' between $6 - 24$. In order to match real-world scenarios, we selected attributes with cardinality c ranging between $2 - 6$. Specifically, 50% of the selected attributes have cardinality 2, 30% have cardinality 4, and 20% have cardinality 6. Figure 6.8 shows the experiment result. We can see that when m' is small, TA-SKY outperforms other algorithms. This is because, with small query size, TA-SKY can discover all the skylines by accessing only a small portion of the tuples in the dataset. However as m' increases, the likelihood of a tuple dominating another tuple decreases. Hence, the total number of tuples accessed by TA-SKY before the stopping condition is satisfied also increases. Hence, the performance gap between TA-SKY and ST-S starts to decrease. Both ST-S and ST-P exhibits better performance compared to their baseline algorithms (SaLSa and BSkyTree). Algorithms such as ST-P, BSkyTree, and LS do not scale for larger values of m'. This is because all these algorithms operate by constructing a lattice over the query space which grows ex-

ponentially. Moreover, even though TOP-DOWN initially performed well, it did not not complete successfully for $m' > 4$.

Figure 6.9 demonstrates the effect m' and z on the performance of TA-SKY and ST-S. For this experiment, we created two datasets with cardinality $c = 6$ and different z values. In the first dataset, all the attributes have same z value (i.e., $z = 1.01$), whereas, for the second dataset, z values of the attributes are evenly distributed within the range $(1, 2]$. By setting $z = 1.01$ for all attributes, we increase the frequency of tuples having preferable (i.e., higher) attribute values. Hence, the skyline size of the first dataset is less than the skyline size of the second dataset. This is because tuples with preferable attribute values are likely to dominate more non-skyline tuples, resulting in a small skyline size. Moreover, this also increases the likelihood of the stopping condition being satisfied at an early stage of the iteration. Hence, TA-SKY needs less time for the dataset with $z = 1.01$. In summary, TA-SKY performs better on datasets where more tuples have preferable attribute values. The right-y-axis of Figure 6.9 shows the skyline size for each query length. One can see that as the query size increased, the chance of tuples dominating each other decreased, which resulted in a significant increase in the skyline size. Please note that the increases in the execution time of TA-SKY are due to the increase in the skyline size which is bounded by n. Moreover, as m' increases, there is an initial decrease in skyline size. This is because when m' is small (i.e., 2), the likelihood of a tuple having highest value (i.e., preferable) on all attribute is large.

Effect of Dataset Size (n): Figure 6.10 shows the impact of n on the performance of TA-SKY and ST-S. For this experiment, we used dataset with cardinality $c = 6$, $m' = 12$ and varied n from 500K to 2M. As we increase the value of n, the number of skyline tuples increases. With the increase of skyline size, both TA-SKY and ST-S needs to process more

tuple before satisfying the stop condition. Therefore, total execution time increases with the increase of n.

Effect of Attribute Cardinality (c)**:** In our next experiment, we investigate how changing attribute cardinality affects the execution time of TA-SKY and ST-S. We set the dataset size to $n = 1$M while setting the query size to $m' = 12$, and vary the attribute cardinality c from 4 to 8. Figure 6.11 shows the experiment result. Increasing the cardinality of the attributes increases the total number of skyline tuples. Therefore, effects the total execution time of TA-SKY and ST-S.

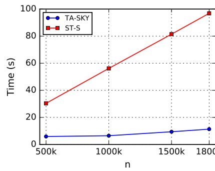


Figure 6.16: AirBnB: Varying the number of tuples

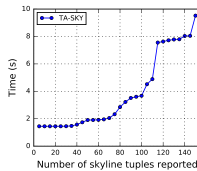


Figure 6.17: AirBnB: Time vs the number of skylines returned

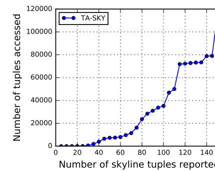


Figure 6.18: AirBnB: Number of accessed tuples vs the number of skylines

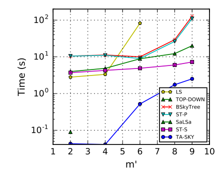


Figure 6.19: Zillow: Varying query size

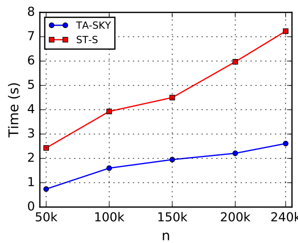


Figure 6.20: Zillow: Varying the number of tuples

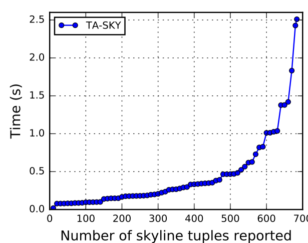


Figure 6.21: Zillow: Time vs the number of skylines returned

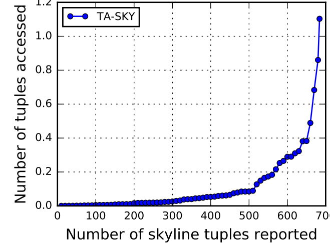


Figure 6.22: Zillow: Number of accessed tuples vs the number of skylines

Progressive Behavior of TA-SKY: Figure 6.12 and 6.13 demonstrates the incremental performance of TA-SKY for discovering the new skylines for a specific query of size $m' = 12$, while $n = 1$M and all the attributes having cardinality $c = 12$. Figure 6.12 shows the CPU time as a function of the skyline size returned. We can see that even though the full skyline discovery takes 250 seconds, within the first 50 seconds TA-SKY outputs more than 50% of the skyline tuples. Figure 6.13 presents the number of tuples TA-SKY accessed as a function of skyline tuples discovered so far. The skyline contains more than 33k tuples. In order to discover all the skylines, TA-SKY needs to access almost 700K (70%) tuples. However, we can see that more than 80% of the skyline tuples can be discovered by accessing less that 30% tuples.

6.6.3 Experiments over AirBnB Dataset

In this experiment, we test the performance of our final algorithm, TA-SKY, against the real Airbnb dataset. We especially study (i) the effects of varying m' and n on the performance of the algorithm and (ii) the progressive behavior of it.

