ENABLING EXPLORATORY MINING OVER HIDDEN DATABASES

by

SARAVANAN THIRUMURUGANATHAN

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 2015

To my parents.

ACKNOWLEDGEMENTS

I would like to acknowledge and express my deepest appreciation to my supervising professor and committee chair, Dr.Gautam Das. The research in this dissertation would not have been possible without his constant support and mentoring. I have learned tremendous amount over the past few years starting from formulating a problem, fleshing it out, exploring its edges, coming up with theoretically rigorous solution, writing meticulous yet easy to read papers and all the way to presenting the idea to other researchers. His ability to see through the core of a messy research problem and coming up with easier to analyze analogies never ceases to amaze me. Over the years he has introduced me to diverse set of fascinating problems and enabled collaboration with many great researchers. His distinct style of research, teaching, mentoring and managing had a huge and positive impact on me. The last few years have been intellectually exhilarating and productive.

I am grateful to my numerous research collaborators who have introduced me to a number of interesting problems. I have learned a lot from these incredible researchers by observing their style of research. Three of them deserve special mention. Dr.Nan Zhang of George Washington University has practically been a second advisor to me from whom I have learned an incredible amount. His sheer intellect and hard work has always been an inspiration. Dr.Sihem Amer-Yahia of CNRS, LIG France has taught me the importance of always keeping the reader in mind, to design simple yet powerful models and avoid falling into the pitfall of creating overly complicated models just for the sake of complexity. Finally, Senjuti Basu Roy of University of Washington at Tacoma has been a constant mentor and provided lot of tips for me

to emerge as an independent researcher. Her hard work and persistence has inspired and motivated me to work harder!

I would like to thank my committee members Dr.Manfred Huber, Dr.Chengkai Li and Dr.Vassilis Athitsos. Dr.Manfred Huber and Dr.Gergely V. Záruba 'volunteered' me into doing my MS thesis thereby kick starting my research career. It is conceivable that I would have stopped with MS and missed all the fun if not for these wonderful mentors. I have very fond memories of the time spent in the lab discussing all the topics under the sun. Dr.Chengkai Li's courses on Data Mining and Web Mining were two of the earliest courses I took on data mining. His (extremely) detailed analysis and positive feedback for both my course projects gave me lot of confidence for doing research in data mining. Taking Dr.Vassilis Athitsos' gruelling AI-I course in the first semester of my MS gave me a template that I have regularly used for mastering other courses both in my MS and PhD. I had some wonderful internships at Microsoft Research, QCRI and Yahoo! Labs and I am grateful to my mentors for the research experiences.

I would also like to extend my gratitude to the department of CSE at UTA and Dr. Das for providing me with financial supports during my entire graduate studies. I was also fortunate to teach two graduate courses and learned a lot from student feedback. I also like to thank my lab mates Azade, Habib, Farhad and Abol. I would especially cherish the long, funny and technical discussions with Azade. Many thanks to my friends Mahashweta, Rasool, Nandish, Kruthi, Naffi, Afroza, Shrikant, Weimo and Sofiane for making my PhD memorable.

Finally, I would like to convey my heartfelt gratitude to my parents, brother and my vast extended family. Their constant encouragement and support have been critical to my success.

July 28, 2015

ABSTRACT

ENABLING EXPLORATORY MINING OVER HIDDEN DATABASES

SARAVANAN THIRUMURUGANATHAN, Ph.D.

The University of Texas at Arlington, 2015

Supervising Professor: Gautam Das

Almost all popular websites (such as Amazon, EBay, microblogs such as Twitter, Instagram, collaborative content sites such as IMDB, Yelp etc) are powered internally by large data repositories. We designate them as *hidden databases* as their underlying data is accessible only through proprietary form-like interfaces that require users to query the system by entering desired values for a few attributes. Further, these web databases also impose a number of restrictions. For example, the *top-k output constraint* ensures that when there are a large number of tuples matching the query, only a few of them (top-k) are preferentially selected and returned by the website, often according to a proprietary ranking function. The *rate limit constraint* restricts the number of queries/API calls that could be issued per day. The rank information of low ranked tuples not in top-k are often not provided due to *rank constraint*. Most microblogging platforms such as Twitter also enforce *recency constraint* that limits the results of their APIs to recent data. This stymies the efforts to perform analytics over historic data. Finally, most collaborative content sites provide only aggregate information over items such as number of likes, average rating etc instead of granular information needed for mining. These restrictions prevent scientists with

limited resources from performing novel analytics tasks. Similarly, it also prevents third parties from building innovative services over these data.

Most prior work on exploratory analysis and mining are not applicable for hidden databases due to the aforementioned restrictions. In this dissertation, we present efficient techniques for enabling exploratory mining over hidden databases. This is achieved by developing novel algorithms that allows a third party (such as an analyst or a scientist) to retrieve relevant data from hidden databases for exploratory mining by issuing a small number of carefully constructed queries that enables them to work around the restrictions. We design algorithms that sidestep the top-k output constraint so that it is possible to retrieve the top- h tuples where $h > k$. In order to work around rank constraint, we designed statistical estimators that can estimate the rank of a given tuple which works well for both high and lowly ranked tuples. For microblog platforms, we design algorithms that allows users to perform aggregate estimation over historic content thereby circumventing the recency constraint. Finally, we propose a novel featureset uncertainty model and algorithms that can enable exploratory mining over coarse aggregate user feedback data. For all the problems, we provide rigorous theoretical analysis and extensive experiments over real-world data and online experiments over popular hidden web databases.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

Introduction

1.1 Hidden Databases

The vast majority of data available in the internet is part of the so called "deep web" that is hundreds or thousands of times larger than the surface web that is crawled by search engines[1]. Almost all popular websites (such as Amazon, EBay, American Airlines, microblogs such as Twitter, Instagram, collaborative content sites such as IMDB, Yelp etc) are powered internally by large data repositories. Often, their content is not available to search engines. Most of these websites treat the data as proprietary and a key competitive advantage for ensuring customer retention. Hence negotiations with data owners to establish data-access channels often end up failing. We designate these data repositories as *hidden databases*.

Unlike traditional databases, they do not allow a user to issue arbitrary queries using SQL or other query languages. Instead, they allow external users to browse the databases in a controlled manner. In other words, many web databases are "hidden" behind (i.e., only accessible via) a restrictive interface that allows a user to form a search query by specifying the desired values for a few attributes. These interfaces can be form-based, keyword based, API based or in the case of microblogs, neighborhood based. Users often issue a query by entering desired values for a few attributes and the system responds by returning a small number of tuples matching the search query.

In addition, the hidden databases enforce a number of constraints to prevent users from gleaning more data than needed. We now describe some of the constraints most commonly enforced.

Top-k Output Constraint: Almost all such interfaces enforce the *top-k output constraint* - i.e., when more than k tuples (where k is typically a predetermined small constant) match the user-specified query, only k of them are preferentially selected according to a (often proprietary) ranking function and returned to the user. For example, American Airline's (AA) flight search-by-schedule [1] has a default value of $k = 10$. Similarly, Amazon's best sellers list [2] for any category only displays the top-100 products.

Rate Limit Constraint: Most of the web databases also enforce strict *rate limit constraints* by imposing a per user/IP limit on number of queries one can issue over a given time frame For e.g., Google Search API allows only 100 free queries per user per day. Twitter allows only 180 queries per 15 minutes. Reddit allows no more than one request every two seconds and so on.

Rank Constraint: Another popular restriction is the *rank constraint* whereby most web databases do not explicitly disclose a tuple's rank beyond the top-k tuples. For example, Amazon advertises top-100 products in a given category[3]. However, if a product has a rank higher than 100 then its rank is not revealed. A similar phenomenon occurs in a number of other data sources such as Apple AppStore, Google Play etc.

Recency Constraint: This constraint is often enforced in microblogs and collaborative content sites. The search interface over these databases provide preferential treatment to recent items. For example, Twitter search API limits its result to tweets from the last few weeks. This constraint stymies the efforts to perform analytics over historic data.

[1] *http://www.aa.com/reservation/searchFlightsSubmit.do*. By default $k = 10$. A user may configure k to be as large as 50. No page down is allowed.

[2] *http://www.amazon.com/Best-Sellers/zgbs*

[3] *http://www.amazon.com/Best-Sellers/zgbs*

Data Access Constraint: A number of collaborative content sites (where users voluntarily provide feedback over items) only provide access to coarse aggregate data. For example, IMDB might only provide information such as average rating for a movie. Instagram might only provide the number of views for an image. Facebook might provide only the total number of likes received by an item. However, granular data that provides additional context is often much more useful for analytics purpose.

1.2 Motivation and Challenges

While hidden web databases and microblogs contain very interesting information, the various data access barriers imposed prevent novel applications from being built. For example, while the restrictive form interfaces of hidden databases might suffice for the simplest use-cases, i.e., that of a normal user searching for some items in these databases, they often cannot satisfy users with specific needs and also prevent many interesting third-party services from being developed over web databases.

Consider the top-k output constraint. There exist legitimate reasons for setting a small value for k such as speeding up query processing and thwarting web scraping. However, in order to accommodate the needs of website users, the value of k should not be too small. Given these two conflicting goals, in practice k is often set to the minimum necessary value, according to the database owner's belief, which provides the user with "enough" choices within the returned tuples. However, this strategy is misguided and is often insufficient for all but the simplest use cases. It often cannot satisfy users with specific needs and also prevents many interesting third-party services from being developed.

Consider a third-party service which enables a user to filter query results according to attributes that cannot be specified in the original form-like interface. For

example, American Airline's (AA) flight search-by-schedule[4], a top-10 interface, does not allow a user to specify filtering conditions such as finding the top-10 flights with in-flight wifi. If a third-party service wants to provide such a feature, it must somehow "bypass" the top-k constraint because otherwise one might not be able to find enough (or any) wifi-equipped flights from the top-10 results.

Circumventing the rank constraint also has a number of applications. For example, the author of a book on sale at Amazon would be keen to track and monitor the ranking of her book within a set of similar competitors (e.g., how does it rank in sales, or customer reviews, etc., compared to other similar books on science fiction?). Likewise, competitors to an app available at Apple's iOS and Mac App Stores would be interested in monitoring the app's grossing rank and measure the market response to determine if it is time to start competing with the app. Investors may be keen to do simultaneous monitoring of numerous products newly released by competing companies, in order to determine which one is likely to be a hit and which company to invest in.

Microblogging platforms provide free (but limited) public access to their data in the form of restricted APIs, which offer great opportunities for non-commercial applications, such as the type of studies that would be most useful to a social scientist. However, many of them require access to historic data. For example, a social researcher may wish to analyze publicly available microblog conversations and postings to determine the change in general public's attitudes on individual privacy before and after the news of Edward Snowden's leakage of NSA surveillance became public. Other examples can include studies of the spread of obesity-promoting attitudes,

[4] *http://www.aa.com/reservation/searchFlightsSubmit.do* By default $k = 10$. A user may configure k to be as large as 50. No page down is allowed.

the mechanisms of bullying in colleges or schools, and the early detection of suicidal discourse.

1.3 Dissertation Overview and Impact

All of these interesting applications and other exploratory mining techniques is often not feasible due to the data access barriers constructed by the owners of the hidden databases. In this dissertation, we present efficient techniques for enabling exploratory mining over hidden databases. This is achieved by developing novel algorithms that allows a third party (such as an analyst or a scientist) to retrieve relevant data from hidden databases for exploratory mining by issuing a small number of carefully constructed queries that enables them to work around the restrictions. We would like to note that our objective is to use the existing hidden database interface and operate under the restrictions they impose (such as top-k output constraint) while seeking to retrieve relevant data that is needed for exploratory mining. In other words, we do not, in any way, seek to modify the input interfaces or break the restrictions by other nefarious means. We also face additional challenges such as minimizing the number of queries issued (due to the rate limit constraint).

There has been several recent works on developing techniques to enable additional functionality over such databases that operate via the restrictive interface, such as *sampling* and aggregate estimation (see [2, 3, 4] and references therein). These techniques try to obtain a big-picture view of the hidden database from a small sample by using sophisticated statistical estimators. Sampling over microblogs such as Twitter, was an open problem, that required non-trivial adaptation of sampling techniques for social networks. At the other end of the spectrum is *crawling* where the objective is to retrieve all tuples from hidden database locally. Lower-bound results derived in [5] show that crawling requires a prohibitively high cost of at least $\Omega(m \cdot n^2/k^2)$

queries for certain categorical databases with a top-k interface - where m and n are the number of attributes and tuples, respectively.

However, none of these techniques are appropriate for the applications that we described previously. Circumventing the data access barriers to enable exploratory mining require development of novel techniques many of which occupy a niche space in the spectrum between sampling and crawling. We now provide a high level overview of our contributions:

1. **Ranked Retrieval:** Given a hidden database that enforces top-k output constraint, our first contribution is a GetNext operator that could retrieve the top ranked tuples from a hidden database in an ordered manner. Specifically, given the top-h tuples (where $h > k$), the GetNext operator could efficiently retrieve the tuple ranked $h + 1$. By calling GetNext iteratively, it is possible to retrieve as many top-ranked tuples as necessary for a user-specified query - thereby enabling both sample applications discussed above without the need of crawling all tuples from the database. Because of the query-number limitations enforced by web databases, an important objective in the design of GetNext is to maintain a small *query cost* - a goal shared by most existing studies on exploring hidden web databases (e.g., [3, 6, 7]).
2. **Rank Discovery:** The rank is a fundamental property of an item since it determines the item's visibility. We circumvent the rank constraint of hidden database by designing statistical estimators that, given a query q and a tuple t that satisfies q, estimates $rank(t, q)$, the rank of t among all tuples in the database that satisfy q. Our estimators work well for both high and lowly ranked tuples.

   Although the rank discovery problem, as stated above, appears deceptively compact and simple, it is challenging because most web databases do not explicitly

disclose a tuple's rank beyond the top-k tuples. The rank has to be discovered indirectly, by carefully issuing multiple related queries to the website's query interface and recovering/inferring the tuple rank by piecing together the information returned from these queries. Moreover, our investigations revealed that different websites have widely varying characteristics, resulting in a myriad of interesting facets and variants of the rank discovery problem that require fundamentally different approaches in their solutions.

3. **Aggregate Estimation over Historic Microblog Data:** Given a microblog platform that does not provide direct access to historic data, we seek to design an algorithm to obtain samples that can be used to answer interesting statistical investigations over the historic data. We also seek to design an estimator for performing aggregate analytics over historic data. A core functionality to facilitate such analytics is to answer *aggregate queries* over publicly available microblog data, which is the focus of this paper. An example of aggregate query is "How many Twitter users used the keyword `privacy` in 2013?". We shall consider SUM, COUNT, AVG queries on various attributes of microblog users or posts (e.g., users' age or posts' length), with selection conditions on keywords and other attributes like time. This would also require the design of a subgraph of the underlying social network that is more conducive to sampling.
4. **Exploratory Mining over Coarse User-Item Interactions:** Recently, it has become easy for businesses to put Facebook "Like" buttons[5] or "+1" buttons[6] from Google over their item webpages which can be clicked by users to demonstrate they like an item. While the various user-item interactions such

[5]http://developers.facebook.com/docs/reference/plugins/like/

[6]https://developers.google.com/+/web/+1button/

as visits, likes, +1s and ratings provide a rich window into *what* users like, answering the question of *why* a user likes the items is much trickier.

If each user had provided elaborate comments or reviews detailing why she liked the item, then identifying the most important features can be solved by extracting semantic information using text mining and information extraction [8, 9]. However, online users are prone to be laconic - the fraction of users that provide detailed comments is minuscule [10]. Hence, the vast majority of user-item interaction information is in its *most rudimentary form* - where we only know whether users visited or liked an item or not. This makes the problem of mining user-item interactions with rudimentary information to be very important.

Given a database with structured items and user-item interactions, we seek to provide additional probabilistic context about the interactions. Specifically, we propose a novel featureset uncertainty model and algorithms that can enable exploratory mining over coarse aggregate user feedback data.

1.4 Dissertation Organization

In Section 2, we consider the novel problem of "ranked retrieval" over hidden web databases. Our key contribution is the meta-algorithm GetNext that can retrieve the next ranked tuple from the hidden web database using only the restrictive interface of a web database without any prior knowledge of its ranking function. This algorithm can then be called iteratively to retrieve as many top ranked tuples as necessary. We develop principled and efficient algorithms that are based on generating and executing multiple reformulated queries and inferring the next ranked tuple from their returned results. We provide theoretical analysis of our algorithms, as well as

extensive experimental results over synthetic and real-world databases that illustrate the effectiveness of our techniques.

In Section 3, we investigate the problem of "rank discovery" over hidden databases. To address the problem, we introduce a taxonomy of ranking functions, and show that different types of ranking functions require fundamentally different approaches for rank discovery. Our technical contributions include principled and efficient randomized algorithms for estimating the rank of a given tuple, as well as negative results which demonstrate the inefficiency of any deterministic algorithm. We show extensive experimental results over real-world databases, including an online experiment at Amazon.com, which illustrates the effectiveness of our proposed techniques.

Microblogging platforms such as Twitter have experienced a phenomenal growth of popularity in recent years, making them attractive platforms for research in diverse fields from computer science to sociology. Twitter, for example has more than 200 million active users who generate over 400 million tweets every day, In Section 4, we consider a novel problem of estimating aggregate queries over microblogs, e.g., "how many users mentioned the word 'privacy' in 2013?". We propose novel solutions exploiting the user-timeline information that is publicly available in most microblogging platforms. Theoretical analysis and extensive real-world experiments over Twitter, Google+ and Tumblr confirm the effectiveness of our proposed techniques.

Finally, in Section 5 we consider the problem of hidden featureset mining. Given user-item transactions over a database with structured items, our objective is to identify the frequent featuresets (set of features) by mining item transactions. We propose a featureset uncertainty model where each item transaction could have been generated by various featuresets with different probabilities. We describe a novel approach to transform item transactions into uncertain transaction over featuresets and estimate their probabilities and propose diverse algorithms to mine frequent featuresets. Our

experimental evaluation provides a comparative analysis of the different approaches proposed.

CHAPTER 2

Ranked Retrieval over Hidden Databases

2.1 Introduction

Many web databases are "hidden" behind (i.e., only accessible via) a restrictive form-like interface which allows a user to form a search query by specifying the desired values for a few attributes; and the system responds by returning a small number of tuples matching the search query. Almost all such interfaces enforce the *top-k constraint* - i.e., when more than k tuples (where k is typically a predetermined small constant) match the user-specified query, only k of them are preferentially selected according to a (often proprietary) ranking function and returned to the user. For example, American Airline's (AA) flight search-by-schedule[1] has a default value of $k = 10$. Similarly, Amazon's best sellers list [2] for any category only displays the top-100 products.

How to properly set the value of k is an interesting design challenge for a web database owner. On one hand, the owner may prefer a small k to (1) speed up query processing and shorten the returned webpage, and/or (2) thwart web/tuple scraping. However, in order to accommodate the needs of website users, the value of k should not be too small. Given these two conflicting goals, in practice k is often set to the minimum necessary value, according to the database owner's belief, which provides the user with "enough" choices within the returned tuples. While such a strategy might suffice the simplest use-cases, it often cannot satisfy users with specific needs

[1] *http://www.aa.com/reservation/searchFlightsSubmit.do* By default $k = 10$. A user may configure k to be as large as 50. No page down is allowed.

[2] *http://www.amazon.com/Best-Sellers/zgbs*

and also prevents many interesting third-party services from being developed over web databases - e.g.,

- Consider a third-party service which enables a user to filter query results according to attributes that cannot be specified in the original form-like interface. For example, American Airline's (AA) flight search-by-schedule[1], a top-10 interface, does not allow a user to specify filtering conditions such as finding the top-10 flights with in-flight wifi. If a third-party service wants to provide such a feature, it must somehow "bypass" the top-k constraint because otherwise one might not be able to find enough (or any) wifi-equipped flights from the top-10 results.
- Consider a web aggregator or a web mashup which joins tuples from multiple hidden web databases and returns the joined results - e.g., a mashup joining Orbitz.com (a hotel booking website) with Tripadvisor.com (a hotel review website) to return the top-k cheapest hotels that have an average review of at least 4 stars. Once again, such a mashup must somehow break the top-k constraint because not enough matching tuples may be discovered from the mere k tuples returned by each web database.

To enable these third-party services and many other interesting applications (e.g., data analytics) that are currently disabled/handicapped by the top-k constraint, a trivial solution is for the third-party service provider to negotiate a private agreement with each web database owner in order to establish data-access channels beyond the top-k web interface. Nonetheless, such negotiations are difficult even between large organizations[3] due to revenue sharing, security and myriad of other thorny issues - thus making the solution not scalable to a large number of web databases. As such, our focus is to develop automated third-party algorithms that only use the

[3] *http://online.wsj.com/article/SB121755825030403467.html*

public interfaces of web databases without requiring any additional cooperation from the database owners.

Another seemingly straightforward solution to the above problems is *crawling* - i.e., the retrieval of *all* tuples in a hidden web database by issuing multiple queries through its web interface [11, 12]. Once all tuples are downloaded, they can be treated as a local database to support all of the above applications. Nonetheless, a key pitfall of this solution is its prohibitively high query cost (i.e., the numerous search queries one needs to crawl all tuples from a web database) - which can be simply infeasible for real-world web databases which often impose a per user/IP limit on number of queries one can issue over a given time frame (e.g., Google Search API allows only 100 free queries per user per day).

2.1.1 Breaking the Top-k Barrier

Given the pitfalls of crawling, we propose to study a novel problem of digging deeper into a web database to retrieve (more than k) top-ranked tuples which satisfy a user-specified search query - and thereby "breaking" the top-k barrier. Specifically, we consider the following fundamental operator:

> GetNext: Given the top-h tuples ($h \geq k$) satisfying a user-specified query, retrieve the next-highest-ranked (i.e., No.$(h+1)$) tuple from the hidden web database by issuing search queries through its public interface, without any knowledge of its ranking function.

One can see that, by calling GetNext iteratively, it is possible to retrieve as many top-ranked tuples as necessary for a user-specified query - thereby enabling both sample applications discussed above without the need of crawling all tuples from the database. Because of the query-number limitations enforced by web databases, an important objective in the design of GetNext is to maintain a small *query cost*

- a goal shared by most existing studies on exploring hidden web databases (e.g., [3, 6, 7]).

It is important to understand that the most critical efficiency measure here is the *number of queries* the algorithm requires to issue through the web interface of the hidden database, *not* the actual processing time of issued queries and/or the local processing overhead. To understand why, note that our objective is to build a scalable third-party service that breaks the top-k barrier for a large number of users. A key obstacle for building such a service, as mentioned earlier, is the query allowance enforced by the hidden database owner - which completely shuts down our access to the database if we issue more queries than a pre-determined threshold. Compared with this hard cutoff, the variance of local processing overhead and/or query processing time is relatively negligible - especially given the fact that most well-designed hidden databases return query answers within a short period of time.

In summary, we have the following problem definition

Breaking the top-k barrier: *For a hidden web database with a top-k interface, given a search query q that can be specified through the input interface, design an automated procedure GetNext that can be iteratively called to retrieve as many top ranked tuples that satisfy q as possible (according to the ranking function used by the hidden web database).*

To the best of our knowledge, this novel problem has not been considered in prior research.

The parameter h also offers its own challenges. For what values of h is it actually feasible to retrieve the top-h tuples? In the extreme case, can one retrieve the entire global order of all tuples in the database, if such an order exists? In this work we determine necessary and sufficient conditions that will enable solutions for our central

problem, and the implications of these conditions on the different problem variants. For example, for certain cases it is simply not possible for any client-side algorithm to determine a total order of the top-h tuples - in this case we develop algorithms that return the most informative partial order.

2.1.2 Outline of Technical Results

To design GetNext, the technical challenge may have subtle differences across various web databases, mainly because of the different ranking functions being used. At one extreme, some websites allow users to choose their own ranking function (from a predetermined set) - e.g., airlines websites allow users to sort by attributes such as by price, departure time, etc. At the other extreme, a website might feature a complex and proprietary *query-specific* ranking function (e.g., "relevance" of a tuple to a query) that may never be deterministically inferred from other query answers. Other possible ranking functions include a global order that is nevertheless hidden from the input interface - e.g., Amazon uses popularity as the default ranking function but does not allow it to be specified in a search query. For most of this work, we focus on the case where the ranking function is a query-independent global order of all tuples. The implications of other ranking-function variations on our solutions are discussed separately.

There are two key components of our proposed solution to GetNext: *candidate generation* and *candidate testing*.

Candidate Generation: Given the top-h tuples, the candidate generation step aims to identify a complete yet small set of tuples that can potentially have the rank $h+1$. A key observation here is that the problem is equivalent to finding a small set of queries, each of which matches fewer than k tuples in the top-h, while together cover the rest of the database. One can see that, since each query in the set returns at least

one non-top-h tuples, the No.$(h+1)$ tuple must be returned by at least one query in the set. Based on this key observation, we propose a tuple-chain-construction based technique which further reduces the query cost required for candidate generation significantly.