Effect of Varying Query Size (m'): In our first experiment on AirBnB dataset, we compared the performance of different algorithms proposed in the paper with existing works. We varied the number of attributes in the query (i.e., m') from 2 to 24 while setting the number of tuples to 1,800,000. Figure 6.14 shows the experiment result. Similar to our experiment on the synthetic dataset (Figure 6.8), TA-SKY and ST-S perform better than the remaining algorithms. Even though initially performing well, TOP-DOWN did not scale after query length 4. This is because, with $m' > 4$, the skyline hosts shift to the middle of the corresponding query lattice, requiring TOP-DOWN to query many lattice nodes. Figure 6.15 shows the relation between the performance of TA-SKY and the skyline size. Unlike the generally accepted rule of thumb that the skyline size grows exponentially as the number of attributes increases, in this experiment, we see that the skyline size originally

started to decrease as the query size increased and then started to increase again after query size 12. The reason for that is because when the query size is small and n is relatively large, the chance of having many tuples with (almost) all attributes in $\mathcal{Q}$ being 1 (for Boolean attributes) is high. None of these tuples are dominated and form the skyline. However, as the query size increases, the likelihood of having a tuple in the dataset that corresponds to the top node of the lattice decreases. Hence, if the query size gets sufficiently large, we will not see any tuple corresponding to the top node. From then the skyline size will increase with the increase of query size.

Effect of Varying Dataset Size (n): In this experiment, we varied the dataset size from 500,000 to 1,800,000 tuples, while setting m' to 20. Figure 6.16 shows the performance of TA-SKY and ST-S in this case. Once can see that between these two algorithms, the cost of ST-S grows faster. Moreover, even though in the worst case TA-SKY is quadratically dependent on n, it performs significantly better in practice. Especially in this experiment, a factor of 4 increase in the dataset size only increased the execution time by less than a factor of 3.

Progressive Behavior of TA-SKY: As explained in §6.4.2, TA-SKY is a progressive algorithm, i.e., tuples that are inserted into the candidate skyline set are guaranteed to be in $\mathcal{S}_{\mathcal{Q}}$. This characteristic of TA-SKY makes it suitable for real world (especially web) applications, where, rather than delaying the result until the algorithm ends, partial results can gradually be returned to the user. Moreover, we can see that TA-SKY tends to discover a large portion of the skyline quickly within a short execution time with a few number of tuple accesses (as a measure of cost in the web applications). To study this property of the algorithm, in this experiment, we set $n = 1,800,000$ and $m' = 20$ and monitored the execution time, as well as the number of tuple accesses, as the new skyline tuples are discovered. Figures 6.17 and 6.18 show the experiment results for the execution time and the

number of accessed tuples, respectively. One can see in the figure that TA-SKY performed well in discovering a large number of tuples quickly. For example, (i) as shown in Figure 6.17, it discovered more than $\frac{2}{3}$ of the skylines in less that 3 seconds, and (ii) as shown in Figure 6.18, more than half of the skylines were discovered by only accessing less than 2% of the tuples ($20,000$ tuples).

6.6.4 Experiments over Zillow Dataset

We performed the similar set of experiments on Zillow dataset.

In our first experiment, we varied the number of attributes from 2 to 9 while the n is set to 236,194. The experiment result is presented in Figure 6.19. Similar to our previous experiments, ST-S and TA-SKY outperforms the remaining algorithms. This result also shows the effectiveness of ST-S and TA-SKY on categorical attributes with large domain size. For the next experiment, we varied the dataset size (n) from 50,000 to 240,000 tuples, while setting m' to 9. Figure 6.20 shows the performance of ST and TA-SKY for this experiment. Figure 6.21 and 6.22 demonstrate the progressive behavior of TA-SKY for $m' = 9$ and $n = 236,194$. We can see that 90% of skylines are discovered withing the first second and by accessing only 1% tuples.

6.7 Final Remarks

In this paper, we studied the important problem of subspace skyline discovery over datasets with categorical attributes. We first designed a data structure for organizing tuples in candidate skyline list that supports efficient dominance check operations. We then propose two algorithms ST-S and ST-P algorithms for answering subspace skyline queries for the case where precomputed indices are absent. Finally, we considered the existence of precomputed sorted lists and developed TA-SKY, the first threshold style algorithm for skyline

discovery. In addition to the theoretical analysis, our comprehensive set of experiments on synthetic and real datasets confirmed the superior performance of our algorithms.

6.8 Appendix

6.8.1 Tree Data Structure Optimization

Early termination: The tree structure described in §6.3.1, does not store any information inside internal nodes. We can improve the performance of primitive operations (i.e., reduce the number of nodes visited) by storing some information inside each internal node. Specifically, each internal node maintains two variables *minScore* and *maxScore*. The *minScore* (resp. *maxScore*) value of an internal node is the *minimum* (resp. *maximum*) tuple score of all the tuples mapped in the subtree rooted at that node. The availability of such information at each internal node assists in skipping search inside irrelevant regions.

While searching the tree to discover tuples dominated by or dominating a specific tuple t, we also maintain an additional variable *currentScore*, which initially is the same as $score(t)$ at the root of the tree. During traversals, if we follow an edge that matches the corresponding attribute value of t, *currentScore* remains the same[7]. However, if the edge selected by the algorithm differs from the actual attribute value, we update the *currentScore* value accordingly. In the PRUNE-DOMINATED-TUPLES(t) operation, we compare the *minScore* value of each internal node visited by the algorithm with *currentScore*. If the *minScore* value of a node u is higher than *currentScore*, we stop searching in the subtree rooted at u, since it's not possible to have a tuple t' under u that is dominated by t (due to monotonicity). Similarly, while checking if t is dominated by any other tuple in the tree, we stop traversing the subtree rooted at an internal node u if *currentScore* is higher than the *maxScore* value of u.

[7]An edge selected by the algorithm coming out from an internal node at level i matches the attribute value of t if $t[A_i] = 0$ (resp. $t[A_i] = 1$) and we follow the *left* (resp. *right*) edge.

Figure 6.23 presents the value of *minScore* and *maxScore* at each internal node of the tree for the relation in Table 6.3. Consider a new tuple $t = \langle 1, 0, 0, 0 \rangle$. In order to prune the tuples dominated by t, we start from the root node a. At node a *currentScore* = $score(t)$ = 8. Since, $t[A_0] = 1$, we need to search both the left and right subtree of a. The value of *currentScore* at node c remains unchanged since the edge that was used to reach c from a matches the value of $t[A_0]$. However, for b the value of *currentScore* has to be updated. The *currentScore* value at node b is obtained by changing the value of $t[A_0]$ to 0 (values of the other attributes remain the same as in the parent node) and compute the score of the updated tuple. Note that the value of *currentScore* is less than *minScore* in both nodes b and c. Hence we can be sure that no tuple in subtrees rooted at node b and c can be dominated by t.

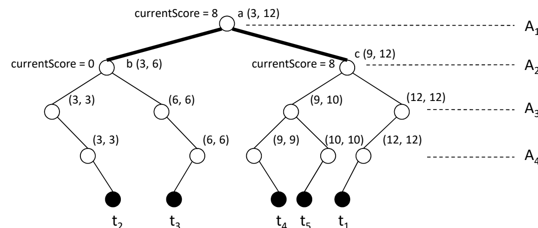


Figure 6.23: Example: Early termination

6.8.2 TOP-DOWN

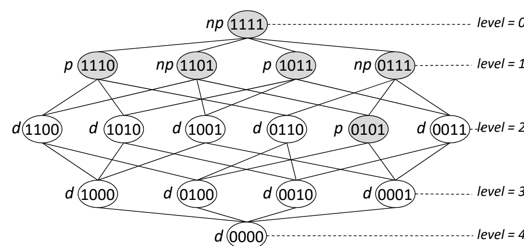


Figure 6.24: Nodes traversed by TOP-DOWN Algorithm

Here we provide the details of the TOP-DOWN algorithm proposed in § 6.4.1. Given a subspace skyline query $\mathcal{Q}$, consider the corresponding subspace lattice. Each node u in the lattice corresponds to a ***unique*** attribute combination which can be represented by a unique id. We assume the existence of the following two functions, (i) $ID(C)$: returns the id of an attribute value combination, and (ii) $InvID(id, m')$: returns the corresponding attribute-value combination for id. The details of these functions can be found in [96].

We observe that given a node identifier id, one can identify the ids of the parents (resp. children) of its corresponding node by calling the two functions $InvID$ and ID. To do so, we first determine the corresponding attribute combination of id. Then identify its parents' (resp. children) combinations by incrementing (resp. decrementing) the value of each attribute, and finally compute the id of each combination using the function ID. TOP-DOWN starts by traversing the lattice from the top node of the lattice. At this node all attributes have the maximum possible value; then conducts a *BFS* over it while constructing the level $(i-1)$ nodes from the non-empty nodes at level i. A node in the lattice is dominated if either one of its parents is dominated or there exists a tuple in the relation that matches the combination of one of its parents.

Let id denote the id of the node in the lattice currently scanned by TOP-DOWN. The algorithm first identifies the parents of the current node and checks if all of them (i) have been constructed (i.e. have not been dominated) and (ii) are marked as *not present* (i.e., there is no tuple in $D_{\mathcal{Q}}$ that had the combination of one of its parents). If so, the algorithm then checks if there exist tuples in $D_{\mathcal{Q}}$ with the same attribute value combination. We use the term *querying a node* in order to refer to this operation. Algorithm 21 presents pseudocode of this operation for a specific attribute value combination. If no such tuple exists in $D_{\mathcal{Q}}$, it marks id as *not present* and moves to the element. Otherwise, it labels id as *present* and outputs the tuples, returned from GET-TUPLES, as the skyline. The TOP-DOWN algorithm queries a node only when the attribute value combination corresponding

to the node is incomparable with the skylines discovered earlier. The algorithm stops when there are no other ids in its processing queue.

Algorithm 21 GET-TUPLES

1: **Input:** Array $values$, Sorted lists $\mathcal{L}_\mathcal{Q}$;

2: **Output:** List of tuples that have the same attribute value assignment as $values$.

3: $tupleIDSet = \emptyset$

4: **for** $i = 1$ to $len(values)$ **do**

5: $currValue = values[i]$

6: $currtupleIDSet =$ Get all tupleIDs from $L_i \in \mathcal{L}_\mathcal{Q}$ that has value $currValue$

7: $tupleIDSet = tupleIDSet \cap currtupleIDSet$

8: $tupleList = [\,]$

9: **for** $tupleID$ in $tupleIDSet$ **do**

10: Construct new tuple t_{new} with attribute values same as $values$ and $t[tupleID] = tupleID$

11: $tupleList.append(t_{new})$

12: **return** $tupleList$;

The lattice structure for the subspace skyline query $\mathcal{Q}$ in Example 2 is shown in Figure 6.24. Each node u in the lattice represents a specific attribute value assignment in the data space corresponding to $\mathcal{Q}$. For example, the top-most node in the lattice represents a tuple t with all the attribute values 1 (i.e., $t[A_i] = 1, \forall A_i \in \mathcal{Q}$). We start from the top node of the lattice. No tuple in $D_\mathcal{Q}$ has value 1 on all the attributes in $\mathcal{Q}$. Therefore, TOP-DOWN marks this node *not present (np)*. We then move to the next level and start scanning nodes from the left. There exists a tuple $t_6 \in D_\mathcal{Q}$ with attribute values $\langle 1, 1, 1, 0\rangle$. Hence, we mark this node *present (p)* and output t_6 as skyline. The algorithm stops after querying

node $\langle 0, 1, 0, 1 \rangle$. TOP-DOWN only needs to query 6 nodes (i.e., check 6 attribute value combinations) in order to discover the skylines. Note that the number of nodes queried by TOP-DOWN is proportional to the number of attributes in $\mathcal{Q}$ and inversely proportional to the relation size n. This is because with large n and small $|\mathcal{Q}|$, the likelihood of having tuples in the relation that correspond to the upper-level nodes of the lattice is high.

Algorithm GET-TUPLES: The algorithm to retrieve tuples in the relation matching the attribute value combination of a specific node is described in Algorithm 21. The algorithm accepts two inputs: (1) $values$ array representing the value of each attribute $A_i \in \mathcal{Q}$, and (2) Sorted lists $\mathcal{L}_{\mathcal{Q}}$. For each attribute $A_i \in \mathcal{Q}$ $(1 \leq i \leq m')$, the algorithm retrieves the set of tupleIDs S_i, that have value equals $values[i]$. This is done by performing a search operation on sorted list L_i. The set of tupleIDs that are discovered in every S_i are the ids of the tuple that satisfy the current attribute value combination. We identify these ids by performing a set intersection operation among all the S_is $(1 \leq i \leq m')$. Once the ids of all the tuples that match values of array $values$ are identified, the algorithm creates tuples for each id with the same attribute value and returns the tuple list.