Candidate Testing: Since the task is now reduced to testing which candidate is the No.$(h+1)$ tuple, the key enabling question becomes how to perform pairwise rank-comparison between two tuples. Interestingly, for certain pairs of tuples, the comparison may be done with a single query to the hidden database. Specifically, consider issuing the most specific query that matches both tuples. If both are returned, then the result reveals their order. If only one is returned, then it must have a higher rank. The challenge, however, is in the worst-case scenario where neither is returned. We start by resolving this scenario with a baseline approach that requires 2^m queries, where m is the number of attributes. Then, we propose two ideas - one connects with the well-studied problem of *minimal infrequent itemsets* mining [13], and the other is a heuristic of query-result inference - which significantly reduce the query cost for candidate testing.

In summary, the main contributions of this work are as follows. We introduce the novel problem of breaking the top-k barrier of a hidden web database to retrieve top ranked tuples that match a user query. We consider several variants of the problem, and study necessary and sufficient conditions under which this problem can be solved. We propose BEYOND-h-GETNEXT and ORDERED-GETNEXT, two algorithms that iteratively uses the two fundamental operations, candidate generation and candidate testing, to retrieve the next-highest-ranked tuple. While BEYOND-h-GETNEXT guarantees the correct retrieval of next ranked tuple[4], ORDERED-GETNEXT further uses an effective heuristic of query-result inference to significantly

[4]if such an order can be uniquely determined from the top-k interface.

reduce the query cost in practice without sacrificing correctness. Our main idea for candidate generation hinges upon the fact that the returned results of a query gives ordering information for k tuples, which allows us to retrieve multiple consecutive ranked tuples in a single invocation. Our main ideas for candidate testing avoids the need to perform a complete crawl of the database. We develop the procedure BEYOND-h-TEST whose performance is related to the problem of generating and counting the number of infrequent minimal infrequent itemsets in a database. We also develop the procedure ORDERED-TEST, based on heuristics that order the execution of the generated queries in such a way that early terminating of the procedure can be achieved. Our contributions also include a careful theoretical analysis of BEYOND-h-GETNEXT and ORDERED-GETNEXT, as well as a through experimental evaluation over both synthetic datasets and real-world websites.

2.2 Preliminaries

In this section, we introduce a model for hidden databases and describe the different types of ranking functions used commonly in hidden databases.

2.2.1 Model of Hidden Databases

Consider a hidden database D with n tuples and m input attributes $A_1, A_2, \ldots, A_m$. Given a tuple t and attribute A_i, let $t[A_i]$ be the value of A_i in t. Let $Dom(A_i)$ be the domain of A_i. For the purpose of this paper, we restrict our attention to categorical attributes and assume the appropriate discretization of numeric ones. We also consider all tuples distinct and without null values. Let $f(.)$ be the ranking function which takes a tuple and a query as input and outputs an integer between 1 and n. Without loss of generality, we assume the output of $f(.)$ to be unique for each tuple.

A user can query the system by specifying the desired values for a subset of $A_1, \ldots, A_m$. Thus, a user query q is of the form `SELECT * FROM` D `WHERE` $A_{i_1} = v_{i_1} \& \ldots \& A_{i_s} = v_{i_s}$, where $\{i_1, \ldots, i_s\} \subseteq [1, m]$. and $v_{i_j} \in Dom(A_{i_j})$. The set of tuples matching query q is denoted as $Sel(q)$. If $|Sel(q)| > k$, an $overflow$ occurs and only the top-k results are returned, along an overflow flag indicating that more tuples matching the query cannot be returned. If $|Sel(q)| = 0$, then an *underflow* occurs as no tuples match the query. Otherwise, i.e., when $|Sel(q)| \in [1, k]$, we say that q is valid. For the purpose of this paper, we make the realistic assumption that $k > 1$.

For the purpose of our paper, we assume that the interface only displays the top-k results and does not allow users to extract additional results by scrolling through the results. The only way to get additional results is to reformulate the input query. This is a reasonable assumption as many real world hidden web databases such as Yahoo! Autos limit the maximum number of page turns a user can perform.

2.2.2 Model of Ranking Function

There are two broad categories of ranking functions: *static* and *query-dependent*.

- A ranking function $f(.)$ is *static* if for a given tuple t, $f(q, t)$ is constant for all queries q - i.e., the rank of a tuple is independent of the query being issued. An example in practice is the "sort by price" used by Yahoo! Autos. Note that the input tuple may feature not only $A_1, \ldots, A_m$ but also the non-input-specifiable attributes (e.g., "popularity" as discussed before).
- A ranking function is *query-dependent* if, for a given t, $f(q, t)$ varies for different queries q. An example of such a ranking function occurs in a fuzzy-matching scenario where all tuples are ordered according to the number of attribute matches between the query and each tuple.

We focus on static ranking functions in this paper. The reason for doing so is simple - if the ranking function is query dependent, no mechanism can be used to fetch the next ranked tuple. To understand why, note that in order to get tuples beyond top-k, it is necessary to reformulate the query. But this has the side effect of arbitrarily changing the ranking of tuples. Hence, with a query-dependent ranking function, no mechanism can guarantee the discovery of tuples with rank greater than k for a given query.

We focus on static ranking functions in this paper. The reason for doing so is simple. We assume that there exists a global order between the top ranked tuples. Consider an arbitrary query-dependent ranking function $f(q, t)$. To correctly retrieve the top ranked tuples, the function $f(q, t)$ must return the tuples in the same global order for any query q. In other words, if for any two tuples u and v and two arbitrary queries q and q', if $f(q, u) > f(q, v)$ and $f(q', u) < f(q', v)$, then no such global order can exist. However, if there exist a global order of tuples, then the order holds irrespective of the query q which makes the ranking function $f(q, t)$ to be *static*.

For the purpose of this paper, we conservatively assume that the ranking function is unbeknown to our algorithm. If the ranking function is known and is based on the attributes returned by the hidden web interface (such as sort by price), it is possible to leverage this information to design algorithms with significantly less query cost. We further discuss this variant in Section 2.5. In addition, we assume that it is possible to infer a unique global order of the top-ranked tuples to be extracted from the web interface. If such an order cannot be inferred from the interface, one of the possible partial orders would be returned, as we shall explain in Section 2.6.

Table 2.1. GetNext: Database D used in Running Example

	A_1	A_2	A_3	A_4	A_5
t_1	0	0	0	0	1
t_2	0	0	0	1	1
t_3	0	0	1	0	1
t_4	0	1	1	1	1
t_5	1	1	1	0	1
t_6	1	1	1	1	1
t_7	0	0	0	0	0

Running Example: Table I shows a simple table which we shall use as running example throughout this paper. There are $m = 5$ Boolean attributes and $n = 7$ tuples which are ranked in the order given in the table. i.e., t_1 is the highest ranked tuple.

2.3 Overview of GetNext

In this section, we first discuss the technical challenges of GetNext, and then outline the structure of our proposed two-step solution - the details of each step shall then be developed in the next two sections, respectively.

2.3.1 Problem Definition

Recall from Section 2.2 that our aim to design GetNext operator that enables the retrieval of $h + 1$-th tuple for any $h > k$ from a hidden web database D which only exposes the top-k ranked tuples in the query answers it returns through the restrictive web interface. The hidden database does not disclose the ranking function it uses. The GetNext operator issues (a carefully selected set of) multiple queries and then infers the $h + 1$-th tuple based on the ordering of returned tuples exposed by multiple query answers. Formally, the problem can be stated as follows.

Problem Statement : *Given a hidden web database with a form based input interface that restricts user access to top-k tuples ordered by an unknown static ranking function, discover the $h+1$ ranked tuple for any $h > k$ while minimizing the query cost.*

2.3.2 Technical Challenges

To illustrate the main technical challenges, we consider a fundamental question: Given two tuples t and t', how can we determine which one ranks higher? We start with a straightforward comparison - i.e., when t and t' match the same query q which returns at least one of the two tuples:

- if q returns t but not t', then t is ranked higher,
- if q returns t' but not t, then t' is ranked higher, or
- if q returns both, then we can make the comparison based on the returned order.

In this case, we call two tuples *directly comparable*, with the higher-ranked tuple *directly dominating* the other one - i.e.,

Definition 1. [**Domination**] *A tuple t is said to **directly dominate** another tuple t', i.e., $t \succ t'$, if and only if t and t' are directly comparable and t ranks higher than t'.*

A tuple can dominate another tuple directly or indirectly. Suppose tuple $t \succ u$ and $u \succ v$. Even if t and v are not directly comparable, we can infer that t indirectly dominates v. By default, we use the term domination to refer to direct domination.

For example, consider the running example with a top-2 interface. We can observe that t_1 and t_3 are directly comparable using the query q_1: `SELECT * FROM D WHERE` $A_1 = 0$ `AND` $A_2 = 0$ `AND` $A_4 = 0$ `AND` $A_5 = 1$ with t_1 ranked higher than t_3. Similarly, tuples t_2 and t_3 are directly comparable using the query q_2: `SELECT`

* FROM D WHERE $A_1 = 0$ AND $A_2 = 0$ AND $A_5 = 1$. The result includes t_2 but not t_3 - i.e., t_2 ranks higher.

A key observation here is that if two tuples are directly comparable, then we need only one query to determine their domination relationship: the *most specific* query which matches both tuples - i.e., the query which contains one predicate for each attribute on which both tuples share the same value. To understand why, note that if this query cannot return at least one of the two tuples, then no other query can - i.e., the two tuples are not directly comparable. For the running example, both q_1 and q_2 shown above are the most specific queries matching the two corresponding tuples.

While the possibility of direct comparison shows promises for ranking tuples in the database, it also illustrates the key technical challenge for GetNext: not every pair of tuples are directly comparable with each other - e.g., neither t_6 nor t_7 in the running example can be returned by the most specific query that matches both of them (i.e., SELECT *).

In this case, the comparison of the two tuples requires one to identify a "bridge" of tuples between them - e.g., $t \succ t_x \succ t'$ for comparing t with t'. The problem, however, is it is unclear how one can find the bridging tuples without actually crawling all tuples from the database and incurring a prohibitively high query cost. In the next subsection, we outline the structure of our proposed solution to address this challenge.

A graphical way to represent the comparability of tuples is by using a directed acyclic graph (DAG). The nodes correspond to tuples and a directed edge exists between two nodes u and v if the respective tuples are directly comparable and u has a higher rank than v. Even if two tuples are directly comparable, their relative rank can still be inferred if a directed path exists between the corresponding nodes in the graph. In this case, the tuple corresponding to the start node has a higher rank than

the one for end node. The DAG can be filled by performing pairwise comparison between the respective tuples. If the tuples are not directly comparable, then no edges exist between them. The simplest DAG that can be constructed is that of a chain where each element is atleast directly comparable to its neighbors.

2.3.3 Outline of Our Proposed Solution

Our proposed solution for GetNext is a two-step process:

- *Candidate Generation:* In this step, we identify a small set of *candidate tuples* which are guaranteed to contain the No. $h+1$ tuple. If the output set has a size of 1, then we can directly output the No. $h+1$ tuple. Otherwise, we call the following candidate testing step. Section 2.4 describes our design for candidate generation.
- *Candidate Testing:* In this step, we take the set of candidate tuples as input and compare between them to determine which tuple is indeed the No. $h+1$. Section 2.5 describes our design for candidate testing.

2.4 Candidate Generation

Given the current set of top ranked tuples, the candidate generation step is supposed to produce a set of candidate tuples, one of which is guaranteed to be the next ranked tuple in the database. In subsection 2.6.3, we discuss finding the next ranked tuple matching a selectivity constraint. The determination of the exact next-ranked tuple from the candidate set is done using the candidate testing oracle described in Section 2.5. In this section, we first describe a baseline approach for candidate generation, and then introduce a more efficient algorithm using a notion of directed acyclic graphs (DAG) of tuples. The DAG based algorithm exploits the ordering information provided by query answers to potentially complete multiple

rounds of candidate generation in a single iteration (i.e., it may answer multiple consecutive GetNext calls without additional query cost). Recall that we make the realistic assumption of $k > 1$.

2.4.1 Baseline Approach

The essence of candidate generation can be stated as follows. Given the top-h tuples, candidate generation needs to identify a set of queries that is guaranteed to "cover" (i.e., return) the next-ranked (i.e., No.$(h+1)$) tuple. One can see that such a set of queries must together match all possible tuples in the database - in order to ensure that no other tuple has a higher rank than the next-ranked tuple being covered.

We start by considering a simple baseline approach as follows: First, find a set of attributes $\mathcal{A}$ such that if we partition the top-h tuples based on their value combinations for attributes in $\mathcal{A}$, then each partition contains fewer than k elements. Since each tuple is unique, such an $\mathcal{A}$ already exists. After finding $\mathcal{A} = \{A_{i_1}, \ldots, A_{i_j}\}$, we construct queries of the form q_i: `SELECT * FROM D WHERE` $A_{i_1} = v_{i_1}$ AND $\cdots$ AND $A_{i_j} = v_{i_j}$ for all possible value combinations of $v_{i_1} \in Dom(A_{i_1}), \ldots, v_{i_j} \in Dom(A_{i_j})$, and execute all such queries. One can see that these queries completely cover the database domain and thus return a candidate set for the No.$(h+1)$ tuple. To understand why, note that the No.$(h+1)$ tuple must be returned by one of the queries issued, because otherwise the query which matches the No.$(h+1)$ tuple must return a tuple that directly dominates the No.$(h+1)$ tuple.

Example 1: Given the top-3 tuples in the running example, suppose we want to retrieve the next ranked tuple. We identify an attribute, say A_3 (or A_4), such that the number of tuples having the values 0 and 1 are less than $k = 3$. We execute two queries by augmenting q - specifically, q_1: `SELECT * FROM D WHERE` $A_3 = 0$ returns

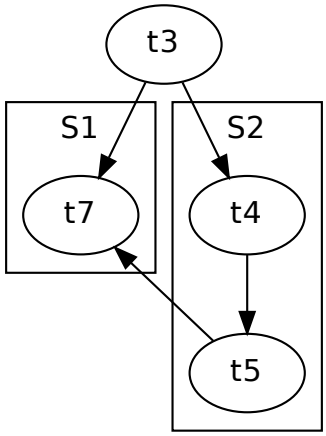


Figure 2.1. GetNext: Representing Tuple Domination as a DAG.

new tuples $\{t_7\}$ and q_2: `SELECT * FROM D WHERE` $A_3 = 1$ returns new tuples $\{t_4, t_5\}$. The candidate set for 4-th ranked tuple is the set $\{t_4, t_5, t_7\}$. If we want to retrieve the 5-th ranked tuple, we can choose any of the attributes A_2, A_3 or A_4 to partition the top-4 tuples.

Analysis: The number of queries executed to identify the candidate set depends on the domain of the attribute(s) selected. Given an attribute set $\mathcal{A}$, the number of queries executed is $\prod_{A \in \mathcal{A}} |Dom(A)|$.

2.4.2 DAG based Approach

In this subsection, we develop a DAG-based algorithm which leverages the order information provided in the query results to further reduce the number of returned candidate tuples, and to identify the candidate sets for multiple next-ranked tuples at a single iteration. In other words, our DAG based approach retrieves the candidate sets for as many next ranked tuples as possible so that subsequent *GetNext* do not incur any additional query cost.

The data structure used in our approach is a directed acyclic graph (DAG) called the *dominance directed graph*. Each node in the DAG correspond to a tuple and a directed edge exist from node u to node v if u dominates v. Given the result of any query q, we can form an DAG from it results. If the query returned $|q|$ tuples, then the DAG would have at most $\binom{|q|}{2}$ edges and an linear chain of $|q|$ tuples as a subgraph. An example of the DAG formed from queries q_1 and q_2 from Example 1 is in Figure 2.1. Given a set of queries q_i, we can form a set of linear chains from their results. Let S_i denote the i-th linear chain and S be the set of all linear chains. The notation $head(S_i)$ returns the tuple with highest rank in S_i while $head(S)$ returns the set of highest ranked tuples in each chain.

The primary aim of this approach is to identify a linear of chain of consecutively ranked tuples, if any. If such a chain exists, then the tuples from the chain can be returned for the subsequent *GetNext* calls without additional query cost. We use two observations to extract this chain. First, the only tuples that can dominate the candidates for t_{h+1} are the ones in the top-h. Second, since the database has a fixed (but hidden) global order of all tuples, there always exists a dominance relationship (i.e., direct comparison) between the tuples with rank h and $h+1$. If not, the ranks of these two tuples can be flipped without violating any other relative rankings.

To see how these observations are useful, consider the augmented queries from the baseline approach. Each such query q_i results in a linear chain S_i. We can see that $head(S_i)$ dominates other tuples from S_i. Hence, $head(S_i)$ is the only tuple from S_i that needs to be added to candidate set. Since tuples t_h and t_{h+1} must be directly comparable, we need to consider only the head of each linear chain and compare it with tuple t_h.

The overview of the algorithm is as follows. We have a list of linear chains (from augmented queries of prior *GetNext* invocations) and the linear chain, say S_i, from

which tuple t_h was extracted. We perform pairwise comparison between tuples from different linear chains. An edge is added from node u to node v, if they are directly comparable and u ranks higher than v. Then we compare the tuple t_h with the head of each chain *except* S_i. If *none* of the heads are directly comparable with t_h, then we can assign $head(S_i)$ to be the next ranked tuple without even performing candidate testing. This is possible due to the fact that consecutively ranked tuples are always comparable. If some of them are comparable with t_h, only these form the candidate set for t_{h+1}. The candidate tuples are then compared pairwise with each other to identify non dominated tuples. The domination can be either direct or indirect. It is easy to see that tuple t_{h+1} is guaranteed to be among the non dominated tuples that are also comparable to tuple t_h.

If there are multiple candidate tuples for t_{h+1}, then the candidate testing oracle must be invoked. If not, we are guaranteed that the only candidate tuple must have rank $h+1$. The candidate tuple is then removed from its linear chain and the process is continued till the number of candidates for the next ranked tuple is more than 1. This can potentially result in multiple consecutive next ranked tuples to be retrieved.

Example 2: Consider the same setting as Example 1. We wish to extract 4-th ranked tuple from a top-3 interface. Using attribute A_3, we construct two augmented queries q_1 and q_2 resulting in two linear chains S_1 and S_2. The last tuple t_3 belonged to linear chain S_2. The resulting DAG can be seen from Figure 2.1. Both the tuples t_7 and t_4 are comparable with t_3 and do not dominate each other. However, t_7 is indirectly dominated by t_4 through t_5. Hence we can immediately declare t_4 as the 4-th ranked tuple. Since t_5 also dominates t_7, it is identified as the 5-th ranked tuple. Note that in both the cases, no calls were made to the candidate testing section. Additionally, we identified two consecutively ranked tuples in a single invocation of *GetNext*.

Analysis : At each iteration, let the number of linear chains be l. The query cost for pairwise comparison of tuples between chains is $\prod_{i=1}^{l} |S_i|$. We also require an addition l queries to compare tuple t_h with the heads of each chain. Thus, the algorithm requires at most $l + \prod_{i=1}^{l} |S_i|$ in any iteration. Note that subsequent iterations do need any additional queries till one of the chains is completely consumed as the comparison information between tuples has already been identified.

2.5 Candidate Testing

In this section, we consider the candidate testing problem - i.e., based on prior knowledge of the top-h ranked tuples $t_1, \ldots, t_h$, what queries does one need to issue to the hidden database in order to test whether a given tuple t has rank $h + 1$? We start with two baseline approaches which can require prohibitively high query costs in practice, and then present our two ideas for improving their efficiency: (1) a reduction to beyond-h minimal queries - which significantly reduces both worst- and average-case query costs, and (2) a heuristic query ordering - which further reduces the query cost in practice. It must be noted that if the ranking function is known and based on the attributes returned by the hidden database (e.g. sort by price), then the next ranked tuple can be directly identified from the candidate tuples without an explicit candidate testing phase or querying the hidden database for comparison.

2.5.1 Baseline Approaches

We start by discussing the requirement of rank testing. Like in the previous section, we focus on the case where all attributes are Boolean - with extensions to generic cases discussed in Section 2.6. To prove that t indeed has rank $h + 1$, we have to ensure that no tuple in the database, other than the top-h ones, dominates t. A seemingly straightforward baseline approach is then to first crawl all other tuples

from the database, and then compare each of them with t to identify any dominance relationship. The problem with this approach, however, is that the crawling step requires at least n/k queries - where n is the number of tuples in the database and k is as in the top-k interface - because each query returns at most k tuples. Most common hidden web databases routinely have hundreds of thousands of tuples with a relatively small value of k, resulting in a prohibitive query cost to test a single tuple.

We now consider another baseline which is enabled by the following observation: according to the definition of dominance relationship shown in Section 2.3, the only queries which may "reveal" a tuple dominating t are those that actually match t - i.e., queries of the form

$$q : \text{SELECT * FROM D WHERE } A_{i_1} = t[A_{i_1}] \text{ AND } \cdots \text{ AND } A_{i_r} = t[A_{i_r}] \tag{2.1}$$

where $\{i_1, \ldots, i_r\} \subseteq \{1, \ldots, m\}$ (recall that m is the number of attributes). Specifically, t has rank $h + 1$ if and only if every query of the form (2.1) either returns t as the highest-ranked non-top-h tuple, or returns only tuples in the top-h.

Thus, our second baseline is to issue all queries matching t. One can see that the query cost for the second baseline is $\binom{m}{0} + \cdots + \binom{m}{m} = 2^m$. While this number is often much smaller than n/k for a practical hidden database (because there are usually only a few, e.g., 5 or 10, attributes that can be specified on the input web interface), issuing 2^m queries for each candidate tuple may still lead to an extremely high query cost. In the following two subsections, we develop our two ideas for reducing query cost respectively.

2.5.2 Beyond-h Minimal Queries

Our first idea is to reduce the space of queries required for rank testing from all queries which match t (i.e., of the form in (2.1)) to a much smaller subset which we refer to as the *beyond-h minimal queries*. In the following, we first define beyond-h minimal queries and show the completeness of such queries - i.e., issuing them suffices for rank testing. Then, we describe a (somewhat surprising) mapping of beyond-h minimal queries to finding minimal infrequent itemsets - a problem that has been extensively studied in the database and data mining communities (e.g., see survey in [14]). Finally, we leverage the existing results on minimal infrequent itemsets to derive an upper bound on the number of beyond-h queries.

Definition and Completeness: For any query q which matches t, we use $S(q)$ to represent the *companion attribute set* of the query - i.e., the set of attributes involved in the query. For example, $S(q) = \{A_{i_1}, \ldots, A_{i_r}\}$ for q in (2.1). Then, we call q a *beyond-h minimal query* if and only if it satisfies both of the following two conditions:

- q must return at least one non-top-h tuples - i.e., q must match fewer than k tuples in $t_1, \ldots, t_h$
- any query q' which matches t and has $S(q') \subset S(q)$ must only return top-h tuples - i.e., q' must match at least k tuples in $t_1, \ldots, t_h$.

One can see from the definition that, as the name suggests, q is a "minimal" query which returns any tuple beyond the top-h. We now explain why issuing only beyond-h minimal queries suffices for rank testing. Consider the testing of whether t is the tuple with rank $h + 1$. A key observation here is that any query q_0 which matches t but is not a beyond-h minimal query must satisfy one of the following two conditions:

- If q_0 matches at least k tuples in $t_1, \ldots, t_h$, then one can already infer the answer to q_0 from the knowledge of $t_1, \ldots, t_h$ - i.e., q_0 is useless for rank testing.
- If q_0 matches fewer than k tuples in $t_1, \ldots, t_h$ but is not a beyond-h minimal query, then there must exist a beyond-h minimal query q'_0 such that $S(q'_0) \subset S(q_0)$. If q'_0 returns t as the top-ranked tuple besides top-h, then we are already certain that no non-top-h tuple matching q_0 can outrank t. Otherwise, we are already certain that t cannot have rank $h+1$ - i.e., in either case, we do not need to issue q_0.

Example : Considering the running example from Table I, we can see that $A_3 = 1$ and $A_4 = 1$ are two examples of beyond-h queries for t_4.

Mapping: We now show that the problem of finding all beyond-h minimal queries is equivalent to finding all minimal infrequent itemsets over a transactional database. To understand why, consider the following procedure which maps the top-h tuples to h transactions. We first map each attribute A_j ($j \in [1, m]$) to an item s_j. Then, for each tuple t_i ($i \in [1, h]$), we map it to a transaction r_i by including in r_i all items corresponding to the attributes on which t_i and the testing tuple t share the same value - i.e.,

$$r_i = \{s_j | t_i[A_j] = t[A_j]\}. \tag{2.2}$$

We can see that, with this mapping, the companion attribute set of each beyond-h minimal query q, i.e., $S(q)$, becomes a minimal infrequent itemset over the h transactions, with the frequency threshold being k/h. This observation can be readily made from the definition of beyond-h minimal queries: Since such a query must match fewer than k tuples in $t_1, \ldots, t_h$, $S(q)$ is infrequent given the threshold of k/h. Since no subset of $S(q)$ can match fewer than k tuples in top-h, $S(q)$ must be min-

imally infrequent. One can see that the inverse also holds - i.e., there is a one-one mapping between $S(q)$ and a minimal infrequent itemset.

Example : Suppose we have extracted the top three tuples and want to determine if tuple t_4 is indeed the 4-th ranked tuple. We first map tuples t_1, t_2, t_3 to transactions as $r_1 = \{A_1 = 0, A_5 = 1\}, r_2 = \{A_1 = 0, A_4 = 1, A_5 = 1\}$ and $r_3 = \{A_1 = 0, A_3 = 1, A_5 = 1\}$. The threshold is $\frac{3}{3} = 1$. The infrequent itemsets are $A_3 = 1$ and $A_4 = 1$ which correspond to beyond-h queries for t_4. Also, the number of beyond-h queries is dramatically smaller than the 2^5 queries needed in the previous approach.