6.8.2.1 Performance Analysis

For each non-dominated node in the lattice, the TOP-DOWN algorithm invokes the function GET-TUPLES. Hence, we measure the cost of TOP-DOWN as the number of nodes in the lattice for which we invoke GET-TUPLES, times the cost of executing GET-TUPLES function. Since the size of all sorted lists is equal to n, applying binary search on the sorted lists to obtain tuples with a specified value on attribute A_i requires $O(\log(n))$; thus the retrieval cost from all the m' lists is $O(m' \log(n))$. Still taking the intersection between the lists is in $O(nm')$, which makes the worst case cost of the GET-TUPLES operation to be $O(nm')$. Let k be the cost of GET-TUPLES operation over $\mathcal{L}_{\mathcal{Q}}$, for the

Algorithm 22 TOP-DOWN

1: **Input:** Query $\mathcal{Q}$, Sorted lists $\mathcal{L}_{\mathcal{Q}}$;
   Output: $\mathcal{S}_{\mathcal{Q}}$.
2: $processed = \emptyset$;
3: $C =$ the attribute combination of $\mathcal{Q}$ with maximum possible value for each attribute
4: addQ$(queue, ID(C))$
5: **while** $queue$ is not empty **do**
6:   $id =$ delQ$(queue)$
7:   **for** pid in parents$(InvID(id))$
8:     **if** $pid \notin processed$ or pid is marked as present
9:       **continue** *//skip this node*
10:   $tupleList =$ GET-TUPLES$(values, L_{\mathcal{Q}})$
11:   **if** $len(tupleList) == 0$:
12:     append $processed$ by $\langle id,$*not present*$\rangle$
13:     $children =$ children$(InvID(id))$
14:     **for** $c \in children$
15:       **if** c is not in $queue$: addQ$(queue, c)$
16:   **else**:
17:     append $processed$ by $\langle id,$*present*$\rangle$
18:     Output all the tuples in $tupleList$ as skyline.

given relation D. Moreover, considering p_i as the probability that a tuple has value 1 on the binary attribute A_i, we use $C(l)$ to refer to the expected cost of TOP-DOWN algorithm starting from a node u at level l of the lattice.

Theorem 12. *Consider a boolean relation D with n tuples and the probability of having value 1 on attribute A_i being p_i, and a subspace skyline query $\mathcal{Q}$ with m' attributes. The*

expected cost of TOP-DOWN on D and $\mathcal{Q}$ starting from a node at level l is described by the following recursive forumula:

$$C(m') = k/m'$$
$$C(l) = \begin{cases} k + (1 - p_{!\emptyset}(l))m'C(l+1) & \textit{if } l = 0 \\ \frac{1}{l}\{k + (1 - p_{!\emptyset}(l)(m'-l)C(l+1)\} & \textit{otherwise} \end{cases} \quad (6.10)$$

where $p_{!\emptyset}(l) = 1 - (1 - \prod_{i=1}^{l}(1-p_i)\prod_{i=1}^{m'-l} p_i)^n$.

Proof. Consider a node u at level l of the lattice. Node u represents a specific attribute value assignment with l number of 0s and $(m' - l)$ number of 1s. Querying at node u will return all tuples in dataset that have the same attribute value assignment as u. Let $p(t, l)$ be the probability of a tuple $t \in \mathcal{D}_{\mathcal{Q}}$ having l number of 0s and $(m' - l)$ number of 1s.

$$p(t,l) = \prod_{i=1}^{l}(1-p_i)\prod_{i=1}^{m'-l} p_i \quad (6.11)$$

If querying at node u returns at-least one tuple then we do not need to traverse the nodes dominated by u anymore. However, if there exists no tuple in $D_{\mathcal{Q}}$ that corresponds to the attribute value combination of u, we at-least have to query the nodes that are immediately dominated by u. Let $p_{!\emptyset}(l)$ be the probability that there exists a tuple $t \in D_{\mathcal{Q}}$ that has the same attribute value assignment as u. Then,

$$p_{!\emptyset}(l) = 1 - (1 - \prod_{i=1}^{l}(1-p_i)\prod_{i=1}^{m'-l} p_i)^n \quad (6.12)$$

There are total $(m' - l)$ number of nodes immediately dominated by u. Therefore, Cost at node u is the cost of query operation (i.e., k) plus with $(1 - p_{!\emptyset}(l))$ probability the cost of querying its $(m' - l)$ immediately dominated nodes.

$$C(l) = k + (1 - p_{!\emptyset}(l))(m' - l)C(l+1) \tag{6.13}$$

Note that a node u at level l has total l number of immediate dominators causing the cost at node u to be computed l times. However, TOP-DOWN only needs to perform only one query at node u. Hence, the actual cost can be obtained by dividing the computed cost with value l. □

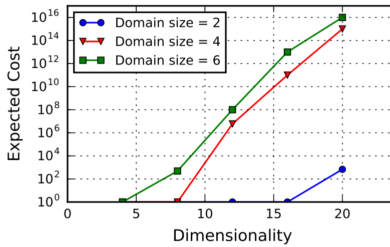


Figure 6.25: Expected number of nodes queried vs. query length

Limitation: We use Equation 6.10 to compute $|C(l)|$ as a function of $|\mathcal{Q}|$ over three uniform relations containing one million tuples with cardinality 2, 4, and 6 respectively. The expected cost increases exponentially as we increase the query length. Moreover, the expected cost also increases when the attributes in $\mathcal{Q}$ have higher cardinality.

6.8.3 Proofs

In this section, we provide detailed proofs for the theorems from the main section of the paper.

THEOREM 9. *Considering a relation with n binary attributes where p_i is the probability that a tuple has value 1 on attribute A_i, the expected cost of IS-DOMINATED($t_{\mathcal{Q}}$) operation on a tree T, containing s tuples is as specified in Equation 6.2.*

Proof. Consider t be the tuple for which we have to check if it is dominated. IS-DOMINATED stops the recursion when we reach a leaf node or move to a node that is empty (i.e., has no tuple mapped under it). Therefore, $C(m', s) = 1$ and $C(l, 0) = 1$.