While (as we shall show below) the mapping enables us to derive an upper bound on the number of beyond-h minimal queries, we would like to remark here two major differences between our problem and the traditional problem of finding minimal infrequent itemsets.

First, even though finding all minimal infrequent itemsets is known to be #P-complete, the time complexity is *not* really a concern for our problem because our input size m - i.e., the number of attributes - is usually much smaller than the number of items in a transactional database. As such, we could simply enumerate all 2^m possible itemsets (and find the minimal infrequent ones) without causing significant overhead. What is a major concern for us, however, is the number of minimal infrequent itemsets because it translates to the number of queries we have to issue through the web interface - a costly and time-consuming process.

Second, our frequency threshold, i.e., k/h, is generally much larger than the threshold traditionally considered for minimal infrequent itemsets. Note that even an $h = 2k$ may bear significant interest as third-party analyzers are most likely interested in those highly ranked, albeit outside top-k, tuples. As we shall show below, this unusually high threshold enables us to improve the upper bound on the number of beyond-h minimal queries when h is small.

Upper Bound: First, according to the existing results on the number of minimal infrequent itemsets, that the number of beyond-h minimal queries can be bounded by $\binom{m}{m/2}$. We now show that when h is small, specifically $h \leq m/2 + k - 1$, the number of beyond-h minimal query q has another upper bound of $\binom{m}{h-k+1}$.

An important observation here is that the number of predicates in a beyond-h minimal query, say q, is at most $h - k + 1$. To understand why, consider a query-construction process in which we start with the SELECT * query, and then gradually add into it one conjunctive predicate in q (i.e., one attribute in $S(q)$) at a time, until the query matches fewer than k tuples in the top-h. One can see that each predicate being added, say $A_i = t[A_i]$, must remove at least one top-h tuple from the set of tuples matching the previous query, because otherwise one can always remove $A_i = t[A_i]$ from q without changing the answer to q - contradicting the fact that q is beyond-h minimal. As such, once $h - k + 1$ predicates are added to the query, the number of top-h tuples matching the query must drop to below k - i.e., $S(q)$ contains at most $h-k+1$ attributes. Again, since all beyond-h minimal queries forms an anti-chain, the number of them is at most $\binom{m}{h-k+1}$ when each beyond-h minimal query contains at most $h - k + 1$ predicates and $h - k + 1 \leq m/2$.

In summary, we have the following theorem:

Theorem 1. *Given the top-h tuples, the maximum number of queries one needs to issue for testing whether a tuple has rank $h + 1$ over a database of m attributes and n tuples, $c(n, m, h + 1)$, satisfies*

$$c(n, m, h + 1) \leq \binom{m}{\min(h - k + 1, m/2)}. \tag{2.3}$$

2.5.3 Query Ordering

Our next idea to reduce query cost that works very well in practical hidden databases is a heuristic - query ordering. Recall that *beyond-h query* is a minimal query that returns at least one non-top-h tuple. Given a candidate tuple t, if all its corresponding beyond-h minimal queries returns t as the highest ranked non-top-h tuple, then we can conclude that no other tuple dominates t and hence t has rank $h+1$. Note that to make this conclusion, it is mandatory to execute all the beyond-h queries.

The key idea in query ordering is that of *elimination*. If we can eliminate all but one tuple from the candidate set, then the remaining tuple has to be the next ranked tuple and we can make that conclusion even without executing any of the beyond-h queries for it. This is due to the fact that the candidate generation step produces a set of tuples one of which is guaranteed to be in the next ranked tuple. The query ordering heuristic takes the idea a little further.

Given a candidate tuple t and one of its beyond-h queries q, there are two possible results : (1) t is the top ranked non-top-h tuple (2) t is *not* the top ranked non-top-h tuple. In the first case, the query q did not give any contradicting evidence for t and the next beyond-h query needs to be executed. On the other hand, the second outcome provides an evidence that disqualifies t from being the next ranked tuple. i.e. the procedure for testing t can be terminated early. The heuristic tries to reorder the execution of beyond-h queries so that if t is not the No. $h+1$ ranked tuple, it is detected earlier.

While reordering the queries of a single candidate tuple is useful by itself, the maximum advantage is obtained when the set of beyond-h queries of *all* the tuples in candidate set are reordered. By ordering queries based on the chance that it eliminates atleast one candidate tuple and executing them in that order, we eliminate as many

candidates as possible in the least number of queries. Furthermore, while executing the queries, any candidate tuple dominate by others can be immediately rejected.

The heuristic relies on two factors that make a beyond-h query q useful. Note that both the factors implicitly favor shorter queries over longer ones.

- The number of tuples in *candidate set* matched by q. If q matches l tuples in candidate set, we can immediately eliminate the $l-1$ dominated candidates after executing q as they cannot have rank $h+1$.
- The *expected* number of tuples in the *database* that is matched by q. If q matches a large fraction of database, then there is a high likelihood that one of such tuples will be ranked higher than candidate tuple t. Of course, since the entire database is not available to us, we estimate the fraction by assuming a random database where the attribute values are uniformly distributed. While this assumption does not always hold, it serves as a useful approximation and heuristic. Given a boolean database with 5 attributes and any query with two attributes can be expected to match 25% of the tuples.

In summary, the query ordering heuristic pools the beyond-h queries of all candidate tuples and reorders them based on a weighted combination of the two factors described above. The weights can be determined using domain knowledge of the hidden database. The queries are executed in the order so as to eliminate the candidate tuples as early as possible. Any candidate tuple dominated by a non top-h tuple or other candidate tuple are eliminated. The process is continued till only one candidate remains.

Example : Suppose we wanted to determine if t_3 or t_4 is the third ranked tuple. $A_3 = 1$ and $A_4 = 1$ are two of the beyond-h queries for t_4 while the corresponding ones for t_3 are $A_3 = 1$ and $A_4 = 0$. Since the query $A_3 = 1$ matches both t_3 and t_4, it

is executed before either of $A_4 = 1$ or $A_4 = 0$. After executing $A_3 = 1$, we note that t_3 is ranked higher than t_4 in the result and hence declare it as the 3-rd ranked tuple.

Analysis : The query cost of heuristic is bounded by the upper bound for the number of beyond-h queries for the tuples in candidate set. In the worst case, this procedure degenerates to executing all the beyond-h queries for *all but one* of the candidate tuples.

2.6 Algorithm Design and Extensions

In this section, we integrate the candidate generation and testing techniques discussed in previous two sections to develop our final algorithms for GetNext. In addition, we shall describe different extensions of our algorithms such as retrieving the top ranked tuples when no unique total order exists among them or retrieving top ranked tuples that satisfy additional user specified filters.

2.6.1 Algorithm Design

We start by integrating our DAG-based candidate generation algorithm with the beyond-h queries based candidate testing algorithm to develop the BEYOND-h-GETNEXT algorithm. To be the next ranked tuple, any candidate tuple must be the top ranked non top-h tuple for each of its beyond-h queries. Algorithm 1 depicts the pseudocode of BEYOND-h-GETNEXT.

We also integrate our candidate generation algorithm with the heuristic candidate testing algorithm to develop the ORDERED-GETNEXT algorithm. The only difference between ORDERED-GETNEXT and BEYOND-h-GETNEXT is in the rank testing phase. In ORDERED-GETNEXT, we first identify the beyond-h queries for *all* candidate tuples and order them based on their likelihood of rejecting a candidate tuple. The queries are executed until all but one candidate tuples have been

Algorithm 1 BEYOND-h-GETNEXT

1: **Input parameters :** $topH$, the set of top ranked tuples
2: Get candidates for t_{h+1} using candidate generation
3: **for** each candidate tuple t **do**
4: Generate and execute beyond-h-queries for t
5: If any tuple other than top-h tuples dominate t, reject t
6: **end for**
7: **return** unrejected tuple as t_{h+1}

rejected. The remaining tuple is declared as No. $h + 1$ tuple. Algorithm 2 depicts the pseudocode of ORDERED-GETNEXT.

Algorithm 2 ORDERED-GETNEXT

1: **Input parameters :** $topH$, the set of top ranked tuples
2: Get candidates for t_{h+1} using candidate generation
3: Collect the beyond-h queries of all candidates and order them based on likelihood to reject candidates
4: **for** each query **do**
5: Execute query
6: Reject any candidate tuple dominated by other candidates or a non top-h tuple
7: If only one candidate left, break
8: **end for**
9: **return** remaining tuple as t_{h+1}

2.6.2 Absence of Total Order within Top Ranked Tuples

One of the assumptions that was made by the algorithms was that the set of top ranked tuples that we wish to retrieve are totally ordered and the order is inferable from the hidden database interface. Specifically, we assumed that tuples t_h and t_{h+1} was directly comparable. In this subsection, we discuss how to handle the different scenarios when the assumption does not hold.

Two tuples can be compared with each other either directly or indirectly and similarly the dominance relationship can be established directly or indirectly through other intermediate tuples. For example, we might have two tuples t and v that are not directly comparable. However, if $t \succ u$ and $u \succ v$, then we can indirectly infer their dominance relationship. If two tuples are not comparable at all, even indirectly, then their dominance relationship cannot be established. Choosing either of the tuples to be the next ranked tuple results in a potentially valid total ordering from the limited information available. The possibility of two tuples not comparable affects both the candidate generation and testing steps.

Candidate Generation: In candidate generation, if the head tuple of *every* linear chain was not comparable to t_h, then we cannot assign the head tuple from the linear chain from which t_h was extracted to be the next ranked tuple (as it is not comparable to t_h). All the non dominated candidate tuples are sent to candidate testing for identifying the next ranked tuple.

Candidate Testing: If multiple tuples from candidate set are not dominated by any other tuple other than the ones in top-h (including other tuples in candidate set), then each of them can potentially be considered as the next ranked tuple. Hence, one of the non dominated candidate tuples is selected uniformly at random as the next ranked tuple and the process is continued. This random selection creates one of the valid partial order of the top ranked tuples. Since the output total order is

no longer accurate, a metric must be chosen to measure the distance between the actual total order and the partial order. The accuracy measure used is the expected distance between a randomly generated total order and the actual total order. The distance between two ranked list can be computed using Kendall τ or the Spearman's footrule.

2.6.3 Top Ranked Tuples with Selectivity Constraints

The discussions in the previous sections described techniques to retrieve the top ranked tuples from the entire database. An equally important and practical scenario is one where the user is interested in the top ranked tuples over a subset of the database. For example, the user might be interested in the cheapest flights with in-flight wifi. An alternate perspective is to view the problem as retrieving top ranked tuples where some of the attribute values are already preset by the user, for e.g. wifi. The specified attributes then partition the entire hidden database into two partitions - one which matches the specified attributes and another which does not match the specified attributes. In this subsection, we discuss how to extend the techniques discussed so far to solve this problem.

An initial approach one might come up with is to keep retrieving top tuples from the *entire* database incrementally till we have adequate number of tuples satisfying the user selectivity constraints. This might be the only possible approach if the user selectivity constraint cannot be filtered through the interface of hidden database. For e.g. if the user is interested in top-10 flights with in-flight wifi. However if the constraint cannot be entered via the airline interface, we can keep retrieving top ranked tuples till we have accumulated 10 flights with in-flight wifi. If the filters are too selective, then the number of tuples to be fetched before we return the user results could be very high.

However, if the user's constraints can be entered via the hidden database interface (but user still needs more that k results), then an alternate approach is possible. As an example, the user might be interested in top-20 flights with wifi on a top-10 interface where the wifi availability is an input attribute. We can directly apply the techniques for extracting top ranked tuples over the subset of database that satisfies the selectivity constraints instead of applying it on the original database. This corresponds to prefixing the selectivity constraints to each of the queries executed by the algorithms. The candidate generation phase produces only tuples that satisfy the constraints.

The algorithms that work only on the database subset might seem to be a more efficient approach to solve the problem and in most scenarios it is. However, there are few factors that influence the output. First, if the selectivity constraints are coarse or not too selective, then a large section of database would be covered. This in turn, increases the chances of finding a correct set of top ranked tuples satisfying the constraints. If the number of tuples that match are small, then there is a high likelihood that the tuples are not comparable. In this case, we are left with a partial order of tuples instead of a total order.

Secondly, even if a total order exist among the top ranked tuples in the subset, it might not be possible to order them by only looking at the candidate tuples matching the constraints. This is because, the tuple(s) that helped to indirectly compare and order the candidate tuples, say t_x and t_y could itself not satisfy the selectivity constraint. In this case, the two tuples are incomparable, even though a global order exist between them. In both the scenarios, we are potentially left with a partial order. The techniques used in linearizing the partial order can be used to solve this issue.

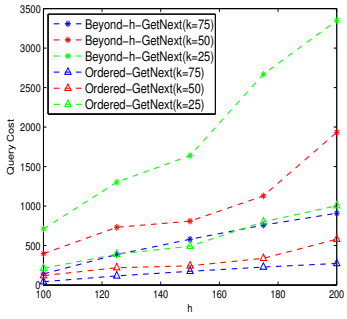


Figure 2.2. GetNext: Query cost vs h on Boolean dataset.

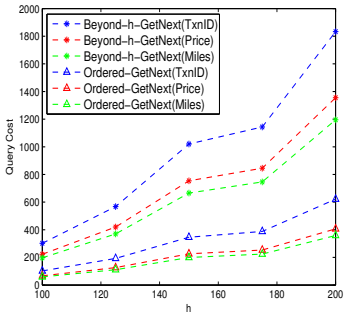


Figure 2.3. GetNext: Effect of different ranking functions on Autos dataset.

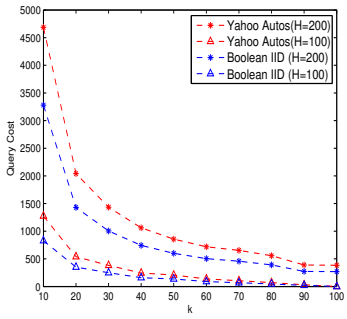


Figure 2.4. GetNext: Query cost versus k.

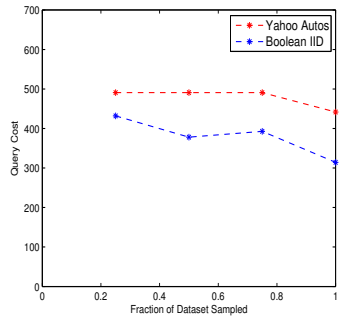


Figure 2.5. GetNext: Query cost versus database size.

2.7 Experimental Results

In this section we describe our experimental setup, compare the performance of algorithms for candidate generation and candidate testing and show the efficiency and accuracy of our methods.

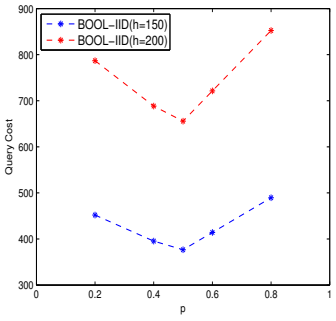


Figure 2.6. GetNext: Query cost versus skew.

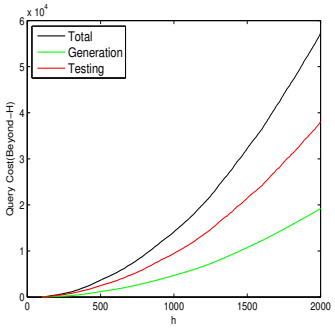


Figure 2.7. GetNext: Query cost versus large h.

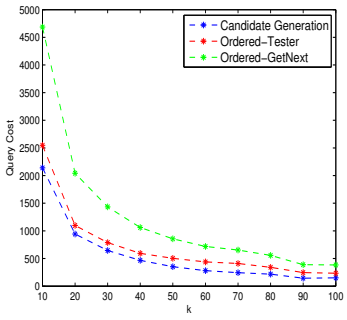


Figure 2.8. GetNext: Comparing candidate generation versus testing.

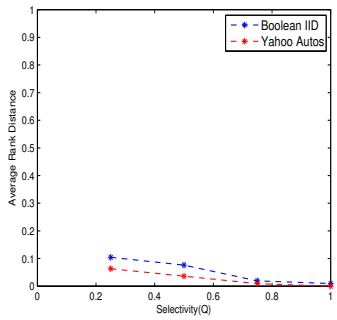


Figure 2.9. GetNext: Rank distance versus query selectivity.

2.7.1 Experimental Setup

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

Datasets: We used both synthetic and real-world data sets in the experiments. The synthetic dataset we used is a boolean one with 200,000 tuples and 80 attributes. The tuples are generated as i.i.d. data with each attribute having probability of p = 0.5 to be 1 (except for one experiment where we created different datasets with different values of p). We refer to this dataset as the BOOL-IID dataset. The real-

world dataset we used consists of data crawled from the Yahoo! Autos website [5], a real-world hidden database. It contains 200,000 used cars for sale in the Dallas-Fort Worth metropolitan area. There are 32 Boolean attributes such as A/C, Power Locks, etc, and 6 categorical attributes, such as Make, Model, Color, etc. The domain size of categorical attributes ranges from 5 to 16.

Real-World Online Experiment: In addition to the offline experiments described above, we also directly applied our techniques *online* over Amazon.com (specifically Amazon's Product Advertising API[6]) to discover the top-250 (according to sales rank) Amazon DVD titles from a top-100 interface[7] provided by the API. Since the individual item description provided by Amazon.com reveals the sales rank of the item, we were able to verify the correctness of all results discovered by our algorithm. For this online experiment, (top-k) search query can be constructed using 15 categorical attributes such as Actor, Artist, Publisher, etc., with their domain sizes ranging from 5 to over 1,000. Amazon.com has a limit of 2,000 queries per IP address per hour.

Algorithms: We tested two algorithms BEYOND-h-GETNEXT and ORDERED-GETNEXT. However, since both these algorithms use the same candidate generation technique, we highlight the behavior of the candidate generation and testing phase separately. In other words, we plot the performance of algorithm GETNEXT for different parameters and then compare the performance of different candidate testing algorithms. This choice of presentation accentuates the improvements provided by the beyond-h-queries and the heuristic query ordering that gets masked when directly comparing BEYOND-h-GETNEXT and ORDERED-GETNEXT.

[5]http://autos.yahoo.com/

[6]https://affiliate-program.amazon.com/gp/advertising/api/detail/main.html

[7]By default Amazon's Product Advertising API provides a top-10 interface, while allowing a user to "Page Down" for up to 9 times, essentially leading to a top-100 interface.

Performance Measures: We use query cost, the number of queries executed on the hidden database as the performance measure. This includes the queries used to retrieve candidate tuples, queries to compare candidates and the beyond-h queries for each candidate. When the total order cannot be inferred, we use expected distance between randomly generated total order and the actual total order. The distance between two ranked lists is computed using Kendall-τ metric.

2.7.2 Experimental Results

In the following discussion we denote the number of top ranked tuples retrieved from the hidden database as h. In other words, it denotes the maximum number of invocations of GETNEXT by the third party service.

Query cost versus h: In our first experiment, we evaluated the performance of our algorithms BEYOND-h-GETNEXT and ORDERED-GETNEXT on the boolean dataset by investigating the query cost as a function of h for various different values of k. As Figure 2.2 shows, the query cost increases with increasing h, as is expected. Moreover, significant savings are achieved by using the ordering heuristic in ORDERED-GETNEXT. We also notice that k plays an important role in the efficiency of the algorithms: larger k results in more efficient performance. To consider a specific performance point, when $k = 75$ and $h = 200$, ORDERED-GETNEXT requires less than 300 additional queries to retrieve the extra 125 tuples.

We also performed similar experiments on the Autos dataset and observed similar trends, with ORDERED-GETNEXT outperforming BEYOND-h-GETNEXT (Figure 2.3). Additionally, we also investigated the effect the specific ranking function used has on the performance of our algorithms. As Figure 2.3 shows, we used three different ranking attributes: TxnID (a unique ID for each tuple), as well as attributes such as Price and Miles. We note that the performance of our algorithms

vary for different ranking functions, but nevertheless are still very efficient in all cases (and as noted earlier, our algorithms do not try to take advantage of any knowledge of these ranking functions).

Query cost versus k: In our next experiment, we investigated the effect of k on the query cost for fixed values of h, for both the boolean dataset as well as the Autos dataset. As Figure 2.4 shows, the positive effect of larger values of k on the query cost is dramatic, with larger values of k being very effective in reducing the query cost of our algorithms. This is to be expected, as our earlier arguments in the paper have shown that large k significantly reduces the number of queries needed in the candidate generation and testing procedures (since the number of minimal infrequent itemsets in a database rapidly reduces with increasing support threshold).

Query cost versus database size: Since our algorithms are designed to retrieve only the top-h tuples from the database, the actual size of the database should not have a significant impact on the performance of our algorithms. This is verified in Figure 2.5, which shows that the query cost remained practically unchanged for ORDERED-GETNEXT, even though we try our experiments on various fractional sizes of the original databases (the slight dip in query cost is attributable to the uncertainty of the sampling process). In this experiment, $k = 100$ and $h = 200$.

Query cost versus skew: We experimented with ORDERED-GETNEXT ($k = 100$, $h = 200$) on several boolean databases created with different values of skew parameter p. As Figure 2.5 shows, the algorithm is most efficient when the database has equiprobable 1s and 0s, but the cost increases when the proportion becomes unbalanced. This is attributable to the fact that when the database contains more 1s (or more 0s), the algorithm has to "dig deeper" - i.e., issue a larger number of (and more specific) queries in order to generate all candidates.

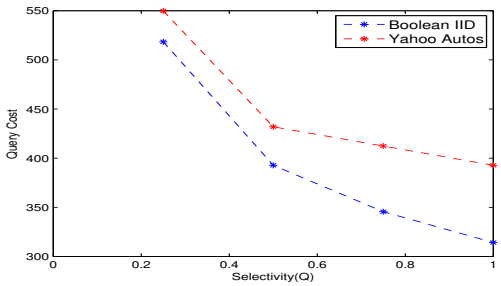


Figure 2.10. GetNext: Query Cost versus Query Selectivity.

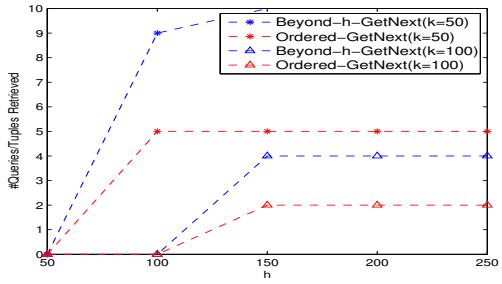


Figure 2.11. GetNext: Query Cost versus h on Amazon DVD website .

Effect of large h: Our earlier experiments were focused on values of h that were at most a small factor larger than k. Such values are meaningful in actual applications where an user is interested in seeing a few more tuples than what has been returned to her by the original query. But we were also interested in stress-testing our algorithms on much large values of h to see how they performed. Figure 2.7 shows the results of such an experiment using ORDERED-GETNEXT on the Autos dataset, where k was set at 100. As can be seen, the query cost increases quite significantly for much larger values of h, which leads to the conclusion that beyond a certain point, it is actually preferable to *crawl* the database and extract the top-h queries rather than use ORDERED-GETNEXT. The figure also profiles the separate query costs of the candidate generation and testing procedures.

Comparing generation versus testing procedures: In Figure 2.8, we compare the query costs of the two main procedures: candidate generation and candidate testing. We ran ORDERED-GETNEXT over the Autos dataset for $h = 200$ and varied k. As can be seen, the query cost is almost equally divided between the generation and test procedures for almost all points of the curve, with testing being slightly more expensive.

Effect of query selectivity: In Figures 2.9 and 2.10, we investigate the impact of selectivity. If a query is extremely selective, then it is clear that no algorithm can extract a total order of the top-h tuples. In such situations, our algorithms return a partial order of the top-h tuples. As discussed in 2.6.2, we compare a random total order that conforms to the returned partial order against the true top-h tuples for that query using Kendall-τ measure. As the query becomes less selective, the rank distance increases and its query cost becomes less, which is to be expected as the candidate testing procedure gets opportunities to terminate early as one needs a smaller number of queries to exclude a tuple from consideration. Similarly, as the query selectivity drops, our algorithm can retrieve the actual total order. Our experiments uses ORDERED-GETNEXT for both datasets, with $k = 100$ and $h = 200$.

Experiment against Amazon DVD Titles : To show the practicality of our algorithms, we retrieved the top-250 Amazon DVD titles in terms of their sales rank. Note that by default, Amazon only displays the top-100 items in any category. The correctness of our algorithm is verified by the checking the individual item description pages of the items discovered by GETNEXT (which reveals the actual sales ranking of the items). The queries were made using the Amazon Product Advertising API and the maximum value of k is 100. A sample query to get the top-10 PG rated

DVDs ordered by their salesrank is shown in footnote[8].Figure 2.11 shows that when $k = 100$, the top-250 titles can be retrieved using fewer that 500 queries, well below the 2000 queries-per-hour-per-IP-address limit imposed by Amazon.com. The figure also shows the behavior of both BEYOND-h-GETNEXT and ORDERED-GETNEXT for different values of k and h.