Let us assume that we are in node u at level l of the tree and there are s tuples mapped in the subtree rooted at u.

If $t[A_l] = 0$, IS-DOMINATED first searches in the right subtree. If no tuple t'_Q in the right subtree dominates t_Q, we then move to the left subtree. Let us assume the right subtree of u contains s_{right} number of tuples ($s_{right} \leq s$). Let $S(l, s_{right})$ be the probability that there exists a tuple in the right subtree of u containing s_{right} tuples that dominates t_Q. In order for a tuple t'_Q to dominate t_Q, it must have at-least value 1 on the attributes in $\mathcal{A}_{ones(t[l+1:m'])}$. This is because, since $t'[A_i] \geq t[A_i]$ ($1 \leq i \leq l-1$) and $t'[A_l] > t[A_l]$, having value 1 on attributes in $\mathcal{A}_{ones(t[l+1:m'])}$ is enough for t'_Q to dominate t_Q. Hence, the probability of t'_Q dominating t_Q is $\prod_{i=1}^{|\mathcal{A}_{ones(t[l+1:m'])}|} p_i$. Therefore,

$$S(l, s_{right}) = 1 - (1 - \prod_{i=1}^{|\mathcal{A}_{ones(t[l+1:m])}|} p_i)^{s_{right}} \quad (6.14)$$

The expected cost of IS-DOMINATED, when $t[A_l] = 0$ is then,

$$(1 - S(l, s_{right}))C(l+1, s - s_{right}) + C(l+1, s_{right}) \quad (6.15)$$

If $t[A_l] = 1$, IS-DOMINATED will always search in the right subtree. Hence, the expected cost when $t[A_l] = 0$ is,

$$C(l+1, s_{right}) \quad (6.16)$$

A node at level-l containing s tuples under it with the probability of having 1 on attribute A_l being p_l, the left subtree will have i tuples with the binomial probability $\binom{s}{i}(1-p_l)^i p_l^{s-i}$. Hence, expected cost node u, $C(l, s)$ is,

$$1 + \sum_{i=0}^{s} \binom{s}{i} (1-p_l)^i p_l^{s-i} (C(l+1, s-i) +$$
$$(1-p_l)(1-S(l, s-i))C(l+1, i)) \quad (6.17)$$

□

THEOREM 10. *Given a boolean relation D with n tuple and the probability of having value 1 on attribute A_i being p_i, the expected cost of PRUNE-DOMINATED-TUPLES(t_Q) operation on a tree T, containing s tuples is as computed in Equation 6.3.*

Proof. PRUNE-DOMINATED-TUPLES(t_Q) stops the recursion when we reach a leaf node or move to a node that is empty (i.e., has no tuple mapped under it). Therefore, $C(m', s) = 1$ and $C(l, 0) = 1$.

Suppose we are in node u at level l of the tree and there are s tuples mapped in the subtree rooted at u.

If $t[A_l] = 0$, we need to search only in the left subtree. Whereas, for $t[A_l] = 1$ we need to search both the left and right subtree.

Let p_l be the probability of having value 1 on attribute A_l. The left subtree of node u at level l (with s tuples under it) will have i tuples with the binomial probability $\binom{s}{i}(1-p_l)^i p_l^{s-i}$. Hence, expected cost at node u, $C(l, s)$, is:

$$1 + \sum_{i=0}^{s} \binom{s}{i} (1-p_l)^i p_l^{s-i} ((1-p_l)C(l+1, i) +$$
$$p_l(C(l+1, i) + C(l+1, s-i))) \quad (6.18)$$

□

LEMMA 2. *Considering p_i as the probability that a tuple has value 1 on the binary attribute A_i, the expected number of tuples discovered by TA-SKY after iterating i lines is as computed in Equation 6.4.*

Proof. The probability that a tuple t is discovered by iterating i rows is one minus the probability that t is not discovered in any of the m' lists in $\mathcal{L}_Q$. Formally:

$$P_{seen}(t,i) = 1 - \Pi_{j=1}^{m'} P_{!seen}(t,i,L_j) \tag{6.19}$$

where $P_{!seen}(t,i,L_j)$ is the probability that t is not discovered at list L_j until row i. $P_{!seen}(t,i,L_j)$ depends on the number of $(tupleId, value)$ pairs with value 1 in list L_j. A list L_j has k number of $(tupleId, value)$ pairs with value 1 if the database has k tuples with value 1 on attribute A_j, while others have value 0 on it. Thus, the probability that L_j has k number of $(tupleId, value)$ pairs with value 1:

$$P_{L_j}(k) = \binom{n}{k}(1-p_j)^{n-k}p_j^k \tag{6.20}$$

t is not seen until row i at list L_j if either of the following cases happen:

- $t[A_j] = 0$ and (considering the random positioning of tuples in lists) t is located after position i in list L_j for all the cases that L_j has k ($k < i$) number of $(tupleId, value)$ pairs with value 1.
- $t[A_j] = 1$ and (considering the random positioning of tuples in lists) t is located after position i in list L_j for all the cases that L_j has k ($k > i$) number of $(tupleId, value)$ pairs with value 1.

Thus:

$$P_{!seen}(t,i,L_j) =$$
$$(1-p_j)\Big(\sum_{k=0}^{i-1} P_{L_j}(k)\frac{n-i}{n-k} + \sum_{k=i}^{n} P_{L_j}\Big) + p_j \sum_{k=i+1}^{n} P_{L_j}(k)\frac{k-i}{k} \tag{6.21}$$

We now can compute $P_{seen}(t,i)$ as following:

$$P_{seen}(t,i) = \\ 1 - \prod_{j=1}^{m'} \Big((1-p_j) \Big(\sum_{k=0}^{i-1} P_{L_j}(k) \frac{n-i}{n-k} + \sum_{k=i}^{n} P_{L_j} \Big) + \\ p_j \sum_{k=i+1}^{n} P_{L_j}(k) \frac{k-i}{k} \Big) \tag{6.22}$$

Having the probability of a tuple being discovered by iterating i lines, the expected number of tuples discovered by iterating i lines is:

$$E_{seen}[i] = nP_{seen}(t,i) = \text{ Equation 6.4}$$

□

THEOREM 11. *Given a subspace skyline query $\mathcal{Q}$, the expected number of sorted access performed by TA-SKY on a n tuple boolean database with probability of having value 1 on attribute A_j being p_j is,*

$$m' \sum_{i=1}^{n} i \times P_{stop}(i)$$

where $P_{stop}(i)$ is computed using Equations 6.7, 6.8, and 6.9.

Proof. Let us first compute the probability that algorithm stops after visiting i rows of the lists. Please note that the algorithm checks the stopping condition at iteration i if $cv_{ij} = 0$ for at least one sorted list. Thus the algorithm stops when (1) $cv_{ij} = 0$ for at least one sorted list AND (2) there exists a tuple among the discovered ones that dominates the maximum possible tuple in the remaining lists.

Suppose i' tuples have seen at least in one of the list so far. Using Lemma 2 we can set $i' = E_{seen}[i]$. Let $P_{j0}(i)$ be the probability that $cv_{ij} = 0$ for sorted list L_j.

$$P_{j0} = (1-p_j)^{n-i} \tag{6.23}$$

Moreover, Consider $P_0(i,k)$ be the probability that after iteration i, $cv_i = 0$ for k sorted lists and $\mathcal{Q}_k$ is corresponding attribute set. Therefore,

$$P_0(i,k) = \binom{m'}{k} \prod_{A_j \in \mathcal{Q}_k} P_{j0} \prod_{A_j \in \mathcal{Q} \setminus \mathcal{Q}_k} (1 - P_{j0}) \tag{6.24}$$

For a given setting that $cv_i = 0$ for k sorted lists, the algorithm stops, *iff* there exists at least one tuple among the discovered ones that dominate the maximum possible value in m' sorted lists; i.e. the value combination that has 0 in k and 1 in all the remaining $m' - k$ positions.

A tuple t need to have the value 1 in all the $m' - k$ list and also *at least one value* 1 *in one of the* k *lists* ($\mathcal{Q}_k$) to dominate the maximum possible remaining value. The probability that a given tuple satisfies this condition is:

$$P_{stop}(t, \mathcal{Q}_k) = \prod_{\forall A_j \in \mathcal{Q} \setminus \mathcal{Q}_k} p_j (1 - \prod_{\forall A_j \in \mathcal{Q}_k} (1 - p_j)) \tag{6.25}$$

Thus, the probability of having at least one tuple that satisfies the dominating condition is:

$$P_{dominate}(i,k) = \binom{m'}{k} \times (1 - (1 - P_{stop}(t, \mathcal{Q}_k))^{i'}) \tag{6.26}$$

We now can compute the probability distribution of the algorithm cost as following:

$$P_{stop}(i) = \sum_{k=1}^{m} P_0(i,k) \times P_{dominate}(k) \tag{6.27}$$

Finally, the expected number of sorted access performed by TA-SKY is:

$$m' \sum_{i=1}^{n} i \times P_{stop}(i) \tag{6.28}$$

□

REFERENCES

[1] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, “A density-based algorithm for discovering clusters in large spatial databases with noise.” in *KDD*, 1996.

[2] P. Xu and S. Tirthapura, “Optimality of clustering properties of space-filling curves,” *TODS*, 2014.

[3] M. F. Mokbel, W. G. Aref, and I. Kamel, “Analysis of multi-dimensional space-filling curves,” *GeoInformatica*, vol. 7, no. 3, pp. 179–209, 2003.

[4] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz, “Analysis of the clustering properties of the hilbert space-filling curve,” *TKDE*, 2001.

[5] C.-L. Hwang and K. Yoon, *Multiple attribute decision making: methods and applications a state-of-the-art survey*. SSBM, 2012, vol. 186.

[6] I. F. Ilyas, G. Beskales, and M. A. Soliman, “A survey of top-k query processing techniques in relational database systems,” *ACM Computing Surveys (CSUR)*, vol. 40, no. 4, p. 11, 2008.

[7] A. Asudeh, G. Zhang, N. Hassan, C. Li, and G. V. Zaruba, “Crowdsourcing pareto-optimal object finding by pairwise comparisons,” in *CIKM*, 2015.

[8] D. Kossmann, F. Ramsak, and S. Rost, “Shooting stars in the sky: An online algorithm for skyline queries,” in *VLDB*, 2002.

[9] S. Borzsony, D. Kossmann, and K. Stocker, "The skyline operator," in *Data Engineering*, 2001.

[10] M. Morse, J. M. Patel, and H. V. Jagadish, "Efficient skyline computation over low-cardinality domains," in *VLDB*, 2007.

[11] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," *JCSS*, vol. 66, no. 4, pp. 614–656, 2003.

[12] W. Liu, M. F. Rahman, S. Thirumuruganathan, N. Zhang, and G. Das, "Aggregate estimations over location based services," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1334–1345, 2015.

[13] N. Dalvi, R. Kumar, A. Machanavajjhala, and V. Rastogi, "Sampling hidden objects using nearest-neighbor oracles," in *SIGKDD*, 2011.

[14] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf, *Computational geometry*. Springer, 2000.

[15] M. Li and et.al., "All your location are belong to us: Breaking mobile social networks for automated user location tracking," in *MobiHoc*, 2014.

[16] D. A. Freedman, *Statistical models: theory and practice*. cambridge university press, 2009.

[17] http://www.census.gov/2010census/data/.

[18] "Google Places API," https://developers.google.com/places/documentation/.

[19] "OpenStreetMap," http://www.openstreetmap.org/.

[20] "WeChat," http://www.wechat.com/en/.

[21] "Sina Weibo," http://weibo.com/.

[22] "," http://investor.starbucks.com/phoenix.zhtml?c=99518&p=irol-financialhighlights.

[23] "WeChat/Weibo Statistics," http://www.guancha.cn/Media/2015_01_29_307911.shtml.

[24] P. Wang and et.al, "An efficient sampling method for characterizing points of interests on maps," in *ICDE*, 2014.

[25] Y. Li, M. Steiner, L. Wang, Z.-L. Zhang, and J. Bao, "Dissecting foursquare venue popularity via random region sampling," in *CoNEXT workshop*, 2012.

[26] Y.-A. de Montjoye and et.al., "Unique in the crowd: The privacy bounds of human mobility," *Scientific reports*, 2013.

[27] C. Y. Ma, D. K. Yau, and N. S. Rao, "Privacy vulnerability of published anonymous mobility traces," *ToN*, 2013.