2.8 Related Work

Information Integration and Extraction for Hidden databases: A significant body of research has been done on information integration and extraction over hidden databases - see tutorials [15, 16]. Due to space limit, we only list a few closely-related work: [17] proposes a crawling solution. Parsing and understanding web query interfaces has been extensively studied (e.g., [18, 19]). The mapping of attributes across different web interfaces has also been addressed (e.g., [20]). Also related is the work on integrating query interfaces for multiple web databases in the same topic-area (e.g., [21, 22]). Our paper provides results orthogonal to these existing techniques as it represents the first formal study on retrieving top-h ($h > k$) tuples matching a user-specified query by reformulating the query through a top-k interface.

Data Analytics over Hidden Databases: There has been prior work on crawling, sampling, and aggregate estimation over the hidden web, specifically over text [23, 24] and structured [17] hidden databases and search engines [25, 26, 27]. Specifically, sampling-based methods were used for generating content summaries [28, 29, 30],

[8]http://ecs.amazonaws.com/onca/xml?Service=AWSECommerceService
&AWSAccessKeyId=[fill]&Operation=ItemSearch&SearchIndex=DVD
&ResponseGroup=Large,SalesRank&Sort=salesrank&AudienceRating=PG
&Timestamp=[fill]&Signature=[fill]

processing top-k queries [31], etc. Prior work (see [3] and references therein) considered sampling and aggregate estimation over structured hidden databases.

Top-k Query Processing: There have been extensive studies on retrieving the top-k tuples over a traditional database - see [32] for a survey. Our approach differs by allowing the retrieval of top-h tuples through a restricted top-k web interface.

Frequent Itemset Mining: We map the discovery of beyond-h queries to the problem of infrequent-minimal-itemset mining - a problem well studied in data mining [14]. [13] provides additional details about algorithms and properties for infrequent itemset mining.

2.9 Conclusion

In this paper we have initiated study on the problem of retrieving the top-h ($h > k$) tuples from a hidden web database that only provides a top-k search interface. To address the fundamental operator GetNext, we proposed a two-step process, candidate generation and candidate testing, and developed efficient algorithms for both steps. We conducted comprehensive set of experiments over synthetic datasets and real-world hidden databases which demonstrate the effectiveness of our proposed techniques. There are multiple exciting directions for future research. We intend to investigate the possibility of retrieving the top ranked tuples approximately - for e.g., retrieve as many top ranked tuples under budget cost or in a rank agnostic fashion. Further, we plan to build attractive demonstrations of mashup applications against real-world hidden web databases.

CHAPTER 3

Rank Discovery From Hidden Web Databases

Many web databases are only accessible through a proprietary search interface which allows users to form a query by entering the desired values for a few attributes. After receiving a query, the system selects the top-k matching tuples according to a pre-determined ranking function, and then return them to the user. Since the rank of a tuple largely determines the attention it receives from website users, ranking information for any tuple - not just the top-ranked ones - is often of significant interest to third parties such as sellers, customers, market researchers and investors. In this paper, we define a novel problem of rank discovery over hidden web databases. To address the problem, we introduce a taxonomy of ranking functions, and show that different types of ranking functions require fundamentally different approaches for rank discovery. Our technical contributions include principled and efficient randomized algorithms for estimating the rank of a given tuple, as well as negative results which demonstrate the inefficiency of any deterministic algorithm. We show extensive experimental results over real-world databases, including an online experiment at Amazon.com, which illustrates the effectiveness of our proposed techniques.

3.1 Introduction

3.1.1 The Rank Discovery Problem

Many web databases, e.g., Yahoo! Autos, Amazon.com, are "hidden" behind (i.e., only accessible via) a restrictive form-like interface which allows a user to form a search query by specifying the desired values for a few attributes; and the system

responds by returning a small number of tuples matching the search query. Almost all such interfaces enforce the top-k constraint - i.e., when more than k tuples (where k is typically a predetermined small constant) match the user-specified query, only k of them are preferentially selected according to a ranking function and returned to the user. While such restrictive form interfaces of hidden databases might suffice for the simplest use-cases, i.e., that of a normal user searching for some items in these databases, they often cannot satisfy users with specific needs and also prevent many interesting third-party services from being developed over web databases. There has been several recent works on developing techniques to enable additional functionality over such databases that operate via the restrictive interface, such as sampling and aggregate estimation (see [3, 4, 2] and references therein).

In this paper, we consider a novel problem, that of discovering rank-related information from a hidden web database:

> RANK DISCOVERY PROBLEM: Given a query q, and a tuple t that satisfies the selection conditions of q, compute the rank of t among all tuples in the web database that satisfy q.

In a general sense, the rank discovery problem is a fundamental data analytics task, because it seeks to determine the rank/position of an item as compared to similar competing items along multiple attributes/facets. In the case of web databases, since the rank of a tuple largely determines the attention it receives from website users, ranking information for any tuple - not just the top-k ranked ones - is often of significant interest to third parties such as sellers, customers, market researchers and investors. Solutions to the rank discovery problem has the potential of enabling new third-party application scenarios that have not been considered in earlier work. For example, the author of a book on sale at Amazon would be keen to track and monitor the ranking of her book within a set of similar competitors (e.g., how does it rank in

sales, or customer reviews, etc., compared to other similar books on science fiction?). Likewise, competitors to an app available at Apple's iOS and Mac App Stores would be interested in monitoring the app's grossing rank and measure the market response to determine if it is time to start competing with the app. Investors may be keen to do simultaneous monitoring of numerous products newly released by competing companies, in order to determine which one is likely to be a hit and which company to invest in.

Although the rank discovery problem, as stated above, appears deceptively compact and simple, it is challenging because most web databases do not explicitly disclose a tuple's rank beyond the top-k tuples. The rank has to be discovered indirectly, by carefully issuing multiple related queries to the website's query interface and recovering/inferring the tuple rank by piecing together the information returned from these queries. Moreover, our investigations revealed that different websites have widely varying characteristics, resulting in a myriad of interesting facets and variants of the rank discovery problem that require fundamentally different approaches in their solutions. In the rest of this introductory section, we provide an overview of this spectrum of problem variants, highlight their difficulties and challenges, and summarize our technical contributions - both algorithmic as well as negative results.

3.1.2 Problem Variants and Challenges

Web databases use a broad variety of ranking functions. These functions typically compute a score for each tuple matching the query conditions, and return the k tuples with the highest scores. These ranking functions can be classified along several different dimensions/categories. One dimension is whether the function is *static* or *query dependent*. A static ranking function assigns tuple scores independent of the query - i.e., all tuples are globally ordered in the database. For example, Amazon

allows users to search for books by specifying a few desired attributes (e.g., Language, Format, Genre, Release date, Title, etc.), and the system returns up to k matching books, ranked by price, average customer review, popularity (i.e., sales amount), recency, etc. One can see that all these ranking functions are static. Other examples are the "sort by bestsellers" or "sort by grossing" static ranking functions used by Apple's iOS and Mac App Stores. A ranking function is query-dependent if the score of a tuple varies for different queries - e.g., where all tuples are ordered according to the number of attribute matches between the query and each tuple, or by a more sophisticated notion of "relevance".

Within static ranking functions, a second dimension for categorization is whether the function is *observable* or *proprietary* (i.e., unobservable). Observable ranking functions are those where the end-user can determine the score of a tuple from the tuple values alone - e.g., in the aforementioned Amazon example, if the ordering is by price, the score of a tuple is obvious. Observable ranking functions may be further categorized into whether the scoring attribute can be *queried* or not. For example, users can query for products in Amazon by specifying desired price ranges, but cannot specify desired recency, although both scores are observable in returned products. A proprietary ranking function is one where the tuple's score is hidden from the public's view - e.g., the actual values of popularity (i.e., sales amount) and/or gross sales for Amazon and App Stores are never revealed to the end user. In fact, it is an open secret that many websites often use proprietary ranking functions to promote high-profit or slow-selling products to customers.

We note that no matter what variant of web database we encounter, trivial solutions to the rank discovery problem are possible if (1) all input tuples are very highly ranked so as to enter the top-k results returned by the hidden database, and/or (2) the third-party analyzer can negotiate a private agreement with the web database

owner in order to retrieve the ranking of the entire query results beyond the top-k limitation. Nonetheless, note that the information most useful to an investor and/or competitor occurs *before* a book/app enters the top-seller list and becomes more or less known to the general public anyway. On the other hand, private negotiations are often very difficult due to revenue sharing, legal requirements, security and myriad of other thorny issues. As such, our focus in this paper is to develop automated third-party algorithms that only use the public interfaces of web databases without requiring any additional cooperation from the database owners.

Another seemingly straightforward solution to issue all possible queries through the web interface so as to crawl all rank-related information one could possibly infer from the hidden database - and then analyze the query answers locally to address the above problems. Nonetheless, a key pitfall of this solution is its prohibitively high query cost (note that just crawling all tuples can be extremely expensive [5]) - which is simply infeasible for real-world web databases which often impose a per user/IP limit on the number of queries one can issue over a given time frame (e.g., Google Search API allows only 100 free queries per user per day).

Given the pitfalls of the above mentioned approaches, in this paper we develop algorithms to solve the rank discovery problem only using the public interfaces of web databases, with the important design objective of maintaining a *small query cost* - a goal shared by most existing studies on exploring hidden web databases because of the query-number limitations enforced by web databases (e.g., [3]). Our algorithms produce approximate answers, and thus another important design objective is to produce answers with small relative error.

3.1.3 Outline of Technical Results

As mentioned above, different types of ranking functions require fundamentally different approaches for rank discovery. We first study the two extreme cases: For ranking functions that are static, observable, and can be queried through the search interface, we show that a simple solution exists for rank discovery: use the existing aggregate estimation algorithms [3] to estimate the COUNT of tuples with a higher rank. On the other extreme - i.e., when the ranking function is query dependent - we show that it is impossible for any algorithm to discover the rank of t among tuples matching q - unless q can be queries through the search interface, and t is returned among the top-k results.

For the remaining cases - i.e., when the ranking function is (i) static and proprietary, or (ii) static, observable but cannot be queried - we develop RANK-EST, a rank estimation algorithm that interleaves the following two methods:

- RANK-EST-S, a sampling-based process which first draws uniform random samples from the hidden databases, and then perform rank companions between the input tuple and the samples to enable rank estimation.
- RANK-EST-H, a randomized process which randomly constructs and issues queries that return tuples ranked higher than the input tuple, and use the query answers to directly produce a rank estimation.

While RANK-EST-S works well for most tuples in the database, it cannot effectively handle highly ranked tuples, because a very large sample is required to accurately estimate their ranks. RANK-EST-H, on the other hand, is designed specifically for these tuples. As such, by interleaving the two processes, RANK-EST achieves efficient and accurate rank estimation over all tuples in the database.

Interestingly, while RANK-EST works for both proprietary and observable ranking functions, the technical challenges facing the design of RANK-EST-S (and

consequently, the query cost required by it) differs drastically between the two. Specifically, while comparing rank between the input and a sample tuple is straightforward for observable ranking functions, it can be extremely difficult for proprietary ones. Indeed, we prove in the paper that guaranteeing the correctness of rank comparison can be downright impossible in the worst-case scenario. While doing so is possible for the vast majority of real-world databases, we prove a hardness result showing that no deterministic algorithm can do so without issuing an extremely large number of queries. To address this problem, we devise LV-RANK-COMPARE, a randomized rank comparison algorithm, inside RANK-EST-S. LV-RANK-COMPARE is a Las-Vegas algorithm - i.e., it always produces the correct answer, but with varying query costs across different executions.

In summary, the major contributions of this work are as follows. We introduce and motivate the novel problem of rank discovery over hidden web databases. We define a comprehensive spectrum of ranking functions according to various dimensions such as query-dependent vs. static, observable vs. proprietary, and whether the scoring attribute can be queried or not. We discuss the feasibility of rank discovery for each type of ranking function, and show that different types of ranking functions require fundamentally different approaches for rank discovery. For proprietary and observable ranking functions, we develop RANK-EST which interleaves two separate procedures for handling high and low ranked tuples, respectively. We present careful theoretical analysis including negative results that preclude efficient deterministic solutions. We present a thorough experimental evaluation of our algorithms over real-world hidden web databases. We also describe online experiments over Amazon.com which demonstrates the effectiveness of our proposed algorithms.

3.2 Rank Discovery Problem

In this section, we introduce a taxonomy of ranking functions commonly used and, for each type of ranking functions, discuss the feasibility of rank discovery. Then, we define the technical problem addressed in the paper.

3.2.1 Model of Hidden Databases

Consider a hidden database D with n tuples and m input attributes $A_1, A_2, \ldots, A_m$. Given a tuple t and an attribute A_i, let $t[A_i]$ be the value of A_i in t. Let $Dom(A_i)$ be the domain of A_i. For the purpose of this paper, we restrict our attention to categorical attributes and assume the appropriate discretization of numeric ones. We also consider all tuples distinct and without null values.

A user can query the system by specifying the desired values for a subset of $A_1, \ldots, A_m$. Thus, a user query q is of the form `SELECT * FROM` D `WHERE` $A_{i_1} = v_{i_1} \& \ldots \& A_{i_s} = v_{i_s}$, where $i_1, \ldots, i_s \in [1, m]$ and $v_{i_j} \in Dom(A_{i_j})$. Real-world hidden databases generally restrict users' access to top-k tuples - which may be presented on one page or over multiple pages (accessed by page turns or clicking next at the bottom of the results page)[1].

Formally, let the set of tuples matching q be $Sel(q)$. If $|Sel(q)| > k$, an $overflow$ occurs and only the top-k results are returned, along with an overflow flag indicating that more tuples matching the query cannot be returned. If $|Sel(q)| = 0$, then an *underflow* is said to occur. Otherwise, i.e., when $|Sel(q)| \in [1, k]$, we say that q is valid - i.e., the user retrieves all tuples matching the issued query.

[1]For example, Google limits the number of page turns to 10 if 100 results are displayed per page, and 100 if 10 results per page - effectively resulting in a top-1000 interface.

3.2.2 Taxonomy of Ranking Functions

Ranking function is what the hidden database uses to determine which k tuples to return when a query overflows. Consider a ranking function $f(\cdot)$ which takes a tuple and a query as input and outputs a score. There are two broad categories of ranking functions: *static* and *query-dependent*. A ranking function $f(\cdot)$ is *static* if $\forall q_1, q_2,\ f(q_1, t) = f(q_2, t)$. Otherwise, it is *query-dependent*.

Within static ranking functions, we can further partition them into two types, *observable* or *proprietary*. A ranking function is *observable* if $f(t)$ is displayed along with (or can be inferred from) other attributes when a tuple is returned in a query answer. Otherwise, it is proprietary. Examples of observable ranking functions include Price, Listed Date, etc., used by many e-commerce websites. Proprietary ranking functions include "Popularity" in numerous websites such as Amazon.com, Priceline, Kickstarter, etc., "sort by gross sales" used by Apple's and Android's App Stores, as well as many proprietary scoring functions such as Moviemeter in IMDB and "Rank by value" in Seatguru.

Finally, within observable (static) ranking functions, we can further partition them into two categories according to whether the scoring attribute can be *queried* through the search interface. Specifically, a ranking function can be queried if it is possible to issue `SELECT * FROM D WHERE` $f(t) = c$ through the interface. An example is the Recency ranking function used by Amazon.com.

Figure 3.1 depicts the above taxonomy of ranking functions.

3.2.3 Feasibility of Rank Discovery

In this subsection, we consider the feasibility of rank discovery over four types of ranking functions: (i) query-dependent, (ii) static, observable can be queried through

the interface, (iii) static, observable but cannot be queried, and (iv) static and proprietary, respectively. Figure 3.1 summarizes the following feasibility results:

Query-Dependent Ranking Functions: Note that rank discovery is *infeasible* for query-dependent ranking functions, unless the input tuple is returned in the top-k results. Intuitively, this is because, in order to get tuples beyond top-k, it is necessary to reformulate the query. But this has the side effect of arbitrarily changing tuple ranks. Hence, with a query-dependent ranking function, no mechanism can achieve rank discovery.

Observable and Queriable Ranking Functions: At the other extreme - when a ranking function is static, observable, and can be queried through the search interface - rank discovery can be reduced to the problem of aggregate query processing, which has been addressed in existing work [3]. The reason is simple - since `SELECT * FROM D WHERE` $f(t) = c$ is supported by the interface, the aggregate estimation algorithms in [3, 4, 2] can be readily applied to estimate `SELECT * FROM D WHERE` $f(t) > f(t_0)$, which is exactly the rank of input tuple t_0.

Observable and Non-Queriable Ranking Functions: When an observable ranking function is nevertheless not queriable through the interface, the simple solution described above no longer applies. Nonetheless, it is easy to see that rank discovery is always feasible - specifically, a naive approach is to crawl all tuples from the database, and then rank them locally according to observations of the scoring attribute.

> **Running Example:** Table 1 shows a simple table which we use as running example throughout this paper. There are $m = 5$ Boolean attributes and $n = 8$ tuples which are ranked in the order of their indices. i.e., t_1 is top-ranked.

Proprietary Ranking Functions: For proprietary ranking functions, a key concept for understanding the feasibility of rank discovery is the *direct domination* relationship between two tuples. To understand the concept, consider what rank-related

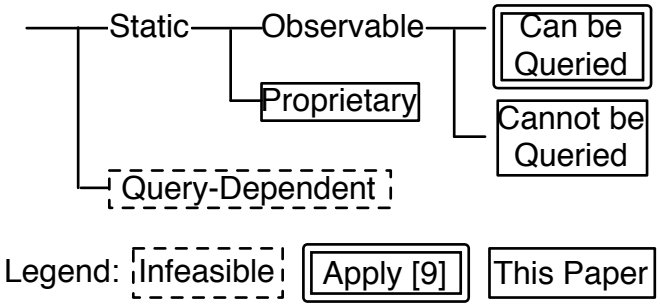


Figure 3.1. Rank Discovery: Taxonomy.

	A_1	A_2	A_3	A_4	A_5
t_1	0	0	0	0	1
t_2	0	0	0	1	1
t_3	0	0	1	0	1
t_4	0	1	1	1	1
t_5	1	1	1	0	1
t_6	1	1	1	1	1
t_7	1	0	0	0	0
t_8	0	0	0	0	0

Table 3.1. Rank Discovery: Database used in Running Example

information a query answer q reveals. It is easy to see that for the (at most) k tuples returned by q, their ranks can be compared according to the query answer. In addition, it is also possible to infer from q rank-related information for tuples that are *not* returned by it. Specifically, for two tuples t and t' which *match* q, we can determine which has a higher rank if (at least) one of them is returned by q:

- if q returns t but not t', then t is ranked higher,
- if q returns t' but not t, then t' is ranked higher, or
- if q returns both, then we can make the comparison based on the returned order.

For two given tuples, if there exists a query such that any of the three cases occurs, we say the two tuples are *directly comparable* with each other, with the higher-ranked tuple *directly dominating* the other one - i.e.,

Definition 2. [**Direct Domination**] *A tuple t is said to* ***directly dominate*** *another tuple t', i.e., $t \succ t'$, if and only if t and t' are directly comparable and t ranks higher than t'.*

For example, consider the running example shown in Table 1 with a top-2 interface. We can observe that t_1 and t_3 are directly comparable using the query q_1: `SELECT * FROM D WHERE` $A_1 = 0$ `AND` $A_2 = 0$ `AND` $A_4 = 0$ `AND` $A_5 = 1$ with t_1 ranked higher than t_3. Similarly, tuples t_2 and t_3 are directly comparable using the query q_2: `SELECT * FROM D WHERE` $A_1 = 0$ `AND` $A_2 = 0$ `AND` $A_5 = 1$. The result includes t_2 but not t_3 - i.e., t_2 ranks higher.

Given the direct domination relationships, the feasibility of rank discovery for a tuple t boils down to whether, for all other tuples t' in the database, it is possible to find a sequence of tuples $t_1, \ldots, t_h$, such that $t \succ t_1 \succ \cdots \succ t_h \succ t'$ or vice versa. If the chain can be found for all other tuples, then the rank of t can be precisely discovered. Otherwise, the fewer chains we can find, the wider a range we have to settle on estimating the rank. We call this problem the potential *discrepancy between real and revealed ranks*.

Fortunately, as we shall show in Section 3.6.2 and Section 3.7, while the discrepancy problem does exist in theory, the probability for it to occur in practice is extremely low - i.e., in almost all cases, the real rank is exactly disclosed by the top-k interface. Thus, rank discovery is indeed feasible for proprietary ranking functions.

Before concluding this subsection, we make an important observation that, if two tuples are directly comparable, then we need only *one* query to determine their domination relationship: the *most specific* query which matches both tuples - i.e., the query which contains one predicate for each attribute on which both tuples share the same value. To understand why, note that if this query cannot return at least one of the two tuples, then no other query can - i.e., the two tuples are not directly

comparable. For the running example, q_1 shown above is the most specific query matching t_1 and t_3, while q_2 is the one matching t_2 and t_3.

3.2.4 Formal Problem Definition

Discussions in the above subsection leave us with two types of ranking functions for which the rank discovery problem is feasible and unsolved - those that are proprietary, and those that are observable but cannot be queried. As discussed in Section 3.1, both types are widely prevalent in practice. Thus, we focus on solving the rank discovery problem for these two types in the paper.

Objective of Rank Discovery: Intuitively, the objective of rank discovery is to find the rank of a given tuple, i.e., the number of tuples with a higher rank than the given tuple, within a user-defined subset of the hidden database - which we model using a filtering query q_{F}. For example, a user may be interested in the rank of a car within all Honda Accords, in which case q_{F} is `SELECT * FROM D WHERE` Make = Honda `AND` Model = Accord. If a user is interested in the global rank within the entire database, then q_{F} is `SELECT * FROM D`. In this paper, we support any filtering query q_{F} as long as whether $t \in Sel(q_{\text{F}})$ can be determined solely upon knowledge of q_{F} and t (and not other tuples in the database[2]).

Given the closeness of real and revealed ranks as discussed in Section 3.2.3, we define the problem of rank discovery as the extraction of a tuple's rank revealed through the top-k interface. Specifically,

Definition 3. [Problem Definition] *Given a hidden database D and a filtering condition q_{F}, The objective of rank discovery is to compute $r(t, q_{\text{F}}) = |\Omega(t, q_{\text{F}})|$, where $\Omega(t, q_{\text{F}}) \subseteq Sel(q_{\text{F}})$ satisfies that $\forall t' \in \Omega(t, q_{\text{F}})$, either $t' \succ t$ or there exists tuples*

[2]e.g., queries such as `SELECT * FROM D WHERE` Price `> (SELECT AVG(`Price`) FROM D)` are not supported.

$t_1, \ldots, t_h \in D$, *such that* $t' \succ t_1 \succ \cdots \succ t_h \succ t$, *where* $\succ$ *is the direct domination relationship defined in Definition 2.*

Performance Measures: A key performance measure for rank discovery is *query cost* - i.e., the number of queries one has to issue through the web search interface of the hidden database. Accuracy-wise, we consider the *relative error* measure for an estimated rank $\tilde{r}(t, q_{\text{F}})$ as

$$\delta = \frac{|r(t, q_{\text{F}}) - \tilde{r}(t, q_{\text{F}})|}{r(t, q_{\text{F}})}. \tag{3.1}$$

Note that, compared with the absolute error measure (i.e., $|r(t, q_{\text{F}}) - \tilde{r}(t, q_{\text{F}})|$), relative error is more meaningful in practice. To see why, consider an example where an algorithm produces $\tilde{r}(t_1, q_{\text{F}}) = 200$ for a 100-th ranked tuple and $\tilde{r}(t_2, q_{\text{F}}) = 98761$ for a tuple with rank 98661. One can see that, while the error on t_1 represents a significant misconception of t_1's status in the database, the error on t_2 is hardly noticeable - yet both have the same absolute error of 100. According to this observation, we focus on the relative error measure in our theoretical analysis, while measuring both relative and absolute errors in the experiments section.

3.3 Overview of Technical Approach

In this section, we start by describing three baseline techniques and their respective problems. Then, we provide an overview of our technical approach to address these problems - with details discussed in Section 3.4 and 3.5. Note that, in this section and for most part of the paper, we focus on discovering the *global rank* of a tuple within the entire database (i.e., when q_{F} in Definition 3 is `SELECT * FROM D`). Then, we shall discuss in Section 3.6.1 a simple extension to support other q_{F}. To simplify the notations, we denote the global rank of t by $r(t) = r(t, \texttt{SELECT * FROM D})$.

3.3.1 Baseline Techniques

We start by describing three baseline ideas for solving the rank discovery problem, and point out their respective problems and/or unsolved technical challenges which motivate our proposed design of RANK-EST in this paper.

Crawling: The first baseline idea is to *crawl* all tuples and the associated rank information from a hidden database, and then rank all tuples locally (as in traditional databases) to derive the rank of the input tuple. The main problem of this approach is the extremely high query cost incurred by crawling, especially for large hidden databases. Lower-bound results derived in [5] show that crawling requires a prohibitively high cost of at least $\Omega(m \cdot n^2/k^2)$ queries for certain categorical databases with a top-k interface - where m and n are the number of attributes and tuples, respectively.

Sampling: The second baseline approach is to first draw a uniform random *sample* of the hidden database [33, 34, 35], and then compare the rank of the input tuple with all sample tuples to extrapolate its rank in the database. This approach has two main problems:

First, it is unclear how to effectively compare the ranks of two given tuples when the ranking function is proprietary. Note that while doing so for two directly comparable tuples (as defined in Definition 2) are easy, many pairs of tuples cannot be directly compared. For example, t_6 and t_8 in the running example do not have a direct domination relationship, because the only query which matches both tuples is `SELECT * FROM D` which returns neither. But one can still compare t_6 with t_8 by using t_7 as a "bridge", because t_6 is returned by $A_1 = 1$, while both t_7 and t_8 are returned by $A_5 = 0$. How to find such bridges, however, is a challenge for applying the sampling idea to proprietary ranking functions.