[28] M. Srivatsa and M. Hicks, "Deanonymizing mobility traces: Using social network as a side-channel," in *CCS*, 2012.

[29] H. Zang and et.al., "Anonymization of location data does not work: A large-scale measurement study," in *MobiCom*, 2011.

[30] A. Dasgupta and et.al., "Unbiased estimation of size and other aggregates over hidden web databases," in *SIGMOD*, 2010.

[31] A. Dasgupta, G. Das, and H. Mannila, "A random walk approach to sampling hidden databases," in *SIGMOD*, 2007.

[32] A. Dasgupta, N. Zhang, and G. Das, "Leveraging count information in sampling hidden databases," in *ICDE*, 2009.

[33] ——, "Turbo-charging hidden database samplers with overflowing queries and skew reduction," in *EDBT*, 2010.

[34] W. Liu and et.al, "Aggregate estimation over dynamic hidden web databases," in *VLDB*, 2014.

[35] T. Liu, F. Wang, and G. Agrawal, "Stratified sampling for data mining on the deep web," *FCS*, 2012.

[36] F. Wang and G. Agrawal, "Effective and efficient sampling methods for deep web aggregation queries," in *EDBT*, 2011.

[37] Z. Bar-Yossef and M. Gurevich, “Random sampling from a search engine’s corpus,” *Journal of the ACM*, vol. 55, no. 5, 2008.

[38] M. Zhang, N. Zhang, and G. Das, “Mining a search engine’s corpus: efficient yet unbiased sampling and aggregate estimation,” in *SIGMOD*, 2011.

[39] ——, “Mining a search engine’s corpus without a query pool,” in *CIKM*, 2013.

[40] A. Broder and et.al., “Estimating corpus size via queries,” in *CIKM*, 2006.

[41] M. Shokouhi and et.al., “Capturing collection size for distributed non-cooperative retrieval,” in *SIGIR*, 2006.

[42] M. F. Rahman, S. B. Suhaim, W. Liu, S. Thirumuruganathan, N. Zhang, and G. Das, “Analoc: Efficient analytics over location based services,” in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 2016, pp. 1366–1369.

[43] A. Magdy, L. Alarabi, S. Al-Harthi, M. Musleh, T. M. Ghanem, S. Ghani, S. Basalamah, and M. F. Mokbel, “Demonstration of Taghreed: A system for querying, analyzing, and visualizing geotagged microblogs,” in *ICDE*, 2015.

[44] A. Magdy, A. M. Aly, M. F. Mokbel, S. Elnikety, Y. He, and S. Nath, “Mars: Real-time spatio-temporal queries on microblogs,” in *ICDE*, 2014.

[45] W. Liu, M. F. Rahman, S. Thirumuruganathan, N. Zhang, and G. Das, “Aggregate estimations over location based services,” *PVLDB*, 2015.

[46] M. F. Rahman, W. Liu, S. B. Suhaim, S. Thirumuruganathan, N. Zhang, and G. Das, “Density based clustering over location based services,” in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 2017, pp. 461–469.

[47] W. Liu, M. F. Rahman, S. Thirumuruganathan, N. Zhang, and G. Das, “Aggregate estimations over location based services,” *PVLDB*, 2015.

[48] C. Xia, W. Hsu, M. L. Lee, and B. C. Ooi, “Border: efficient computation of boundary points,” *TKDE*, 2006.

[49] B.-Z. Qiu, F. Yue, and J.-Y. Shen, “Brim: An efficient boundary points detecting algorithm,” in *PAKDD*, 2007.

[50] M. Halkidi, Y. Batistakis, and M. Vazirgiannis, “On clustering validation techniques,” *JIIS*, vol. 17, no. 2, pp. 107–145, 2001.

[51] P. Berkhin, “A survey of clustering data mining techniques.” Springer, 2006.

[52] A. Hinneburg and D. A. Keim, “An efficient approach to clustering in large multimedia databases with noise,” in *KDD*, 1998.

[53] H. Yan, Z. Gong, N. Zhang, T. Huang, H. Zhong, and J. Wei, “Crawling hidden objects with kNN queries,” *TKDE*, 2016.

[54] G. Karypis, E.-H. Han, and V. Kumar, “Chameleon: Hierarchical clustering using dynamic modeling,” *Computer*, 1999.

[55] H. V. Jagadish, “Linear clustering of objects with multiple attributes,” in *ACM SIGMOD Record*, 1990.

[56] S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis, “Automated ranking of database query results,” in *In CIDR*. CIDR, 2003.

[57] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum, “Probabilistic ranking of database query results.” VLDB, 2004.

[58] M. F. Rahman, W. Liu, S. Thirumuruganathan, N. Zhang, and G. Das, “Privacy implications of database ranking,” *Proceedings of the VLDB Endowment*, vol. 8, no. 10, pp. 1106–1117, 2015.

[59] V. Hristidis, N. Koudas, and Y. Papakonstantinou, “Prefer: A system for the efficient execution of multi-parametric ranked queries,” in *ACM SIGMOD Record*, 2001, pp. 259–270.

[60] J. Li, B. Saha, and A. Deshpande, “A unified approach to ranking in probabilistic databases,” *VLDB*, 2009.

[61] W. Kießling, "Foundations of preferences in database systems," in *VLDB*, 2002, pp. 311–322.

[62] N. Bruno, S. Chaudhuri, and L. Gravano, "Top-k selection queries over relational databases: Mapping strategies and performance evaluation," *ACM TODS*, vol. 27, no. 2, 2002.

[63] A. Motro, "Vague: A user interface to relational databases that permits vague queries," *ACM TOIS*, vol. 6, no. 3, pp. 187–214, 1988.

[64] Y. Rui, T. S. Huang, and S. Mehrotra, "Content-based image retrieval with relevance feedback in mars," in *ICIP*, 1997.

[65] C. Farkas and S. Jajodia, "The inference problem: a survey," *ACM SIGKDD Explorations Newsletter*, vol. 4, no. 2, pp. 6–11, 2002.

[66] J. Domingo-Ferrer, "A survey of inference control methods for privacy-preserving data mining," in *Privacy-preserving data mining*. Springer, 2008, pp. 53–80.