The second problem with this sampling-based approach is that the estimated rank may not be accurate enough for *highly ranked* tuples, unless one incurs a very high query cost to draw a large sample. To understand why, note that according to the accuracy measure in (3.1), to achieve the same accuracy level, the absolute error of rank estimation has to be much smaller for highly ranked tuples than lower ranked ones. On the other hand, it is easy to see that the sample size is inversely proportional to the square of absolute error - i.e., one needs a much larger sample for highly ranked tuples. For example, for a 1-million tuple database with $k = 50$, just to estimate a 100-th ranked tuple's rank within a relative error of 50%, the sampling-based approach needs at least an expected number of 1,000,000 / 100 = 10,000 samples - which could mean hundreds of thousands of queries even with the state-of-the-art sampler [35].

Ordered Crawling: The third baseline we consider is *ordered crawling*. It provides a remedy for the high-rank problem of the sampling-based approach without incurring as high a query cost as crawling all tuples. The key idea here is to crawl tuples in the descending order of their ranks - an operation enabled by the *getNext* primitive recently proposed for hidden databases with any static ranking function [36]. One can see that this method is capable of obtaining the rank of a highly ranked tuple without incurring the query cost for a complete crawl of the database.

The problem of ordered crawling is, still, its high query cost. Not only is it apparently unsuitable for lowly ranked tuples[3], even for highly ranked tuples the query cost can be very high. For example, when $k = 100$, calling the getNext primitive for a 2,000-th ranked tuple in a 200,000-tuple database requires nearly 60,000 queries [36] - prohibitively expensive in practice.

[3]Note that if the input tuple happens to be last-ranked, then this method is reduced to crawling the entire database.

3.3.2 Overview of Our Approach

Technical Challenges: Given the pitfalls of these three baseline approaches, we identify two key technical challenges for supporting efficient rank estimation:

- *Effective Rank Comparison:* To address the problem of applying sampling-based estimation to proprietary ranking functions, a key challenge is to efficiently compare the ranks of two tuples that do not have a direct domination relationship.
- *Efficient Rank Estimation for Highly Ranked Tuples:* Since the sampling-based approach is not effective for highly ranked tuples, a further challenge is to efficiently position a highly ranked tuple without crawling all higher-ranked ones.

Roadmap of Our Approach: In the remaining part of the paper, we shall address the two challenges respectively, before combining the techniques to form a comprehensive solution to rank discovery.

Specifically, we start by addressing the rank comparison problem for proprietary ranking functions in Section 3.4. We first describe a deterministic algorithm and its problem with high query cost - which motivates us to propose a efficient probabilistic solution. An interesting feature of this probabilistic solution is that it is a *Las Vegas algorithm* - i.e., it always produces the correct result[4]. In most cases, the algorithm terminates much sooner (i.e., requires much fewer queries) than the deterministic algorithm - only in the worst-case scenario will it be reduced to the deterministic algorithm itself.

Then, in Section 3.5, we address the rank-estimation problem for highly ranked tuples. Our key idea here is to first identify all queries which might reveal tuples with higher rank than the input tuple. Then, we select a small subset of these queries in

[4]as long as such a result is revealed by the top-k interface - which, according to Theorem 3, is highly likely.

a random yet judicious manner, and issue them to form a COUNT estimation for all higher-ranked tuples - without actually crawling the tuples.

Finally, in Section 3.6, we combine the techniques proposed in Section 3.4 and Section 3.5 to form our final rank estimation algorithm RANK-EST, which can efficiently yet accurately estimate ranks for both highly and lowly ranked tuples. Also in this section, we explain why we aim for rank "estimation" instead of precise "computation" in the paper by proving *hardness results*. The results show that it is not only impossible to precisely compute the rank of a given tuple in an efficient manner, even approximating the rank within a (small) fixed ratio mandates an extremely high query cost in the worst-case scenario.

3.4 Challenge 1: Rank Comparison

In this section, we address the first challenge - rank comparison for proprietary ranking functions. The key task here is to find a sequence of "bridge" tuples connecting the two inputs through direct domination relationships. We start by describing RANK-COMPARE, a deterministic yet inefficient algorithm to solve the problem. Then, we identify the fundamental problem underlying RANK-COMPARE's excessive query cost, and address it with LV-RANK-COMPARE, our randomized solution to the problem.

3.4.1 RANK-COMPARE

Description: The deterministic algorithm starts by testing if t and t' are directly comparable with each other. If not, it finds all tuples which directly dominate t, denoted by $\mathcal{D}(t)$, by issuing all 2^m queries which match t. Then, for each tuple t_i in $\mathcal{D}(t)$, RANK-COMPARE tests if t' directly dominates t_i - if so, then a bridge $t' \succ t_i \succ t$ is found. It does the same for t' - i.e., find $\mathcal{D}(t')$ and test if t dominates

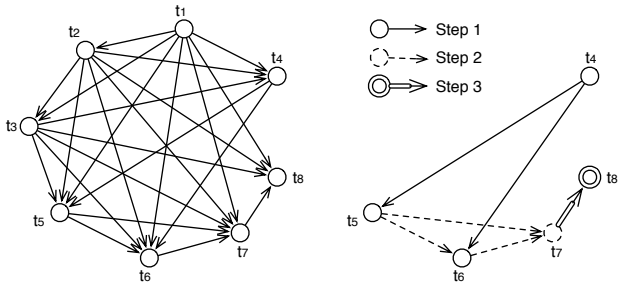


Figure 3.2. Rank Discovery: Iteratively build a bridge from t_4 to t_8.

any tuple within[5]. If no bridge is found, RANK-COMPARE identifies all tuples directly dominating (at least) one tuple in $\mathcal{D}(t)$ and $\mathcal{D}(t')$, respectively, and uses it to attempt building a two-hop bridge, and (if failed) repeats this process until finding a sequence of bridge-tuples $t_{\text{b}1}, \ldots, t_{\text{b}h}$, such that either $t \succ t_{\text{b}1} \succ \cdots \succ t_{\text{b}h} \succ t'$ or $t' \succ t_{\text{b}1} \succ \cdots \succ t_{\text{b}h} \succ t$. If no such a sequence can be found, then the top-k interface does not reveal enough information for comparing the ranks of t and t'.

In essence, this iterative process is a classic breadth-first-search-based graph reachability algorithm, if we consider all tuples in the hidden database as vertices and the direct domination relationships as edges. Figure 3.2 demonstrates such a correspondence and an example of building a bridge from t_4 to t_8 in the running example.

Pitfalls of RANK-COMPARE: An obvious problem of RANK-COMPARE is its query cost: in order to find a bridge connecting t with t', the iterative process of RANK-COMPARE repeatedly computes $\mathcal{D}(\cdot)$ which requires numerous queries when m, the number of attributes, is large. The key reason underlying this problem

[5]One might wonder why we do not pursue the other direction - i.e., instead of finding $\mathcal{D}(t)$, find the set of tuples which *are dominated by* t. The reason is that finding this set may require crawling the entire database - e.g., note that the tuples returned by SELECT * FROM D directly dominate all other tuples.

is the way RANK-COMPARE attempts to build the bridge. Specifically, RANK-COMPARE tends to waste a large number of queries testing tuples that are very *unlikely* to be on the bridge, as we shall show in the following example.

Consider the rank comparison between t_8 and t_4 in the running example, which is also illustrated in Figure 3.2. A key observation here is that $\mathcal{D}(t_8)$ includes not only the tuple that will eventually serve on the bridge (i.e., t_7), but a large number of other tuples (i.e., t_1, t_2, t_3). These tuples tend to be highly ranked - indeed, note that tuples returned by SELECT * FROM D directly dominate any other tuple in the database. As a result, they are highly unlikely to appear on the bridge, especially when t and t' have close yet low ranks, because any tuple which resides on a bridge between t and t' must be ranked between them. Unfortunately, RANK-COMPARE still wastes queries testing these tuples - wasting queries building a bridge-to-nowhere that has surpassed the rank of t'.

In the next subsection, we shall introduce our idea to significantly speed up the bridge-construction process by finding tuples that are most likely to serve on the bridge. Nonetheless, it is important to note that the uncertainty of bridge-construction, i.e., the lack of knowledge on which tuple to select next for building the bridge, is an inherent obstacle for any deterministic rank comparison algorithm. Indeed, we shall prove in Section 3.6.3 a negative results which shows that no deterministic algorithm can achieve a realistic worst-case query cost for rank comparison - a motivation for our proposal of a randomized algorithm.

3.4.2 LV-RANK-COMPARE

In this subsection, we start by describing the overarching scheme of LV-RANK-COMPARE, a *rank-based random walk with restart* on the domination-relationship graph (depicted in Figure 3.2). Then, we discuss two key design issues for random

walk, the selection of the next step and the decision of when to restart the random walk, before presenting the LV-RANK-COMPARE algorithm.

Random Walk For Bridge Construction: To compare the ranks of t and t', we perform the random walk both ways - i.e., we simultaneously start two random walks from t and t', respectively, until one walk reaches the other tuple. Without loss of generality, we consider the walk from t to t'. At each step, we choose a tuple from $\mathcal{D}(t)$, the set of tuples directly dominating t, by issuing a query matching t and finding a higher-ranked tuple from its answer[6]. We repeat this step to form a random walk $t \prec t_{\mathrm{b}1} \prec \cdots \prec t_{\mathrm{b}h}$. For each new tuple $t_{\mathrm{b}i}$ encounter in the walk, we issue one extra query (according to the method in Section 3.2.3) to determine if $t' \succ t_{\mathrm{b}i}$ - which indicates the successful construction of a bridge. Otherwise, we either continue or restart the random walk.

One can see that this vanilla algorithm does (partially) address the problem of RANK-COMPARE, as it now quickly "rejects" many tuples that overshoot t'. For example, if we apply this vanilla algorithm to the comparison of t_8 and t_4 in the running example, then the random walk from t_8 can quickly reject t_1, t_2, and t_3 if they are chosen as $t_{\mathrm{b}i}$, because these tuples directly dominate our target tuple t_4. Nonetheless, this vanilla approach still has the following two problems which adversely affect its query cost in practice:

- *Ineffective selection of $t_{\mathrm{b}i}$:* While the vanilla approach rejects many ineffective selections of $t_{\mathrm{b}i}$, it fails to carefully select the most favorable $t_{\mathrm{b}i}$ for building the bridge. Specifically, it does not consider the likelihood for a selection to be directly comparable with t'. It also fails to evaluate the chance for a selection to overshoot t' (before incurring query cost to test it).

[6]We shall discuss next how to choose which query to issue and which tuple to select.

- *Late Restart:* The vanilla method only restarts the random walk after reaching a tuple directly dominating t' - a strategy that could still lead to a long (yet wasteful) random walk consisting of tuples that far outrank yet are not directly comparable with t'. On the other hand, we found through real-world experiments two important observations about bridge construction: (1) a correct bridge very rarely involves more than a few (e.g., 2) tuples, and (2) there are a large number of such "short bridges". Thus, a more effective approach is to proactively restart a random walk (and hopefully hit one of the many other short bridges) instead of continuing on to a long path that is likely to have already outranked t'.

There are two key design issues one must address in this framework: (1) how to choose the next stop of random walk, $t_{\mathrm{b}i}$, from $\mathcal{D}(t)$, and (2) whether to restart or continue a random walk if a bridge is not (yet) found. We address the two issues respectively as follows.

Selection of $t_{\mathrm{b}i}$: For the first problem, we argue that an ideal selection of $t_{\mathrm{b}i}$ should satisfy the following two conditions:

α. The rank of $t_{\mathrm{b}i}$ should be as close to $t_{\mathrm{b}i-1}$ as possible, so as to ensure that the bridge does not overpass t'.

β. $t_{\mathrm{b}i}$ should share as many common attribute values with t' as possible, so as to increase the possibility for $t_{\mathrm{b}i}$ to be directly comparable with t'.

One might find Condition α counter-intuitive - rather than trying to make as much "progress" (i.e., rank-increase) as possible towards t', we are seemingly making the "progress" as little as possible. To understand the rationale behind Condition α, we make an important observation on what "progress" really means in the bridge-building process: Note that our objective is to find $t_{\mathrm{b}i}$ which is directly dominated by

t'. A key observation here is that, as long as t_{bi} does not "overshoot" t' (i.e., t_{bi} has a lower rank), whether t_{bi} is directly dominated by t' has nothing to do with the rank of t_{bi}, but (instead) is only determined by two factors: (1) the common attribute values shared between t_{bi} and t', and (2) given the most-specific query matching both t_{bi} and t', whether t' has a sufficiently high rank to be returned by the query. Since the rank of t' is solely input-dependent, Conditions α and β capture the two objectives under our control: Condition α aims to ensure that t_{bi} does not overshoot t', while Condition β aims to maximize the probability for t_{bi} and t' to be directly comparable.

In order to efficiently select t_{bi} according to the above two conditions, we start by finding queries matching t which are most likely to return tuples with close ranks to t. Specifically, we start with q : SELECT * FROM D, choose predicates matching t (i.e., $A_1 = t[A_1], \ldots, A_m = t[A_m]$) uniformly at random and add one at a time to q until reaching a query q' which returns t. For example, consider the rank comparison between t_4 and t_7 in the running example when $k = 3$. Suppose that, for the random walk from t_7, we happen to choose predicates $A_4 = 0$ and $A_1 = 1$ in order. We will stop at query q': SELECT * FROM D WHERE $A_4 = 0$ AND $A_1 = 1$, because t_7 is not returned by SELECT * FROM D or SELECT * FROM D WHERE $A_4 = 0$.

Then, from the tuples returned by q' which are ranked higher than t, we choose the one which has the most common attribute values with t', and use it as the next stop in the random walk. Note that a special case arises when t is the highest-ranked tuple returned by q'. In this case, we use the parent of q' (by removing the last-added predicate) to find the next stop.

Restart of Random Walks: We now consider the case where the random walk reaches a node t_{bi} that is not directly dominated by t'. An obvious condition for restarting the random walk is when t_{bi} indeed directly dominates t' - a clear evidence

of overshooting the target. However, if we only restart the random walk in this case, it is still possible for a long (yet wasteful) random walk to continue despite of reaching tuples that far outrank yet are not directly comparable with t'. To address this problem, we make two important observations from real-world experiments: (1) a correct bridge very rarely involves more than a few (e.g., 2) tuples, and (2) there are a large number of such "short bridges". Thus, a more effective approach is to proactively restart a random walk (and hopefully hit one of the many other short bridges) instead of continuing on to a long path that is likely to have already outranked t'.

According to these two observations, we introduce our strategy of proactively restarting a random walk. A key idea here is that, instead of directly limiting the length of a random walk, we instead place a upper bound c_{N} on the number of *new tuples* involved in a random walk (i.e., which have never been included in previous random walks). The purpose for doing so is to ensure the correct discovery of a bridge in the worst-case scenario where even the shortest bridge has a long length. As we shall show in Section 3.7, we found through real-world experiments that $c_{\mathrm{N}} = 3$ is often the optimal setting for hidden databases in practice.

3.4.3 Algorithm RANK-EST-S

Algorithms 3 and 4 depict the pseudocode of Algorithm LV-RANK-COMPARE and its usage in Algorithm RANK-EST-S, our sampling-based rank estimation algorithm, respectively. Note that in LV-RANK-COMPARE, we simultaneously build bridges from t and t', respectively, in order to enable rank comparison no matter which tuple is ranked higher.

Before concluding the section, we would like to note how LV-RANK-COMPARE compares against RANK-COMPARE. Note that, with the breadth-first scheme, RANK-COMPARE always identifies the *shortest* bridge from t to t'. Such an exhaustive

search is unnecessary when *any* valid bridge would suffice. The randomized algorithm LV-RANK-COMPARE instead aims to quickly identify the most likely path from t to t' which can serve as a bridge, thereby significantly reducing the query cost.

Algorithm 3 LV-RANK-COMPARE

Input : t and t', the tuples to be compared
loop
  Set $t_{b0} = t$, $t_{c0} = t'$, $l = 1$
  repeat
    Choose t_{bl} randomly from $\mathcal{D}(t_{bl-1})$
    Choose t_{cl} randomly from $\mathcal{D}(t_{cl-1})$
    return t **if** $t \succ t_{cl}$, or t' **if** $t' \succ t_{bl}$
  until bridges $t_{b0} \dots t_{bl}$ and $t_{c0} \dots t_{cl}$ has c unseen tuples
end loop

Algorithm 4 RANK-EST-S

Input : t , q_F
Take a random sample S from $Sel(q_F)$
n = Estimate of size of D from S
c = number of tuples in S that outrank t
return $\frac{c}{|S|} * n$

3.4.4 Bi-Directional Random Walk

In this subsection, we develop a novel idea of *bi-directional random walk* which further reduces the query cost of rank comparison. Recall from the previous sub-

section that, in order to verify $t \prec t'$, we always initiate the random walk from t, and then keep finding tuples directly dominating the previous ones before reaching t'. With bi-directional random walk, the path-finding effort works both ways: i.e., simultaneous with the random walk from t, we also initiate another random walk from t' which works backwards by choosing a tuple t'_{b1} from $\mathcal{D}'(t')$.

Before describing the details of our algorithmic design, we first explain why bi-directional random walks can significantly outperform their uni-directional counterparts. There are two main reasons: (1) query sharing between the two random walks, and (2) a significantly higher probability of finding a bridge.

First, note from the design of LV-RANK-COMPARE-UNI that, in order to handle both cases of $t \prec t'$ and $t \succ t'$, there are indeed two uni-directional random walks which start with the algorithm: one from t to a tuple in $\mathcal{D}(t)$ and the other from t' to a tuple in $\mathcal{D}(t')$. A key observation here is that, with the drill-down based method for selecting a tuple in $\mathcal{D}(t')$, it often takes no additional query to select a tuple from $\mathcal{D}'(t')$ as well. This can be seen from the following example. Consider again the drill-down path 4 in Figure 3.4. Once we reach the end of drill-down, we also get a tuple that is directly dominated by t_5: t_6 which is returned by $A_1 = A_3 = 1$. Note that t_6 naturally satisfies Condition α for building the bridge - as it is returned immediately after t_5. As such, to support a bi-directional random walk and sample $\mathcal{D}'(t')$, we only need to augment the drill-down process for sampling $\mathcal{D}(t')$ by continuing the drill-down until collecting k tuples that are dominated by t (or reaching a valid node, whichever comes first). From these k tuples, we choose the one that shares the most common attribute values with t - the same way as in the uni-directional version. One can see that this new process adds minimal additional query cost to the original uni-directional random walk.

Second, the bi-directional random walk significantly increases the success probability for bridge construction. To understand why, note that a uni-directional random walk from t only terminates (successfully) when the newly added tuple $t_{\mathrm{b}i}$ is directly dominated by t'. With a bi-directional random walk, however, successful termination can be triggered by $t_{\mathrm{b}i}$ being dominated by any of the tuples at the opposite direction - i.e., $t', t'_{\mathrm{b}1}, \ldots, t'_{\mathrm{b}i}$. In other words, a bi-directional random walk succeeds as long as any pair of tuples $\{t_{\mathrm{b}u}, t'_{\mathrm{b}v}\}$ $(u, v \in [0, i])$[7] are directly comparable with each other. One can see that the probability for constructing a bridge now increases roughly quadratically with the length of the random walk, rather than linearly as in the uni-directional version (barring the random walk overshooting the destination - i.e., $t'_{\mathrm{b}i} \prec t$ or $t_{\mathrm{b}i} \succ t'$). As such, the idea of bi-direction random walk significantly increases the probability of bridge construction while issuing only a small number of new queries.

Algorithm 5 depicts our final LV-RANK-COMPARE algorithm which features bi-directional random walks. In the experimental evaluations in Section 3.7, we shall show that LV-RANK-COMPARE significantly outperforms LV-RANK-COMPARE-UNI (and RANK-COMPARE) in terms of query cost over real-world hidden databases.

3.5 Challenge 2: Handling Highly Ranked Tuples

We now consider how to enable rank discovery for highly ranked tuples (when the ranking function is either proprietary or observable). We start by describing a deterministic solution RANK-COMPUTE, before introducing a randomized algorithm RANK-EST-H with a significantly lower query cost. As we show in §3.7, RANK-EST-H is the algorithm of choice for as much as top 25% of the database (after which RANK-EST-S becomes competitive).

[7]Let t and t' be $t_{\mathrm{b}0}$ and $t'_{\mathrm{b}0}$, respectively.

Algorithm 5 LV-RANK-COMPARE

1: **Input parameters:** t and t' : input tuples to compare
2: **loop**
3: // Case $t \succ t'$
4: Start *forward* random walk r_1 from t to t'.
5: Start *reverse* random walk r_2 from t' to t.
6: **return** t **if** $r_1 \cap r_2 \neq \emptyset$
7: // Case $t' \succ t$
8: Start *forward* random walk r_1 from t' to t.
9: Start *reverse* random walk r_2 from t to t'.
10: **return** t' **if** $r_1 \cap r_2 \neq \emptyset$
11: **end loop**

3.5.1 RANK-COMPUTE

Before introducing RANK-COMPUTE, we would like to first note that, while the concept of direct domination relationship was introduced in §3.2 for proprietary ranking functions, it readily applies to observable ones as well. Even though direct domination relationships (or chains of them) are no longer needed for comparing observable ranks, the concept is still important for understanding how to retrieve tuples that outrank the input, as we show below.

To count the rank of t, we have to consider two types of tuples: those in $\mathcal{D}(t)$ - i.e., directly dominating t - and those that outrank t but are not directly comparable with t, which we denote to as $\mathcal{D}_{\text{i}}(t)$. RANK-COMPUTE crawls all tuples in $\mathcal{D}(t)$ and $\mathcal{D}_{\text{i}}(t)$ in an iterative fashion - i.e., it starts by computing $\mathcal{D}(t)$, followed by computing $\mathcal{D}(\cdot)$ for each tuple in $\mathcal{D}(t)$, and does so iteratively until no new tuple is found. One

can see that the key challenge here is to efficiently compute $\mathcal{D}(\cdot)$ - which we address next.

Computing $\mathcal{D}(t)$: As mentioned in Section 3.2.3, a baseline method for computing $\mathcal{D}(t)$ is to issue all $\binom{m}{0} + \cdots + \binom{m}{m} = 2^m$ queries which match t. To reduce the large query cost of 2^m, we consider a query-issuing strategy described as follows: First, we organize these 2^m queries into a lattice structure - an example of which (when $t = t_7$ in the running example) is shown in Figure 3.3. One can see that the root of the lattice is `SELECT * FROM D`, while the bottom is the fully specified, m-predicate, query. Each node on Level-i represents a query with i conjunctive predicates (all matching t).

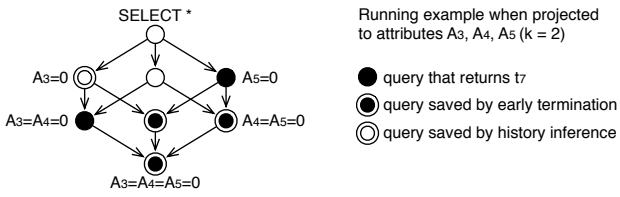


Figure 3.3. Rank Discovery: Illustration of Lattice and Two Ideas.

Instead of issuing all 2^m queries in the lattice, as in the baseline, we reduce the query cost according to two key ideas. One is *early termination* - i.e., if a query q in the lattice returns t, then we do not need to issue any (lower-level) successors of q because any tuple returned by these queries which ranks higher than t must also be returned by q. In the running example depicted in Figure 3.3, no descendants of $A_5 = 0$ needs to be issued according to early termination, because $A_5 = 0$ already returns t_7.

The next cost-saving strategy is *history inference*. Note that this includes not only the simple leverage of historic queries - i.e., do not issue a query if it has been

issued before - but also the non-trivial inference of a query answer from a collection of historic queries. Specifically, consider the case where, before issuing a query q, we have already obtained answers to all predecessors of q in the lattice (i.e., queries on the path between the root and q) and, among the returned tuples, at least k of them match q. In this case, we do not need to issue q because it is impossible for it to return any tuple we have not yet seen. For example, in Figure 3.3 we do not need to issue $A_3 = 0$ due to history inference, because the root `SELECT * FROM D` returns t_1 and t_2 which have $A_3 = 0$.

Algorithm 6 RANK-COMPUTE

1: **Input:** t; **Output:** $r(t)$

2: $H(t) = \{ \mathcal{D}(t) \}$ (set of tuples ranked higher than t)

3: **repeat** $H(t) = H(t) \cup \mathcal{D}(t') \ \forall t' \in H(t)$

4: **until** no new tuples added to $H(t)$

5: **return** $|H(t)| + 1$

Leveraging both ideas, RANK-COMPUTE collects $\mathcal{D}(t)$ by performing a breadth-first search (BFS) of the lattice. In the design of lattice-BFS, we skip any query that can be inferred from history, and invoke early termination if reaching a node that returns t.

Theoretical Analysis: Algorithm 6 depicts the pseudocode of RANK-COMPUTE. The following theorem provides an upper bound on the query cost of RANK-COMPUTE.