[67] F. Chin, "Security problems on inference control for sum, max, and min queries," *JACM*, vol. 33, no. 3, pp. 451–464, 1986.

[68] "Catch22dating frequently asked questions," http://www.catch22dating.com/faq.

[69] X. Geng, T.-Y. Liu, T. Qin, A. Arnold, H. Li, and H.-Y. Shum, "Query dependent ranking using k-nearest neighbor," in *SIGIR*. ACM, 2008, pp. 115–122.

[70] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant, *Range queries in OLAP data cubes*, 1997, vol. 26, no. 2.

[71] D. R. Cox, *Principles of statistical inference*. Cambridge University Press, 2006.

[72] L. Von Ahn, M. Blum, N. J. Hopper, and J. Langford, "Captcha: Using hard ai problems for security," in *EUROCRYPT*. Springer, 2003.

[73] "Eharmony," http://www.eharmony.com.

[74] B. McFee and G. R. Lanckriet, "Metric learning to rank," in *ICML*, 2010, pp. 775–782.

[75] A. Dasgupta, G. Das, and H. Mannila, "A random walk approach to sampling hidden databases," in *SIGMOD*, 2007.

[76] A. Dasgupta, X. Jin, B. Jewell, N. Zhang, and G. Das, "Unbiased estimation of size and other aggregates over hidden web databases," in *SIGMOD*. ACM, 2010, pp. 855–866.

[77] "Renren," http://www.renren.com.

[78] "Catch22dating," http://www.catch22dating.com/.

[79] "Amazon goodreads," https://www.goodreads.com/.

[80] T. J. Green and V. Tannen, "Models for incomplete and probabilistic information," in *EDBT*, 2006.

[81] S. Thirumuruganathan, N. Zhang, and G. Das, "Rank discovery from web databases," *VLDB*, 2013.

[82] Z. Zheng, H. Zha, T. Zhang, O. Chapelle, K. Chen, and G. Sun, "A general boosting method and its application to learning ranking functions for web search," in *NIPS*, 2008.

[83] A. Podelski and A. Rybalchenko, "A complete method for the synthesis of linear ranking functions," in *Verification, Model Checking, and Abstract Interpretation*. Springer, 2004.

[84] F. Y. Chin, P. Kossowski, and S. Loh, "Efficient inference control for range sum queries," *Theoretical Computer Science*, vol. 32, no. 1, pp. 77–86, 1984.

[85] M. E. Nergiz, M. Atzori, and C. Clifton, "Hiding the presence of individuals from shared databases," in *SIGMOD*, 2007.

[86] N. R. Adam and J. C. Worthmann, "Security-control methods for statistical databases: a comparative study," *ACM Computing Surveys (CSUR)*, vol. 21, no. 4, pp. 515–556, 1989.

[87] F. Y. Chin and G. Ozsoyoglu, "Auditing and inference control in statistical databases," *TSE*, vol. 8, no. 6, pp. 574–582, 1982.

[88] R. Agrawal and R. Srikant, "Privacy-preserving data mining," *ACM Sigmod Record*, vol. 29, no. 2, pp. 439–450, 2000.

[89] C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating noise to sensitivity in private data analysis," in *Theory of Cryptography*. Springer, 2006, pp. 265–284.

[90] M. L. et al, "All your location are belong to us: Breaking mobile social networks for automated user location tracking," ser. MobiHoc, 2014.

[91] M. F. Rahman, A. Asudeh, N. Koudas, and G. Das, "Efficient computation of subspace skyline over categorical domains," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. ACM, 2017, pp. 407–416.

[92] A. Asudeh, S. Thirumuruganathan, N. Zhang, and G. Das, "Discovering the skyline of web databases," *VLDB*, 2016.

[93] A. Gupta, V. Harinarayan, and D. Quass, "Aggregate-query processing in data warehousing environments," 1995.

[94] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis, "Answering top-k queries using views," in *VLDB*, 2006.

[95] A. Y. Halevy, "Answering queries using views: A survey," *VLDBJ*, 2001.

[96] T. Preisinger and W. Kießling, "The hexagon algorithm for pareto preference queries," in *M-PREF*, 2007.

[97] R. Fagin, "Combining fuzzy information from multiple systems," in *ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, 1996.

[98] I. Bartolini, P. Ciaccia, and M. Patella, "Efficient sort-based skyline evaluation," *TODS*, vol. 33, no. 4, 2008.

[99] J. Lee and S.-W. Hwang, "Scalable skyline computation using a balanced pivot selection technique," *Information Systems*, vol. 39, 2014.

[100] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, “Skyline with presorting: Theory and optimizations,” in *IIPWM*, 2005.

[101] P. Godfrey, R. Shipley, and J. Gryz, “Maximal vector computation in large data sets,” in *VLDB*, 2005.

[102] S. Zhang, N. Mamoulis, and D. W. Cheung, “Scalable skyline computation using object-based space partitioning,” in *SIGMOD*, 2009.

[103] D. Papadias, Y. Tao, G. Fu, and B. Seeger, “An optimal and progressive algorithm for skyline queries,” in *SIGMOD*, 2003.

[104] K. C. Lee, B. Zheng, H. Li, and W.-C. Lee, “Approaching the skyline in z order,” in *VLDB*, 2007.

[105] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang, “Efficient computation of the skyline cube,” in *VLDB*, 2005.

[106] J. Pei, W. Jin, M. Ester, and Y. Tao, “Catching the best views of skyline: A semantic approach based on decisive subspaces,” in *VLDB*, 2005.

[107] J. Lee and S.-w. Hwang, “Toward efficient multidimensional subspace skyline computation,” *The VLDB Journal*, vol. 23, no. 1, pp. 129–145, 2014.

[108] Y. Tao, X. Xiao, and J. Pei, “Subsky: Efficient computation of skylines in subspaces,” in *ICDE*, 2006.

[109] T. Xia and D. Zhang, “Refreshing the sky: the compressed skycube with efficient support for frequent updates,” in *SIGMOD*, 2006.

[110] T. Xia, D. Zhang, Z. Fang, C. Chen, and J. Wang, “Online subspace skyline query processing using the compressed skycube,” *TODS*, 2012.

[111] S. Maabout, C. Ordonez, P. K. Wanko, and N. Hanusse, “Skycube materialization using the topmost skyline or functional dependencies,” *TODS*, vol. 41, no. 4, 2016.