Theorem 2. *The worst-case query cost for RANK-COMPUTE to compute $r(t)$, i.e., the rank of a tuple t, is $r(t) \cdot \binom{m}{\lfloor m/2 \rfloor}$.*

We do not include the proof of this theorem due to space limitations. Note that, for computing $\mathcal{D}(t)$, RANK-COMPUTE requires at most $\binom{m}{m/2} = O(2^{\sqrt{m} \cdot \log m/2})$ queries - a significant reduction from the 2^m queries required by the baseline method. For example, when $m = 12$, the query-cost is reduced from 4,096 to 924.

3.5.2 RANK-EST-H: A Randomized Algorithm

Even though RANK-COMPUTE requires much fewer queries than the baseline method, it can still generate excessive query cost when either m or $r(t)$ is large. For example, the upper bound derived in Theorem 2 exceeds 25,200 queries for a 100-th ranked tuple in a 10-attribute database. This can be attributed to : (1) the iterative process for computing $\mathcal{D}_{\text{i}}(t)$ (tuples that outrank but are not directly comparable with t), as many tuples are repeatedly retrieved in this iterative process, and (2) the actual crawl of all tuples in $\mathcal{D}(t)$ and $\mathcal{D}_{\text{i}}(t)$, when only COUNT is required by rank estimation. We address these two problems respectively as follows.

For the computation of $\mathcal{D}_{\text{i}}(t)$, a key observation here is that the higher ranked t is, the smaller $|\mathcal{D}_{\text{i}}(t)|$ is likely to be. To understand why, recall from Section 3.2 that in order for two tuples t and t' to be *not* directly comparable with each other, the most concrete query q which matches both tuples must not return either of them. One can see that clearly, the higher ranked t and t' are, the less likely it is for q to return neither of them. Thus, $|\mathcal{D}_{\text{i}}(t)|$ is likely to be small for a highly ranked input tuple t. This can be observed from the running example - for the highest ranked 6 tuples $t_1, \ldots, t_6$, $\mathcal{D}_{\text{i}}(\cdot)$ are all zero. On the other hand, the 8-th ranked tuple t_8 has 3 tuples (i.e., t_4, t_5, t_6) in $\mathcal{D}_{\text{i}}(t_8)$.

This key observation leads us to a simple yet (somewhat) crude way of estimating $r(t)$: crawl the lattice structure to retrieve all tuples in $\mathcal{D}(t)$, and then use $|\mathcal{D}(t)|$ as an estimation for $r(t)$. Nonetheless, the second problem (i.e., crawling $\mathcal{D}(t)$

Algorithm 7 RANK-EST-H

1: **Input :** t , q_F

2: Randomly drill down on query lattice augmented with q_F for t until some query node N_q returns t

3: $\mathbb{D}(t)$ = set of tuples returned by ancestors(N_q) and dominate t

4: **return** $\sum_{t_i \in \mathbb{D}(t)} \frac{1}{p(t_i)}$

when only COUNT is required) remains. To address this problem, our main idea is to enable an efficient estimation of $|\mathcal{D}(t)|$ by performing a *random drill-down* from the top of the lattice as depicted in Figure 3.4 - i.e., starting from the root, we choose a branch uniformly at random, and then repeat this process to "drill down" deeper into the lattice in order to sample each tuple dominating t with a positive probability. Figure 3.4 depicts examples of drill downs for the lattice of t_5 in the running example when projected to attributes A_1, A_2, A_3.

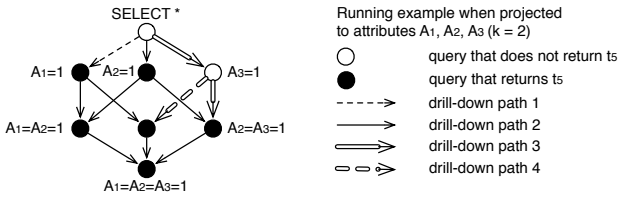


Figure 3.4. Examples of Drill Downs.

Let $t_1, \ldots, t_w$ be the tuples retrieved during this drill down which directly dominate t. One can see that, by applying the Horvitz-Thompson estimator [37], an unbiased estimate for $|\mathcal{D}(t)|$ is $\sum_{i=1}^{w} 1/p(t_i)$, where $p(t_i)$ is the probability for t_i to be picked up by such a random drill-down process, because its expected value

$$E\left[\sum_{i=1}^{w} \frac{1}{p(t_i)}\right] = \sum_{t' \in \mathcal{D}(t)} \left(p(t_i) \cdot \frac{1}{p(t_i)}\right) = |\mathcal{D}(t)|. \tag{3.2}$$

Unfortunately, computing $p(t_i)$ proves to be challenging because a tuple directly dominating t may be returned by multiple drill-down paths. For example, in Figure 3.4, t_4 in the running example may be returned by drill down paths 2, 3 and 4. In addition, note that different drill-down paths are taken with different probability - e.g., while path 2 in Figure 3.4 is taken with probability 1/3, path 3 is take with only 1/6 probability. As such, one may not be able to precisely compute $p(t_i)$ without issuing additional queries after the drill-down process.

To address this challenge, we consider the following heuristics: if, during the current drill down, tuple t_i is first returned at Level-h of the lattice[8], then we assume that all other nodes at Level-h which match t_i also return t_i - and no node above Level-h returns it. With this heuristics, we now compute an estimation of $p(t_i)$ as follows. Note that the probability for the random drill-down process to reach an h-th level node is $1/\binom{m}{h}$. Thus, for a tuple t_i which shares the same value as t on c attributes (note that $c \geq h$ due to the lattice definition), we have $p(t_i) \approx \binom{c}{h} / \binom{m}{h}$.

For example, in Figure 3.4, when we retrieve t_4 at the Level-1 node $A_2 = 1$, the estimate[9] of $p(t_4) \approx \binom{2}{1}/\binom{3}{1} = 2/3$. One can see that we can now estimate $|\mathcal{D}(t)|$ accordingly. For example, if we happen to take drill down path 2 in Figure 3.4, our estimation for $\mathcal{D}(t_5)$ is $\mathcal{D}(t) \approx 2/1 + 1/(2/3) = 3.5$, leading to a rank estimation of 4.5 for $r(t_5)$. Note that we may repeat the random drill down for multiple times (and take the average of estimations) to improve the estimation accuracy for $|\mathcal{D}(t)|$.

[8]Let the root level be Level 0.

[9]where $c = 2$, $m = 3$, and $h = 1$. Note that $m = 3$ because the lattice represents a projection to A_1, A_2, A_3. $c = 2$ because t_4 and the input t_5 have two attributes in common among A_1, A_2, A_3.

3.6 Algorithm RANK-EST

In this section, we start by describing our final RANK-EST algorithm. Then, we tackle two theoretical issues, (1) the possible discrepancy between real and revealed ranks, and (2) the hardness of exact rank computation, respectively.

3.6.1 Description of RANK-EST

Interleaving: From the previous discussions, one can see that the RANK-EST-S and RANK-EST-H have complementary behavior - i.e., RANK-EST-S works poorly for highly ranked tuples, which RANK-EST-H specifically address. We now consider the integration of these two algorithms to produce RANK-EST which work universally for all tuples in the database. Our main idea is to *interleave* the two algorithms. In particular, we first take a pilot sample of the hidden database and use RANK-EST-S to produce a (roughly) estimated rank. If the confidence interval of the estimation falls below a threshold[10], then t likely has a high rank - thus we switch to RANK-EST-H. Otherwise, we continue with the sampling process in RANK-EST-S to reduce the estimation error.

Extension to other q_{F}: So far, we focused on the case where the user-specified filtering query q_{F} in Definition 3 is `SELECT * FROM D`. We now consider the extension to other q_{F} for RANK-EST-S and RANK-EST-H, respectively. Note that, for RANK-EST-S, there is indeed no revision required for handling other q_{F}, as long as the sampling algorithm we call as a subroutine only generates samples that satisfy q_{F}. Then, by calling on existing aggregate estimation algorithms (e.g., [3]) to estimate the COUNT of tuples satisfying q_{F}, we can readily generate the rank of a tuple within q_{F}.

[10] e.g., for a 95% confidence interval $r(t) \in [u, v]$, if $v < \ell$ where ℓ is the pre-determined threshold

For RANK-EST-H, a simple revision is required - in Algorithm 7, when computing $\mathbb{D}(t)$, i.e., the set of tuples returned during drill-down which dominate t, we only include those tuples in $\mathbb{D}(t)$ that satisfy q_{F}. With the revision, the estimation we generate is for the number of tuples in $\mathcal{D}(t)$ matching q_{F}. Note that, if q_{F} happens to be a conjunctive query that can be specified through the top-k interface, then it is possible to further improve the efficiency of RANK-EST-H by *appending* the predicates in q_{F} to every query in the lattice (on which we perform the random drill-downs, as shown in Figure 3.4). This way, all queries issued by RANK-EST-H are focused on tuples matching q_{F}, leading to a reduced query cost. Algorithm 8 depicts the interleaved RANK-EST generic to q_{F}.

Algorithm 8 RANK-EST

1: **Input :** t , q_F **Output :** $\tilde{r}(t)$

2: Fetch pilot sample S from $Sel(q_F)$

3: $\tilde{r_c}(t)$ = Approximate rank estimated by RANK-EST-S(t, q_F)

4: **if** confidence interval of $\tilde{r_c}(t) < threshold$ **then** Estimate $\tilde{r}(t)$ through RANK-EST-H(t, q_F)

5: **else** Continue estimation of $\tilde{r}(t)$ through RANK-EST-S(t, q_F)

3.6.2 Closeness of Real and Revealed Ranks

In the following discussion, we first describe why this problem exists in theory, and then show that it is extremely unlikely to occur in practice. To understand why the a tuple's true rank might differ from what is revealed through the top-k interface, consider a database of two Boolean attributes and three tuples: $t_1 : \langle 0, 1\rangle$, $t_2 : \langle 0, 0\rangle$, and $t_3 : \langle 1, 1\rangle$, with t_1 and t_3 having the highest and lowest rank, respectively,

according to a hidden ranking function. One can see that, if the database has a top-1 interface, then it is impossible to determine which of t_2 and t_3 ranks higher, because the only query that matches both tuples, i.e., `SELECT * FROM D`, returns neither. This example illustrates that, while the true rank of t_3 should be 2 (because two tuples rank higher than it), the best estimation one can make from the top-k interface is 1 - leading to a significant difference between the two values.

Fortunately, we found through theoretical analysis and experimental studies that such a difference is usually extremely small to non-existent in real-world hidden databases, mainly because of two reasons. First, real-world databases often feature a much larger k (than 1), revealing a lot more information about the rank comparison between different tuples. Second, there are often many more attributes, making it unlikely for two highly ranked tuples to be incomparable with each other. The following theorem uses a special case to illustrate the extremely small value of the difference. We shall further evaluate such a difference value with real-world datasets in the experiments section.

Theorem 3. *Consider a database with m attributes, each of which is generated i.i.d. with uniform distribution over a domain size of c. For a tuple t with real rank r and top-k-interface-revealed-rank r', there is* $\Pr\{|r - r'| > \epsilon\} < p \cdot (2p)^{\epsilon-1}$*, where p is upper-bounded by: (note that* $\text{erf}(\cdot)$ *is the error function)*

$$\sum_{i=0}^{m} \left[\binom{m}{i} \cdot \frac{(c-1)^{m-i}}{c^m} \cdot \left(1 - \sum_{j=0}^{k-1} \binom{r}{j} \cdot \frac{(c^i-1)^{r-j}}{c^{i \cdot r}} \right) \right]$$
$$\leq \sum_{i=0}^{m} \left[\frac{\binom{m}{i} \cdot (c-1)^{m-i}}{2c^m} \cdot \left(1 - \text{erf}\left(\frac{(k-1) \cdot c^{i/2} - r}{c^i \sqrt{2r(c^i-1)}} \right) \right) \right]$$

One can see from the theorem that, the larger k is or the smaller m and r are, the smaller the probability for $|r - r'| > \epsilon$ will be - irrelevant of how many tuples the database contains. Specifically, for a 15-attribute database with $k = 100$, the largest

(i.e., worst-case) probability for any tuple (with arbitrary r) to have a relative rank difference of 2% (i.e., $|r - r'| > 0.02 \cdot r$) is lower than 0.00058 - indicating that the top-k interface likely reveals a very accurate rank for each tuple in the database.

3.6.3 Hardness Results

We now consider the hardness of random comparison and computation for a hidden web database. Before constructing the detailed proof, we first briefly describe the intuitive idea behind the hardness results.

Recall the query lattice defined in Section 3.4. One can see that, when there is a large number of attributes in the database, the query cost of rank comparison/computation can be very high if it requires the enumeration of queries at the middle level (i.e., $\lfloor m/2 \rfloor$-th level) of the lattice - because this middle level contains the most queries (i.e., $\binom{m}{m/2}$). To understand how rank comparison/computation may require issuing such middle-level queries, consider an example where one needs to compute the rank of a given tuple t which has such a low rank that it is not returned by any query above the $\lfloor m/2 \rfloor$-th level in the lattice. Note that, unfortunately, this scenario can happen even when the database is very small - with as few as $m/2 + 1$ tuples, as we shall show in the proof. In this case, to determine if there is a tuple t' which (1) shares the values of $A_1, \ldots, A_{m/2}$ with t and (2) directly dominates (or is directly dominated by) t, one has no choice but to issue `SELECT * FROM D WHERE` $A_1 = t[A_1]$ `AND` $\cdots$ `AND` $A_{m/2} = t[A_{m/2}]$ because:

- any query with fewer predicates would not help to determine if $t \succ t'$, because t is not returned by such a query
- any query with more predicates would not help because it does not match t'.

Intuitively, one can see that such a scenario may force the issuing of all queries in the middle level of the lattice.

According to the intuition, we establish the hardness of rank comparison, (exact) rank computation, and approximate rank computation, respectively as follows.

Theorem 4. (Hardness of Rank Comparison) *Given two comparable tuples t and t', no algorithm can guarantee correct comparison with $o(m^{\sqrt{m}/2})$ queries.*

Proof. We construct the worst-case scenario for the proof as follows. We start by considering a top-1 interface. Without loss of generality, suppose that t is all-zero while t' is all 1 (i.e., $\forall i \in [1, m], t[A_i] = 0, t'[A_i] = 1$). Let $h = \lfloor \sqrt{m} \rfloor$. For the sake of simplicity, we assume h exactly divides m. Consider the following h tuples to be the highest-ranked tuples in the database. Note that $t_i = A_1, A_2$ means that t_i has $A_1 = 0$ and $A_2 = 0$, and value 2^{11} for the other attributes.

t_1: $A_2, \ldots, A_h, A_{h+2}, , A_{m-1}, A_m$

t_2: $A_1, A_3, , A_h, A_{h+1}, A_{h+3}, , A_{m-1}, A_m$

t_3: $A_1, A_2, A_4, , A_h, A_{h+1}, A_{h+2}, A_{h+4}, , A_{m-1}, A_m$

...

t_h: $A_1, , A_{h-1}, A_{h+1}, , A_{2h-1}, A_{2h+1}, , A_{m-1}$

The construction follows the following rule: for any i ($i \in [1, h]$), t_i has value 0 on all attributes except $A_i, A_{h+i}, ,$ - i.e., all A_j where $j \equiv i \mod h$. A key implication of this rule is that, for any query q of the form

$$q : A_{i_1} = 0 \ \texttt{AND} \ \cdots \ \texttt{AND} \ A_{i_h} = 0 \tag{3.3}$$

where $\forall j \in [1, h]$, $i_j \equiv j \mod h$, q satisfies two properties:

- q does not match any of $t_1, \ldots, t_h$, and
- once we remove any single predicate from q, the new query matches at least (indeed exactly) one tuple in $t_1, \ldots, t_h$.

[11] Note that this can be any arbitrary value that except t's value (i.e., 0) and t''s value (i.e., 1).

Let $Q = \{q_1, \ldots, q_{\binom{m}{h}}\}$ be the set of all queries of the form (3.3). Note that the size of Q, i.e., $\binom{m}{h} = O(m^{\sqrt{m}/2})$. For each query $q_i \in Q$, we construct a tuple $t_{\text{q}i}$ which has value 0 on all attributes specified in q_i, and 1 otherwise. An important property of $t_{\text{q}i}$, according to the two above-described properties of q, is that the rank comparison between t and $t_{\text{q}i}$ is only inferable from one query - i.e., q_i - because q_i is the most-predicate query which matches both t and $t_{\text{q}i}$, and any shorter query returns neither of them (as it has to return one of the highest ranked tuples $t_1, \ldots, t_h$).

Consider a hidden database of $2 + h + \binom{m}{h}$ tuples - i.e., t, t', $t_1, \ldots, t_h$, and $t_{\text{q}1}, \ldots, t_{\text{q}\binom{m}{h}}$. We first consider the following high-to-low rank assignment: t_1, ..., t_h, an arbitrary order of $t_{\text{q}1}, \ldots, t_{\text{q}\binom{m}{h}}$, and at last t, t'. Note that the correct output here for a rank-comparison algorithm is "CANNOT COMPARE", because it is impossible for one to infer the rank comparison between t and t' from queries returned by the web interface. Specifically, note that the only query which matches both t and t' is `SELECT * FROM D` which returns neither of them. In addition, no other tuple can serve as a "bridge" between the two tuples as they rank immediately next to each other in the assignment.

Suppose a rank-comparison algorithm were able to generate the correct output with $o(m^{\sqrt{m}/2})$ queries. One can see that there is at least one query in Q which the rank-comparison algorithm did not issue. Let such a query be q_i. Consider the following rank assignment. t_1, ..., t_h, an arbitrary order of $t_{\text{q}1}$, ..., $t_{\text{q}(i-1)}$, ..., $t_{\text{q}(i+1)}$, ..., $t_{\text{q}\binom{m}{h}}$, and at last t, $t_{\text{q}i}$, t'. Note that the correct output of a rank-comparison algorithm is now "t-outranking-t'", because (1) q_i returns t, indicating that t dominates $t_{\text{q}i}$, and (2) $t_{\text{q}i}$ and t' are always directly comparable, as the most specific query which matches both tuples only indeed matches these two tuples. One can see that this leads to a contradiction, as the new rank assignment does not change any query answer but q_i, which was not issued by the rank-comparison algorithm.

Thus, no rank-comparison algorithm can guarantee correct output with $o(m^{\sqrt{m}/2})$ queries.

Note that while the proof assumes $k = 1$, when $k > 1$ we can simply extend the proof by duplicating each of $t_1, \ldots, t_h$ for k times (which still reveals all tuples through the top-k interface). □

One can see that the theorem precludes the existence of efficient deterministic algorithms for rank comparison when there is a large number of attributes (e.g., when $m^{\sqrt{m}/2} \gg n$). The following corollaries further eliminate the possibility of having a worst-case-efficient algorithm for computing the exact rank of a given tuple, or even approximating the rank with a deterministic error bound, when there is a large number of attributes in the database and/or the attribute domain sizes are unbounded. On the other hand, we shall show in Section 3.7 that our randomized algorithms LV-RANK-COMPARE and RANK-EST usually requires far fewer queries for real-world datasets.

Corollary 1. (Hardness of Obtaining the Exact Rank) *For a given tuple t, no algorithm can guarantee the computation of the rank of t with $o(m^{\sqrt{m}/2})$ queries. If the domain sizes of attributes are sufficiently large, no algorithm can guarantee rank computation with $o(m^{\sqrt{m}/2} + n^n)$ queries.*

Proof. The first part of the corollary follows directly from Theorem 4, because if one were able to compute the rank of a tuple with $o(m^{\sqrt{m}/2})$ queries, then simply computing the ranks of both t and t' would solve the rank-comparison problem with $o(m^{\sqrt{m}/2})$ queries. We now focus on proving the second part of the corollary.

Recall $h = \lfloor \sqrt{m} \rfloor$ and Q being the set of queries with form (3.3). We now construct a case which shows that, when the domain of each attribute is sufficiently large (specifically, ¿ 2^m), then even for a hidden database with as few as $h+1$ tuples,

one needs to issue at least $|Q|$ queries in order to guarantee the rank computation for t in the worst-case scenario. Since $n = h + 1$ in this case, the construction proves that no algorithm can exactly compute the rank of a tuple with $o(n^n)$ queries, thus completing the proof.

The construction is simply a top-1 interface over a database with an all-zero t as well as $t_1, \ldots, t_h$ as defined in the proof of Theorem 4. The reason why $|Q|$ queries are needed for the rank computation of t can be seen from the following three observations: (1) If, after finishing the rank computation algorithm, one cannot determine whether a query $q \in Q$ matches any tuple other than the given t, then we can always insert such a tuple to the database, rank it directly above t, without changing any query answer one has seen so far. Note that the assumption of sufficiently large domain sizes is used here, as the newly insert tuple needs to avoid all non-zero values included in the issued queries. (2) No query "shorter" than q discloses any information about the answer to q because it always overflows (matching the given tuple t and at least one tuple in $t_1, \ldots, t_h$). (3) No query "longer" than q discloses any information about the answer to q, because it does not match the tuple we propose to insert in the first observation. As such, no algorithm can correctly compute rank without issuing all queries in Q. □

Corollary 2. (Hardness of Rank Approximation) *For a given tuple t, no algorithm can generate a value v such that the rank of t is guaranteed to be in $[v, v \cdot r]$ with $o(m^{\sqrt{m}/2} - \sqrt{m} \cdot r)$ queries. If the domain sizes of attributes are sufficiently large, no algorithm can do so with $O(m^{\sqrt{m}/2} + n^n - \sqrt{m} \cdot r)$ queries.*

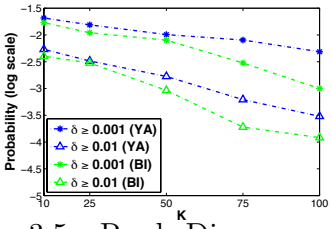


Figure 3.5. Rank Discovery: Evaluating Rank Discrepancy.

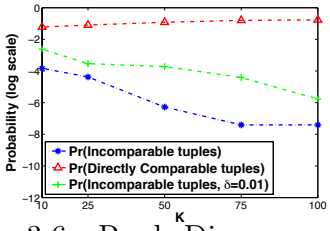


Figure 3.6. Rank Discovery: Evaluating Comparability of Tuples.

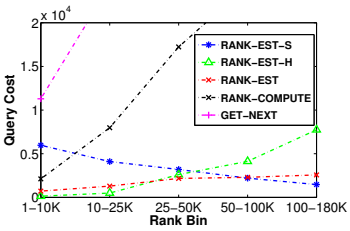


Figure 3.7. Rank Discovery: Varying Input Rank (RE).

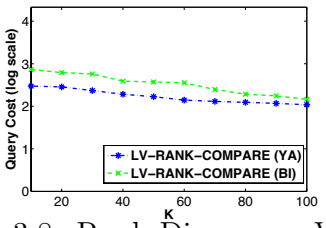


Figure 3.8. Rank Discovery: Varying k (RC).

3.7 Experimental Results

3.7.1 Experimental Setup

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

Datasets: Our primary dataset consists of data crawled from the Yahoo! Autos (YA)[12], a real-world hidden database. It contains 200,000 used cars for sale in the Dallas-Fort Worth metropolitan area. There are 32 Boolean attributes such as A/C, Power Locks, etc, and 6 categorical attributes, such as Make, Model, Color, etc. The domain size of categorical attributes ranges from 5 to 16. We also tested our algorithms over a synthetic boolean dataset (Bool-IID or BI) of 10 million tuples and 100 attributes, primarily for scalability purposes. The tuples are generated as i.i.d. data with each attribute having probability of $p = 0.5$ to be 1. For both

[12]http://auto.yahoo.com

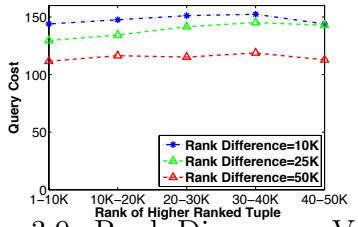


Figure 3.9. Rank Discovery: Varying Input Rank (RC).

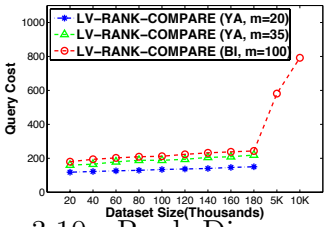


Figure 3.10. Rank Discovery: Varying n (RC).

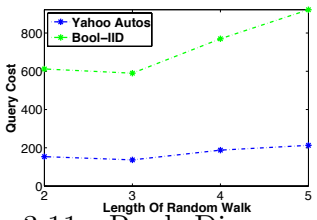


Figure 3.11. Rank Discovery: Varying c_N (RC).

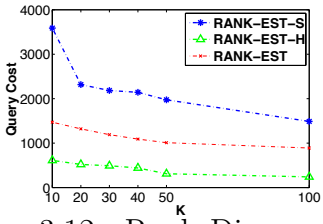


Figure 3.12. Rank Discovery: Varying k (RE).

datasets, the tuples were reordered using a random permutation - with the order then being used as a static ranking function for all experiments. We set $k = 100$ unless otherwise specified. Our charts primarily report the results over for Yahoo! Autos (unless otherwise specified) as the results on the synthetic dataset were similar.

Real-World Online Experiment: We also tested our algorithms online via Amazon.com's Product Advertising API[13]. The API reveals a top-100 interface with sales rank being the ranking function, yet does not reveal the actual rank (by default). To uncover the ground truth, we found that the individual item description provided by Amazon.com reveals the sales rank of certain items[14]. As such, we chose all testing tuples from those that have its real rank disclosed in the description. Specifically, we focused on Amazon's DVD and book items, and constructed search queries using 15 categorical attributes such as Actor, Artist, etc. Amazon.com has a limit of 2,000 queries per IP address per hour.

[13]https://affiliate-program.amazon.com/gp/advertising/api/detail/main.html

[14]Many others, e.g., item No. B009B0JR2C, do not reveal the rank at all.

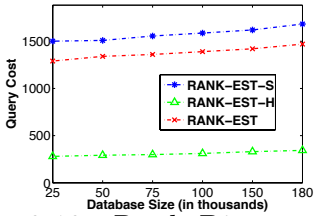


Figure 3.13. Rank Discovery: Varying n (RE).

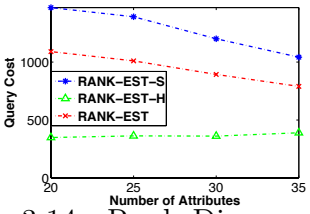


Figure 3.14. Rank Discovery: Varying m (RE).

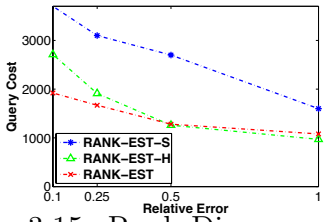


Figure 3.15. Rank Discovery: Trade-off (RE).

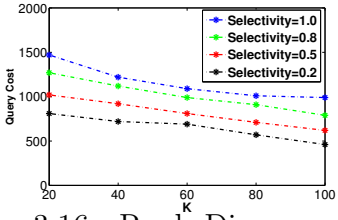


Figure 3.16. Rank Discovery: Varying Selectivity of q_F.

Algorithms: We tested 4 algorithms for rank estimation considered in the paper: RANK-COMPUTE, RANK-EST-S, RANK-EST-H and RANK-EST. Since, RANK-EST-S uses algorithm LV-RANK-COMPARE for rank comparison, we also perform a rigorous set of experiments to evaluate it. For RANK-EST-S and RANK-EST, we used the existing HIDDEN-DB-SAMPLER [33] for the sampling primitive. We also tested as baseline the direct usage of an existing algorithm GetNext [36] for rank computation. Since GetNext is only capable of obtaining the $(h+1)$-th ranked tuple based on the top-h tuples, to use it for rank computation we ran the algorithm repeatedly until reaching the input tuple.

Performance Measures: For all algorithms, we measure efficiency through query cost. In addition, we measure the accuracy of rank estimation through the *relative error* measure defined in Section 3.2.4.

3.7.2 Experimental Results

In this subsection, we first empirically evaluate the discrepancy between real and revealed ranks. Then we describe the results of our offline experiments over Yahoo! Auto dataset and online experiments at Amazon.com for the rank estimation problem.

Empirical Evaluation of Rank Discrepancy and Tuple Comparability : Recall that our aim is to estimate the revealed rank of a given tuple. We observed that problem of discrepancy between real and revealed is exceedingly unlikely in practice. For each tuple in both datasets, we computed its revealed rank using RANK-COMPUTE algorithm and compared the relative error between the real and revealed ranks. Figure 3.5 shows the fraction of tuples that had a rank discrepancy of 0.1% and 1% (or above) for different values of k. For $k = 100$ (a fairly common value), less than 400 tuples out of 200,000 tuples had a rank discrepancy of 1%. This justifies our problem definition in terms of revealed rank and that rank discrepancy is not a big issue in real-world databases.

Figure 3.6 shows the feasibility of rank comparison problem over Yahoo! Autos dataset. The results for Bool-IID dataset was similar. Notice that the probability of finding a pair of tuples that are not comparable is exceedingly low (on the order of 10^{-7} for $k = 100$). We observe that the fraction of tuple pairs that are directly comparable varies between 10-20% and increases with larger values of k. It shows that while almost all pair of tuples are comparable only a fraction of them are directly necessitating practical algorithms. Finally, we identify the fraction of pairs of tuples that have a relative rank difference of 1% (e.g. when tuples ranked 990th to 1010th were compared with the 1000th-ranked tuple). Intuitively, these are the hardest pairs to test as with larger rank difference, the chance of identifying a bridge

dramatically increases. We observed that even for this restrictive scenario, only a minuscule fraction of tuples remain incomparable.

Comparison of Rank Estimation Algorithms: We start by comparing our algorithms with the direct usage of GetNext [36] while varying the rank of the input tuple between 1 and 180,000. Figure 3.7 depicts the (average) query cost required to achieve a relative error of 10% on rank estimation. The query cost of GetNext is only plotted for input rank [1, 10000], because it exceeds all of our algorithms by orders of magnitude in all other categories. We also evaluated our deterministic algorithm RANK-COMPUTE. While this algorithm is much more efficient that GetNext, its query cost rapidly increases with the tuple's rank and becomes prohibitive. One can also observe from the figure that, as discussed in Section 3.5, RANK-EST-S works better for lowly ranked tuples, while RANK-EST-H works better for highly ranked ones. By interleaving the two, RANK-EST works effectively for tuples of all ranks. Given the excessive query cost of RANK-COMPUTE, we only focus on the practical algorithms RANK-EST, RANK-EST-S and RANK-EST-H for comparative analysis.

Before thoroughly evaluating different facets of the rank estimation algorithms, we first evaluate the sub problem of rank comparison as its solution is used by both RANK-EST-S and RANK-EST.

Rank Comparison: We start by evaluating the efficacy of of our Las Vegas algorithm, LV-RANK-COMPARE. Specifically, we randomly selected two tuples with rank between 42,000 and 44,000, and measured the query cost of comparing them while varying k between 10 and 100. One can see from Figure 3.8 that, LV-RANK-COMPARE is extremely practical and identifies the correct comparison rapidly. No-

tice that the query cost decreases rapidly as k increases as more and more tuples becomes directly comparable.

We tested the performance of LV-RANK-COMPARE while varying the ranks of input tuples. Figure 3.9 depicts the results when the rank difference between the two input tuples varies from 10,000 to 50,000, while the higher-ranked tuple is randomly selected from one of the five rank-buckets ranging from [1, 10000] to [40001, 50000]. One can see from the figure that, consistent with intuition, our algorithm requires fewer queries when the ranks of input tuples are further apart. In addition, the performance of our algorithm is not significantly affected by the absolute rank of the input tuples.

We also tested the scalability of LV-RANK-COMPARE by varying n, the number of tuples, and m, the number of attributes. To do so, we sample tuples and attributes uniformly at random (without replacement) from the Yahoo! Auto and Bool-IID datasets. One can see from Figure 3.10 that our query cost increases slowly with n. Note that the jump at the right side of the figure is because we include at the end of x axis the results for Bool-IID when the dataset contains 5 or 10 million tuples - demonstrating the scalability of LV-RANK-COMPARE. In addition, our query cost actually *decreases* with a large m. The reason for the latter is that, when m is larger, the number of tuples directly comparable with t also increases - leading to a higher probability of bridge construction by LV-RANK-COMPARE.

Finally, we tested our parameter setting for c_{N} - i.e., the upper bound on the length of a random walk before triggering proactive restart. One can observe from Figure 3.11 the justification for our heuristics of $c_{\mathrm{N}} = 3$ for both datasets. We observed similar result in our online experiments also. As discussed in Section 3.5, further increasing c_{N} leads to a higher query cost because, when the random walk

"overshoots" the destination, it takes longer to restart before finding one of the many short bridges.

Rank Estimation: Similar to Figures 3.8 and 3.10 for rank comparison, we tested the performance of our rank estimation algorithms against varying k, n and m, respectively, with results depicted in Figures 3.12, 3.13 and 3.14. All these figures depict the number of queries required for reaching a relative error of 10% for rank estimation. For Figures 3.13 and 3.14, we randomly chose the input tuples for RANK-EST-H and RANK-EST-S from rank-bucket [10K, 20K] and [50K, 100K], respectively. For RANK-EST, we randomly chose the input from the entire database. One can see from Figures 3.12 and 3.14 that, as expected, our algorithms require fewer queries when k or m is larger. Figure 3.13, on the other hand, demonstrates the scalability of our algorithms to larger databases. Figure 3.15 further depicts the tradeoff between query cost and the relative error of rank estimation. We also tested the impact of selectivity of different queries q_F. We constructed the filtering queries by first randomly deciding the total number of attributes in q_F which (along with their values) are then chosen randomly. We then chose an arbitrary tuple from $Sel(q_F)$ and estimated its rank within it. The results for different queries with varying level of selectivity are provided in Figure 3.16. As expected, when queries become highly selective, the query cost to estimate its rank drops.

Online Experiments at Amazon.com: Before presenting the results of our online experiments, we would like to note that Amazon's interface provides no efficient way to crawl a large number of lowly ranked tuples without exceeding the query allowance. As such, for the purpose of our experiments, we focused on tuples with rank between 1 and 15,000. Figure 3.17 shows the results. We first tested our LV-RANK-COMPARE algorithm by randomly selecting two tuples with rank difference varying between 1,000

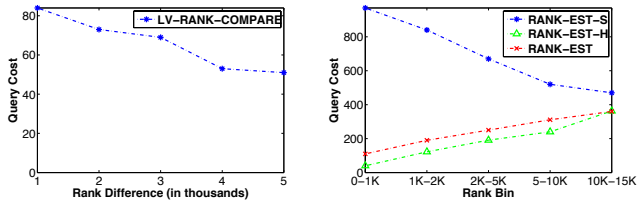


Figure 3.17. Rank Discovery: Rank Comparison and Estimation at Amazon.com.

and 5,000. We can see that our algorithms require fewer than 100 queries for rank comparison. For rank estimation, we can observe that our RANK-EST algorithm requires fewer than 400 queries - far below the hourly limit of 2,000 queries imposed by Amazon - to reach an estimation error of 10%. In addition, the pattern of query-cost change with input rank is consistent with the offline case. Interestingly, RANK-EST-H consistently outperforms RANK-EST-S in our online experiments - indicating that the rank-bucket [1, 15000] we were able to use is still (relatively) highly ranked in the large Amazon database. It is also important to note that our final RANK-EST algorithm only has a slightly higher query cost than RANK-EST-H, indicating our algorithm's ability to quickly switch from RANK-EST-S when the input tuple is highly ranked.

3.8 Related Work

Data Analytics over Hidden Databases: There has been prior work on crawling, sampling, and aggregate estimation over the hidden web, specifically over text [23, 24] and structured [17] hidden databases and search engines [25, 26, 27]. Specifically, sampling-based methods were used for generating content summaries [28, 29, 30], processing top-k queries [31], etc. Prior work (see [3] and references therein) considered sampling and aggregate estimation over attribute values *explicitly* returned by the web interface of the structured hidden database. Our work, on the other hand,

considers rank information of tuples which is *not explicitly* returned, thereby precluding the applicability of prior work. Our paper differs from top-k processing (see [32] for a survey) as our paper aims to discover rank-related information from the top-k answers, instead of studying how th e top-k answers can be retrieved.

Retrieving Rank Information from Hidden Databases: GetNext operator [36] allows an ordered crawling of top ranked tuples for any static ranking function. Specifically, given all tuples ranked in top-(h-1) as input, GetNext operator retrieves the No. h ranked tuple. While [36] solves the problem of retrieving top-h tuples over a top-k interface (where $h > k$), our paper initiates the first formal study on efficiently estimating the rank of any given tuple in a hidden web database. Admittedly, it is possible to use GetNext in a brute-force way to solve the rank computation problem - i.e., by retrieving all tuples ranked higher than the input tuple in an iterative fashion. However, such an approach is prohibitively expensive for all but the highly ranked tuples. Further, it produces an *exact* rank whereas for a number of scenarios as outlined in Section 3.1, *approximate* rank with low query cost is preferable. *Suggestion sampling* [23] estimates the frequency of search queries from ranked lists returned by a search engine's prefix-matching auto-complete interface. Unlike [23], we consider the retrieval of rank information from a structured hidden database with a form-like interface.

Information Integration and Extraction for Hidden databases: A significant body of research has been done on information integration and extraction over hidden databases - see tutorials [15, 16]. Due to space limit, we only list a few closely-related work: [17] proposes a crawling solution. Parsing and understanding web query interfaces has been extensively studied (e.g., [18, 19]). The mapping of attributes across different web interfaces has been addressed in (e.g., [20]). For integrating query interfaces for multiple web databases in the same topic-area see [21, 22]. Our

paper differs from top-k processing (see [32] for a survey) as our paper aims to discover rank-related information from the top-k answers, instead of studying how the top-k answers can be retrieved.

3.9 Final Remarks

In this paper, we defined a novel problem of rank discovery from hidden web databases with restrictive top-k search interfaces. We first introduced a taxonomy of ranking functions according to multiple dimensions, discussed the feasibility of rank discovery for each type of ranking function, and described solutions for all the feasible types. We proposed RANK-EST, a randomized algorithm for efficient rank discovery for proprietary and observable ranking functions. We proved hardness results that preclude the existence of efficient deterministic algorithms. We demonstrated the effectiveness of our proposed algorithms using real-world datasets and also through an online experiment conducted over Amazon.com.

CHAPTER 4

Aggregate Estimation Over a Microblog Platform

4.1 Introduction

The Microblogs Query Aggregation Problem: Online microblogging platforms have experienced a phenomenal growth of popularity in recent years, because they offer easy and compelling ways for millions of users to post content and interact with each other. In addition to providing attractive mediums for person-person interactions, microblogging platforms also offer unprecedented opportunities for microblog data analytics, i.e., big-picture views of what people are saying, because they contain a deluge of opinions, viewpoints, and conversations by millions of users, at a scale that would be otherwise impossible to gather using more traditional methods such as controlled surveys. In fact, microblog service providers such as Twitter and their partners are already attempting to analyze their data, ranging from public opinion to spatiotemporal popularity of topics, and using the results to build advertising campaigns or monitor the reputation of companies.

Although these are important applications for companies, microblogging platforms also provide free (but limited) public access to their data in the form of restricted APIs, which offer great opportunities for other, often non-commercial applications, such as the type of studies that would be most useful to a social scientist. For example, a social researcher may wish to analyze publicly available microblog conversations and postings to determine the change in general public's attitudes on individual privacy before and after the news of Edward Snowden's leakage of NSA surveillance became public. Other examples can include studies of the spread of

obesity-promoting attitudes, the mechanisms of bullying in colleges or schools, and the early detection of suicidal discourse.

A core functionality to facilitate such analytics is to answer *aggregate queries* over publicly available microblog data, which is the focus of this paper. An example of aggregate query is "How many Twitter users used the keyword `privacy` in 2013?". We shall consider SUM, COUNT, AVG queries on various attributes of microblog users or posts (e.g., users' age or posts' length), with selection conditions on keywords and other attributes like time.

We emphasize that our techniques will necessarily generate approximate answers; exact answers are infeasible since they require access to the complete data (and are also often unnecessary in many applications, since approximate aggregates are usually sufficient for obtaining "big-picture" views of the data). Our methods should be efficient in the following sense - the number of API calls made to the microblogging service provider should be as few as possible in generating the approximate aggregate.

Limitations of Existing Microblog APIs: Many of the popular microblog sites like Twitter, Tumblr, Instagram, Yammer, Weibo, Identi.ca (and some other social networking sites like Google+ and Facebook that also offer microblogging features) offer search API calls, which allow retrieving posts containing query keywords, but the results are limited, e.g., past week in Twitter Search API [38]. Other microblogs limit the maximum number of search results one could retrieve to at most a few thousands.

A notable exception to such search APIs is Twitter's Streaming API, which allows retrieving large numbers of posts given keyword and other conditions[1]. Unfor-

[1]If no condition is specified, the streaming API returns a ∼1% sample of all tweets - a ratio too small to reliably compute many aggregates (e.g., those that are conditioned on a keyword).

tunately, the streaming interface only allows retrieving Twitter postings *in the future*, and there is no way to obtain historical tweets. Thus, a a sociologist will never be able to study the origin of a trending keyword unless he/she is somehow (magically) able to predict such a keyword ahead of time. Note that there are companies like GNIP [39] and Datasift [40] that sell historic microblog data; however the subscription fee is often rather high (e.g., $3,000 per month for Twitter alone at Datasift.com [40]) for a non-commercial setting such as social science research.

Limitations of Previous Research on Estimation of Aggregates on Social Networks: There has been work on estimating aggregate functions on social networks [41, 42, 43, 44]. These works generally use random walk-based sampling on the social graph, or adaptations of it like Metropolis-Hastings [45]. However, they are inefficient for the type of aggregate queries that we study for the following reasons: They only consider *broad aggregates*, that is, aggregates on the whole social network, and not constrained by keywords. Most of these techniques enable aggregate estimation by drawing a random sample of *all* microblog users, and extrapolating from the sample. For our purpose, however, aggregate queries have keyword selection conditions that match only an extremely small fraction of these users - e.g., the number of Twitter users who have used the keyword `privacy` in their postings is only 0.4% of all active users. A straighforward solution would be to only consider users who satisfy the selection condition during the sampling random walk. However, we found that this leads to a social subgraph with tightly connected communities that that significantly increase its convergence time (its *burn-in* period). Further details of the limitations of these techniques are discussed in Section 4.4 and Section 4.7.

Outline of Our Results: We develop MICROBLOG-ANALYZER, an efficient platform to enable the accurate estimation of aggregate queries over an online mi-

croblogging service. Its design is based on a central and novel idea: to leverage the *user-timeline interface* (offered by most microblogging platforms) to bypass the above-described limitations on the search API. The user-timeline API inputs a user-id and returns all (for all practical purposes as discussed in Section 4.2) public posts generated by the user.

MICROBLOG-ANALYZER operates as follows: we start from a user who recently generated a post satisfying the aggregate query keyword condition (e.g., who is returned by the search API), and then traverse a carefully constructed *subgraph of the social graph*, where users are nodes and user connections are edges, according to the aggregate query, in order to sample (and retrieve through the user-timeline interface) a small number of user timelines based on which we generate our aggregate estimation. There are two main technical issues facing this design: (1) how to design the aggregate-dependent subgraph, and (2) how to traverse such a subgraph, in order to enable efficient and accurate aggregate estimations. We propose two novel ideas to address these challenges:

First, we propose a *level-by-level subgraph* to address the issue of subgraph design. Specifically, we introduce a novel taxonomy of user connections (i.e., edges) based on the aggregate being estimated and user timelines. A critical feature of this taxonomy is our finding that, while certain types of edges are beneficial to efficient sampling, others are detrimental to it and should be *removed* from the graph. We adjust the original social graph according to this taxonomy to produce the level-by-level subgraph and, by performing simple random walks [46] over it, develop MICROBLOG-ANALYZER-Simple Random Walk (MA-SRW), our first algorithm for aggregate estimations over a microblog platform. We present theoretical analysis and real-world experiments to demonstrate the superiority of MA-SRW over several baseline graph designs.

Then, to address the graph traversal issue, we develop a *topology aware random walk* over the level-by-level subgraph. Previous random walk techniques (e.g., as used in MA-SRW) are oblivious and therefore generic to the topology of the underlying graph. This often requires a large query cost for the so-called burn-in period [45] in order for the sampling probability of each node to converge to a stationary distribution, so that the sampled nodes can be used for aggregate estimations. We show that, by leveraging knowledge of the underlying graph topology - specifically, the level-by-level structure - our traversal algorithm removes the need of this burn-in period (and the associated query cost) - enabling a significantly more efficient and accurate aggregate estimation process. The execution of topology-aware random walk over the level-by-level graph forms our final algorithm, MICROBLOG-ANALYZER-Topology-Aware Random Walk (MA-TARW).

Summary of Contributions:

- We define the novel problem of aggregate estimation over historic microblog data (Section 4.2). We develop a novel idea leveraging the user-timeline access provided by online microblogs to bypass the limitations they place on the search API, and present a platform to tackle the aggregate estimation problem (Section 4.3).
- To effectively sample the social graph according to an aggregate query, we develop a level-by-level subgraph topology and demonstrate through theoretical analysis and experimental results its superiority over a number of baseline graph designs (Section 4.4).
- To efficiently sample a level-by-level graph, we develop a topology aware random walk which leverages the special properties of a level-by-level graph topology

to significantly outperform baseline solutions such as traditional random walks (Section 4.5).

- We present comprehensive experiments on Twitter, Google+ and Tumblr that show the significant improvement our methods offer compared with the state-of-the-art (Section 4.6).

4.2 Problem Definition

In this section, we start with describing a data-access model that abstracts the API interfaces provided by most popular microblogs, and then define the problem of aggregate estimation.

Model of Microblog Data Access: In general, a microblogging platform offers three functionalities: (1) share concise updates in text (e.g., Twitter, Google+, Tumblr), image (e.g., Instagram), or video (e.g., Vine); (2) form social connections with each other (e.g., follower/followee in Twitter, Circles in Google+, Likes in Tumblr); and (3) search, subscribe to, and consume the updates posted by users. Correspondingly to these three functionalities, most microblogging platforms - e.g., Twitter, Tumblr, Instagram, Google+, Weibo, Yammer etc. allow the following three types of *queries*:

1. `SEARCH`: Given a keyword (or keywords) w, return *recent* micro-posts that contain w. Most microblog sites only return posts in recent weeks – e.g., the last weeks posts in Twitter API [38]. Other microblogs restrict search to top-k results where k could be in the low thousands. They do so for two main reasons: recent data are generally more interesting to users, and many microblog service providers consider selling access to historic data an important monetization channel [40, 39].

2. `USER CONNECTIONS`: Given a user u, return all other users "connected" with u. Note that "connections" here are loosely defined - they can be follower/followee relationships (as in Twitter), friendships (as in Friendfeed), etc. Almost all real-world microblogs, e.g., Twitter, Instagram, Tumblr, allow complete access to all user connections (unless a user sets it to private).
3. `USER TIMELINE`: Given a user u, return all posts published by u. To simplify the taxonomy, we assume that a user timeline query also returns the user's profile information (e.g., name, demographics). Like user connection queries, real-world microblogs seldomly limit the returned user timelines, with one notable exception of Twitter which only publishes the most recent 3200 tweets published by a user. Nonetheless, according to recent studies, only a very small fraction of extremely prolific users - 5%[47] - have posted more that 3,200 tweets and even for these users only very old tweets are missing, in contrast to the search API that only goes back one week [38] (e.g., even Justin Bieber only posted 2,500 tweets between Apr and Dec 2013). Given that in this paper we focus on aggregate estimations, it is safe to assume that this small number of incomplete user timelines has little effect on the estimated aggregates.

Note that the above interfaces could alternatively be implemented through *web crawling* of the microblog site if an API is not available. However, web search interfaces often have unknown selection and ranking criteria that make them less desirable for aggregate estimations - e.g., in Twitter, posts may be missing from the web search but not from the search API results [38]; similarly Tumblr and other sites often perform unpredictable query expansion at their Web search interface. Further, many sites do not allow web-page scraping, e.g., as specified in Twitter (https://twitter.com/tos).

Another important interface limitation imposed by microblogging APIs is an upper bound on the number of queries a user can issue in a time period. For example,

Twitter's search API [38] allows only 180 queries over a 15 minute window, and Reddit API allows no more than one request every two seconds.

Problem Definition: In this paper, we address the problem of aggregate estimations over microblogs by issuing queries through the above-described limited microblog interface. Specifically, we consider aggregate queries of the form `SELECT AGGR(`$f(u)$`) FROM` U `WHERE CONDITION` where U is the set of all users, $f(u)$ is any function that returns a numeric measure for each user u (e.g., age or #connections), `AGGR` is an aggregate function such as COUNT, SUM or AVG, and `CONDITION` determines whether a user$u$ should be considered for (i.e., included in) the aggregate.

It is important to note that the above-described form covers not only aggregates over users, but also *aggregates over posts* as well. For example, the COUNT of posts containing keyword `privacy` can be specified as follows: `CONDITION` returns TRUE if a user has `privacy` appearing in its timeline, and FALSE otherwise; $f(u)$ returns the number of posts containing `privacy` in the user's timeline; and `AGGR` is SUM.

While many different predicates can be specified in `CONDITION`, we highlight two specific types: (a) *keyword predicates* - i.e., a user is included iff its timeline contains a pre-determined keyword (e.g., `privacy` in the above example); (b) *time window* - e.g., users who mentioned `privacy` from Jul to Dec 2013. Keyword predicates are prevalent in aggregates required by social science studies because most of these studies focus on one or a few topics specifiable as keywords. For this reason, in this paper we focus on aggregate queries with at least one keyword predicate, optionally a time window, as well as other other predicates on a user's profile attributes (e.g., gender, age, number of connections).

Performance Measures: The performance of an aggregate estimation algorithm is measured in terms of efficiency and accuracy. Given the query-rate limit enforced by

most microblogging platforms, the efficiency is the *query cost* - i.e., the number of queries and/or API calls (on `SEARCH`, `USER CONNECTIONS`, and `USER TIMELINE`) the algorithm issues to the microblog.

For accuracy, given estimation $\tilde{\theta}$ of an aggregate θ, we apply the standard measure of *relative error* $|\tilde{\theta} - \theta|/\theta$. Note that the error is determined by two factors[2]: *bias*, i.e. $E(\tilde{\theta} - \theta)$, and *variance* of $\tilde{\theta}$.

Hence, given an aggregate query with keyword and other predicates, the objective of the *microblog aggregate estimation problem* studied in this paper is to produce an estimation while minimizing both query cost and relative error.

4.3 Overview of MICROBLOG-ANALYZER

This section overviews MICROBLOG-ANALYZER, our system for enabling analytics over a microblog by issuing queries through its limited access interface. We start by presenting a key idea of MICROBLOG-ANALYZER: estimating aggregates by sampling user timelines. Then, we outline the design issues associated with two main components of MICROBLOG-ANALYZER: (1) GRAPH-BUILDER, i.e., the generation of a conceptual graph that connects user timelines together, and (2) GRAPH-WALKER, i.e., the design of an efficient sampling algorithm over such a graph. While Section4.4 and Section4.5 describe these two components in detail, we discuss at the end of this section how we prototyped MICROBLOG-ANALYZER over Twitter and collected ground-truth for its evaluation.

4.3.1 System Architecture

Figure 4.1 depicts the architecture for MICROBLOG-ANALYZER which has two main components: (1) GRAPH-BUILDER that builds a graph connecting users

[2]Specifically, the mean squared error MSE = bias^2 + variance.

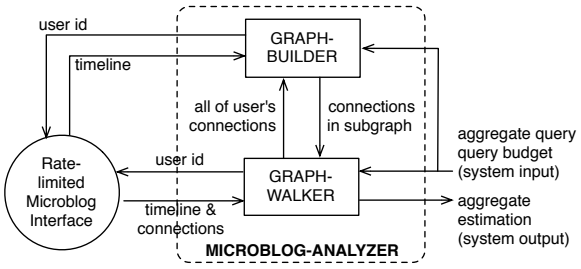


Figure 4.1. Microblog Analyzer: System Architecture.

and (2) GRAPH-WALKER that performs a random walk over such a graph. The system works as follows:

- It receives as input an aggregate query to be estimated (as defined in Section4.2), a query budget (i.e., the maximum number of queries MICROBLOG-ANALYZER can issue to the microblog), as well as one or a few "seed users" which have posted microblogs satisfying the selection condition of the aggregate. Note that such seed users can be easily identified through the limited search API (e.g., for Twitter, users who posted a keyword in the past week).
- Given a seed user, MICROBLOG-ANALYZER uses the GRAPH-BUILDER to determine which other users are its *neighbors*. As we shall show later, the design of GRAPH-BUILDER can range from simply using all social connections of the user to a carefully designed algorithm that takes into account the aggregate being estimated and certain user timeline information to select a subset of such social connections. We shall discuss the design of this component in the next subsection and then in detail in Section4.4.
- Given the neighbors, GRAPH-WALKER determines the probability for MICROBLOG-ANALYZER to "transit to" and sample each neighbor for aggregate estimation. Once again, the design ranges from simply choosing each neighbor uniformly at random (i.e., simple random walk) to a carefully designed algorithm that takes

into account certain topological properties of the graph produced by GRAPH-BUILDER. We shall discuss the design of this component in the next subsection and then in detail in Section4.5.

- The above process can be repeated multiple times until exhausting the query budget, so as to produce a more accurate aggregate estimation as the final output of MICROBLOG-ANALYZER.

Algorithm 9 MICROBLOG-ANALYZER

```
1: Retrieve seed users
2: while Remaining query budget > 0 do
3:     Invoke GRAPH-BUILDER to retrieve user neighborhood
4:     Invoke GRAPH-WALKER and transition to new user
5:     Perform aggregate estimation if possible
6: end while
```

4.3.2 Key Idea: User-Timeline Based Analytics

Feasibility of User-Timeline Based Analytics: To address the often stringent limit on search query interfaces, a key data source MICROBLOG-ANALYZER leverages is the user timeline - i.e., all historic posts published by a user - which, as discussed in Section4.2, is readily accessible through the access interface of many microblogs.

To understand why user-timeline information can be used to answer aggregate queries (especially those with keyword predicates) defined in Section4.2, we start by considering an extremely inefficient technique which nevertheless demonstrates the feasibility of this idea. Note that, as shown by many previous studies [41, 43, 48],

the vast majority of users in a microblogging service are linked in a connected graph through social relationships revealed by the service - e.g., follower/followee in Twitter, Circles in Google+, blog followers in Tumblr, comments on same post in Reddit, etc. For the purpose of this paper, we consider such a *social graph* to be undirected. For directed relationships such as follower/followee on Twitter, one can easily convert them to undirected edges by considering two users to be connected if either follows the other.

Given the social graph, one can simply start with one user and recursively follow edges (using user connections API) to reach and crawl the timeline of most users - making it possible to answer aggregates based on the locally crawled data. While this brute-force method demonstrates the feasibility of acquiring sufficient information (for aggregate estimation) through user-timeline queries, it unfortunately requires a prohibitively high query cost given the access-rate limit discussed in Section4.2. In addition, most crawled data would be completely useless for aggregate estimation - e.g., even for a broad query like the count of users who have tweeted `privacy` in 2013, the vast majority of user timelines would be irrelevant because only a very small percentage ($\approx 0.4\%$ of its active users) of all Twitter users satisfies the selection condition - leading to a significant waste of resources.

To address this problem, MICROBLOG-ANALYZER only *samples* users who satisfy the keyword predicate specified in the aggregate query, and then produce aggregate estimations according to the collected sample. Corresponding to the two components GRAPH-BUILDER and GRAPH-WALKER in the system, there are two design issues that are critical for enabling the sampling-based method:

Design Issue 1 (Subgraph Generation): A straightforward method to sample user timelines is to perform a *random walk* over the social graph - e.g., a simple random

Table 4.1. Microblog Analyzer: Components employed by proposed algorithms

	GRAPH-BUILDER	GRAPH-WALKER
MA-SRW	Level-by-Level (Section 4.4)	Level-by-Level (Section 4.4)
MA-TARW	Simple RW [46]	Topology-Aware RW (Section 4.5)

walk [46] recursively jumps from one user to one of its neighbors chosen uniformly at random - so timelines of sample users (taken after a sufficient number of "burn-in" transitions [45]) can be used for aggregate estimations. A problem with this method, however, is that topology of the social graph is very "*unfriendly*" for sampling and requires a high query cost for random walks to "burn-in". While we shall discuss this finding in detail in Section4.4, an intuitive explanation here is that the social graph contains many "redundant" edges which may "trap" [3] a random walk inside a tightly connected component - i.e., preventing the walk from efficiently sampling all nodes in the graph.

More importantly, it was recently found [49] that the burn-in period required for many social-relationship-induced graphs is much longer than anticipated - again leading to problems with the rate limit of microblogging services. Estimating aggregates which cover only a small portion (albeit still a large absolute amount) of microblog users requires an extremely large number of samples. The reason is simple: Consider an AVG query such as the average age of users who mentioned `Privacy` in 2013. Since the standard error of estimation is inversely proportional to $\sqrt{s \cdot r}$, where s is the sample size and r is the fraction of users who satisfy the selection condition, a small r (e.g., $\approx 0.4\%$ for users who mentioned `Privacy` in Twitter in 2013) mandates a very large sample size s to reach a reasonable level of accuracy. Indeed, this

[3]Note that unlike a spider trap for random walks over a directed graph, here a random walk can still exit the component - albeit with a small probability.

problem occurs for most aggregates concerning one or a few keywords, because rarely any garners the attention of more than 1% of all microblog users [48].

As such, to enable efficient sampling, the first design issue we must address is how to "on-the-fly" remove the redundant edges and find a *subgraph* that satisfies two conditions: (1) *high recall:* it still includes most if not all users who satisfy the selection condition of the aggregate to be estimated, and (2) *sampling-friendly:* the subgraph should have a "well-knit" [46, 50] topology and therefore facilitate an efficient random walk process. One can see that the high-recall requirement ensures the closeness of estimations generated from the subgraph to the ground truth, while the friendliness requirement ensures an efficient random walk process. We shall develop a novel technique for subgraph construction in Section4.4.

Design Issue 2: Sampling Design: In the above discussions, we considered a direct application of traditional random walk techniques (e.g., simple random walk [46] or Metropolis-Hastings random walk [45]) over the user-timeline graph (or subgraph, once the above design issue is addressed). While there has been a large body of work on using these random walks for aggregate estimation over large graphs [46, 42, 51, 43, 44] a key deficiency of it is the significant query cost required by answering COUNT and SUM queries.

While samples collected by random walks can be directly used to estimate AVG queries (as a weighted average of all sample tuples), if one does not know the total number of nodes in the graph (which is often the case in practice), generating estimations for COUNT and SUM often needs to use a significantly more expensive mark-and-recapture [52] based technique (e.g., [44]). However, in this method $\Omega(\sqrt{n})$ samples are needed to produce just one collision over an n-node graph - an extremely high query cost even for a perfectly built subgraph containing only users satisfying the

selection condition. For example, to estimate the COUNT of all users who tweeted `privacy` in 2013 (about 894,000), this means at least thousands of samples must be collected, incurring a very high query cost. To address this deficiency, the second design issue is how to efficiently traverse the graph to estimate AVG, COUNT and SUM aggregates. We shall develop a novel sampling algorithm to achieve these objectives in Section4.5. Table 4.1 shows which subgraph generation (GRAPH-BUILDER) and graph sampling (GRAPH-WALKER) components are employed by the two key proposed algorithms of this paper.

Prototype Design for Twitter Experiments: Before presenting out detailed design of MICROBLOG-ANALYZER in Section4.4 and Section 4.5, we would like to briefly discuss how we prototyped over Twitter, the preeminent micro-blogging platform. Note that while we focus the rest of the paper on this Twitter prototype, the adaption to other micro-blogging platforms is straightforward - e.g., we present experiments in Section 4.6 over Google+ and Tumblr.

Twitter's REST API [53] naturally fits into the data access model detailed in Section4.2. The search API retrieves tweets matching the given keywords which were posted during the past week [38]. The user timeline API provides access to a user's historic tweets (up to the last 3200). Since Twitter allows asymmetric relationship between users, we have to use two APIs to retrieve all the users who follow user u and all users who are followed by u, in order to collect all user connections as defined in the undirected social graph. Each API call returns up to 5000 connections while the vast majority (upwards of 95% [47]) of users have fewer than 100.

We now briefly describe how we collected the ground truth for evaluating our prototype's effectiveness on estimating aggregates such as "COUNT of all users who tweeted about `privacy` in 2013". We used the streaming API to collect all public

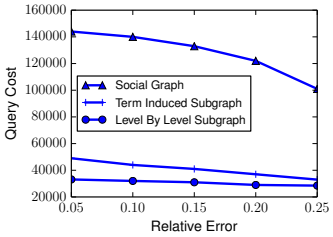


Figure 4.2. Microblog Analyzer: AVG(followers): Users who tweeted privacy.

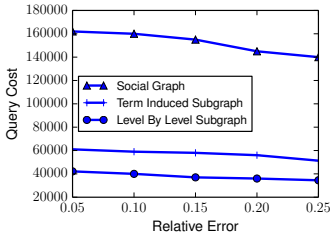


Figure 4.3. Microblog Analyzer: COUNT: Users who tweeted privacy.

tweets mentioning a diverse set of keywords (such as cities, celebrities, organizations, etc.) between Jan 1, 2013 to Oct 31, 2013. Twitter ensures that the stream returns all relevant tweets as long as their frequency is less than about 1% of the entire Twitter Firehose (total volume) [54]. Our specified keywords were selective and did not receive any rate limit exception, which means that this is an accurate ground truth to evaluate aggregate estimation algorithms.

Given a keyword w, we use Twitter's search API to retrieve the set of "seed" users who have used the keyword recently in their tweets. Once the list of seed user ids are obtained, the SUBGRAPH-BUILDER uses the timeline API to get their historic tweets. Specifically, for a given seed user u, it uses the Twitter's connection API to get the list of other users who were followed by u. This allows us to go back in time towards the time frame of interest (2013 in this case). Among the users who were followed by u, SUBGRAPH-BUILDER identifies a subset of them (thereby constructing a virtual subgraph). Some of the users are then pursued by TA-WALKER to perform the level-by-level walk over relevant user timelines.

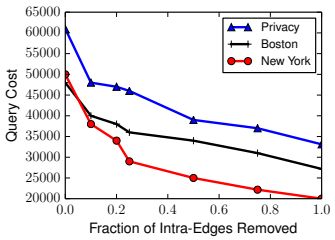


Figure 4.4. Microblog Analyzer: Impact of removing intra-edges on Query Cost.

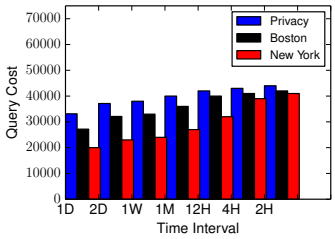


Figure 4.5. Microblog Analyzer: Impact of T on query cost.

4.4 Level-by-Level Subgraph

Recall that GRAPH-BUILDER aims to construct a subgraph (of the social graph defined in Section 4.3) with two properties: (1) a high recall of (timelines of) users who satisfy the selection condition of the aggregate query to be estimated, and (2) a topology that enables efficient sampling of such users. In this section, we start by describing a baseline method that achieves (1) but fails (2). The deficiencies of this baseline motivate us to propose a novel *level-by-level* subgraph to satisfy both. At the end of this section, we present Algorithm MA-SRW which enables aggregate estimation by performing simple random walks over the level-by-level graph.

Running example: Throughout this section and the next, we consider as running example the estimation of the following aggregate query over our Twitter prototype: AVG(number of followers) of users who tweeted the keyword `privacy` in 2013.

4.4.1 Baseline Subgraphs and Their Deficiencies

We start with discussing *term-induced subgraph*, a straightforward subgraph construction which serves as a baseline for our study. Simply put, unlike the original graph which includes all user timelines, the term-induced graph consists of only users who satisfy the *keyword selection condition* of the aggregate query. In the running

example, this leads to a subgraph consisting of all users who have tweeted `privacy` before. From a practical standpoint, this means that, during the random walk process, we always start with a user who has `privacy` in his/her timeline and only transit to users who satisfy the same criteria.

The rationale for this baseline approach is simple: Since nodes in the term-induced subgraph form a superset of those covered by the aggregate, the subgraph has a high recall as long as it remains connected or has a large connected component. On the other hand, the sampling efficiency is likely to be improved because of the reduced graph size. The design of the subgraph balances between the two objectives by filtering nodes only with keyword predicates (defined in Section 4.2) - which, as shown below, vastly reduces the subgraph size while keeping it connected - but not other conditions in the aggregate query - e.g., a time-interval condition which, when overly short, can result in a low recall.

Our experiments on Twitter confirmed the validity for the high-recall assumption - for all keywords and hashtags we tested (from popular ones such as `Fiscalcliff, New York, Superbowl` to more obscure ones such as `Tunisia`, `Simvastatin`), the largest connected component of the subgraph contains almost all (on average 94% - see Table 4.2 for details) nodes in the subgraph - demonstrating the high-recall of a term-induced subgraph. Intuitively, this is because of the strong correlation between social relationships and co-mentioning of keywords - i.e., not only are terms/hashtags likely propagated between followers and followees, but users who have similar interests tend to be connected *and* use the same keywords - leading to the high recall.

For sampling efficiency, our findings were mixed. While the query cost is indeed much lower than the original social graph, it is still very expensive. For the running example (average number of followers for users who tweeted `privacy`), this subgraph required close to 49,000 queries to obtain an estimate with less than 5% relative error.

Table 4.2. Microblog Analyzer: Statistics for Subgraphs

Keyword	**Recall**	**Avg#common neighbors**	**% of intra & cross-level**
`FiscalCliff`	97%	16, 2	27%, 1%
`New York`	91%	49, 3	32%, 2%
`Super Bowl`	93%	34, 1	29%, 2%
`Obamacare`	96%	21, 5	22%, 1%
`Tunisia`	86%	11, 4	28%, 1%
`Simvastatin`	81%	19, 2	24%, 2%
`Oprah Winfrey`	91%	22, 4	29%, 3%

While this value is significantly less than than the 144,000 queries required for the original graph, it is still high considering Twitter's rate limit. Figures 4.2 and 4.3 how the term-induced subgraph performs on estimating AVG(number of followers) and COUNT for users who tweeted `privacy`, respectively.

To understand why the efficiency problem remains with the term-induced subgraph, we note that even though users who tweeted `privacy` only represent a small percentage of all Twitter users, the number of *edges* connecting them in the term induced graph is still very large (e.g., close to 1 million edges connecting approximately 142 thousand nodes for the running example). With such a large and dense graph, the efficiency of sampling critically depends on whether the graph topology is carefully designed to enable efficient random walks.

Unfortunately, we found a special topological property of the term induced subgraph that is indeed very "unfriendly" for efficient sampling: Note that, exactly because of the same reason why the term-induced graph likely has a high recall, keywords are often propagated among users that form tightly connected communities (e.g., measured according to graph modularity [50]). This actually requires a random walk to have a long burn-in period because it is likely "trapped" inside a tightly

connected community before having a sufficient probability to propagate to other parts of the graph. Our experiments on Twitter confirmed this finding. The burn-in period (with Geweke threshold [55] $Z <= 0.1$) for the entire Twitter graph and the term induced subgraph (for `privacy`) were approximately 700 and 610 respectively. Similar behavior was observed for other keywords also (see Figure 4.4 for details).

One can see from the above discussion that the straightforward design of a term-induced subgraph cannot adequately address the sampling-efficiency problem of the original social graph, mainly because of the long burn-in dictated by traversing between tightly connected communities. In the next subsection, we describe our proposed methods for constructing a "sampling-friendlier" subgraph topology - specifically, by exploiting *time dimension* of the term-induced subgraph - i.e., the time order with which users posted a specified term like `privacy`.

4.4.2 Level-by-Level Subgraph

4.4.2.1 Key Idea and Rationale

To develop our idea of a level-by-level subgraph, we start with introducing a taxonomy of edges in the term-induced subgraph and discuss how each type of edges affect the efficiency of random walks. Consider a simple organization of all nodes (users) into multiple *levels* according to the time when a user first qualified for the keyword predicate (i.e., tweeted `privacy` in the running example). Consider an arbitrary time interval, say 1 day. We partition all users in the term-induced subgraph into multiple segments according to the interval (e.g., users published `privacy` between Jan 1, 13 and Oct 31, 13 will be partitioned into 303 segments).

If we draw each segment as a "virtual level" as in Figure 4.6, and place these levels from top to bottom in chronological order, then we can classify all edges in the

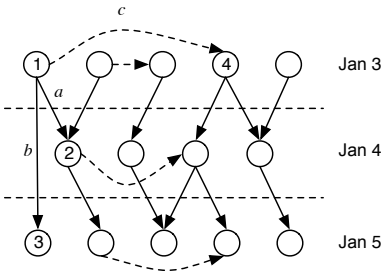


Figure 4.6. Microblog Analyzer: Level-by-Level View of term-induced Subgraph.

subgraph into three categories: *(a) Adjacent-level edges* connect two users in adjacent levels - e.g., Edge a in Figure 4.6 connects User 1 who first tweeted `privacy` on Jan 3 and User 2 who did so on Jan 4. *(b) Cross-level Edges* connect two users in unequal and non-adjacent levels - e.g., Edge b in Figure 4.6. *(c) Intra-level Edges* connect two users in the same level - e.g., Edge c in Figure 4.6.

The reason why we introduce such a ternary classification is because, interestingly, these three types of edges serve different roles in facilitating or deterring the random walk process. Specifically, we found that, for a "reasonable" time interval (>1 hour), (more) intra-level edges are *detrimental* to the efficiency of random walks, while (more) adjacent-level edges are *beneficial* to it. Cross-level edges, on the other hand, contribute to more efficient random walks but are relatively rare in practice (e.g., less than 1% for `privacy`. See Table 4.2 for other keywords).

While we shall verify this finding both theoretically and experimentally in Section 4.4.2.2, we would like to start here with an intuitive explanation for the varying effects different types of edges have on sampling efficiency. Intuitively, intra-level edges usually exist between users in a tightly connected component (as described in Section 4.4.1), while adjacent- and cross-level edges are most often not. This has been observed before - e.g., it was found in [56] that 92% retweets produced by followers of a user occur within 1-hour of the original tweet, demonstrating that most

followers "respond" within very short time, leading to intra-level edges between users in a tightly connected component. Our experiments confirmed this observation - e.g., Table 4.2 (column 2) contrasts, for various keywords, the average number of common neighbors shared by two users connected by an intra-level edge and those who are not. We can observe that, on average, one in four edges in the term induced subgraph is an intra-level edge. Further, the users connected by an intra level edges have significantly more common neighbors.

One can see from this explanation that, to "burn-in" to a stationary distribution, a random walk needs to cross adjacent- and/or cross-level edges and cannot get "stuck" inside a small group of users tightly connected by intra-level edges. Combine this with the fact that a substantial percentage of edges in a real-world term-induced graph are intra-level ones (e.g., even for a short interval of 1 hour, more than 28% of edges for keyword `privacy` are intra-level ones), a key idea for our subgraph design subgraph is to *remove all intra-level edges* from the term-induced graph. We refer to this subgraph as the *level-by-level subgraph* because it only contains edges between different levels. From a practical standpoint, this means that the random walk needs to follow a simple rule: transit from a user to its neighbor if and only if they did not first tweet `privacy` in the same day.

One can see that, to properly design a level-by-level subgraph, one needs to address two key issues. One is, of course, to verify that removing intra-level edges indeed improves the efficiency of random walks. We shall discuss this verification in Section 4.4.2.2. The other has to deal with the *time interval* used in defining intra-level edges. Note that an intra-level edge could be classified as adjacent- or even cross-level edge with a different time interval. We shall develop the proper setup of time interval for an aggregate in Section 4.4.2.3.

4.4.2.2 Effect of Intra-Level Edges

We now analyze the effect of intra-level edges on the efficiency of random walks in two steps. First, we present theoretical analysis on a simple example of level-by-level graph to illustrate how the removal of intra-level edges makes the graph more "well-knit" and more efficient for random walks. Then, we present experimental findings from our Twitter prototype to demonstrate the efficiency improvements achieved by removing intra-level edges.

For theoretical analysis, we consider the change of *graph conductance* [46] after the removal of intra-level edges. The *conductance* $\varphi(G)$ of a graph G measures how "well-knit" G is - i.e., how fast a random walk can converge to its stationary distribution. Specifically, we have

$$\varphi(G) = \min_{S \subseteq V} \frac{\sum_{v_i \in S, v_j \in \overline{S}} a_{ij}}{\min\left(a(S), a(\overline{S})\right)} \tag{4.1}$$

where V is the set of vertices in G, S and $\overline{S} = V \backslash S$ form a partition of V into two disjoint subsets, $a_{ij} = 1$ if there is an edge connecting v_i and v_j in G and 0 otherwise, and $a(S) = \sum_{v_i \in S} \sum_{v_j \in V} a_{ij}$. In general, a simple random walk burns-in faster on graphs with higher conductance [57].

Given the complexity of analyzing the conductance of an arbitrary graph, for the purpose of this paper, we consider a simple example of a level-by-level subgraph G as follows. Let there be n nodes in the graph which are distributed evenly across h levels (so each level contains n/h nodes). The adjacent-level edges in the graph are constructed such that each node at Level i ($i \in [1, h-1]$) is connected with d nodes chosen uniformly at random from those at Level $i+1$. The intra-level edges, on the other hand, connect each node at Level i with d' other nodes chosen uniformly at random from Level i. While this simple model does not match real-world

graph topologies, it nevertheless gives us an indication of how intra-level edges affect conductance, as demonstrated in the following theorem.

Theorem 5. *The conductance for G is*

$$\varphi(G) = \begin{cases} \frac{h}{(k+d)(h-1)n} & \text{if } d \leq \frac{n}{2h} \ , \ k \leq \frac{n}{2h} \\ \min\left(\frac{2kh-n}{kh+dn}, \frac{2d}{2d(h-1)+hk}\right) & \\ & \text{if } d \leq \frac{n}{2h} \ , \ \frac{n}{2h} < k < \frac{n}{h} \\ \min\left(\frac{2dh-n}{kh+dn}, \frac{2d}{2d(h-1)+hk}\right) & \\ & \text{if } \frac{n}{2h} < d < \frac{n}{h} \ , \ k \leq \frac{n}{2h} \\ \min\left((k-\frac{n}{2h})\frac{2dh-n}{kh+dn}, \frac{2d}{2d(h-1)+hk}\right) & \\ & \text{if } \frac{n}{2h} < d < \frac{n}{h} \ , \ \frac{n}{2h} < k < \frac{n}{h} \end{cases} \tag{4.2}$$

After removing all intra-level edges, the conductance of G' is

$$\varphi(G') = \begin{cases} \frac{h}{nd(h-1)} & \text{if } d \leq \frac{n}{2h} \\ \min\left(\frac{2hd-n}{nd}, \frac{1}{h-1}\right) & \text{if } \frac{n}{2h} < d < \frac{n}{h} \end{cases} \tag{4.3}$$

Proof. We give a proof sketch due to space limitations by showing *adding* intra level edges to a level by level graph actually decreases conductance. For simplicity, we consider a level-by-level graph (G') with h levels where each level has exactly n/h nodes. Each node is connected with $d (d \ll n/h)$ randomly chosen nodes in adjacent levels. In order to compute the conductance of this graph, we have to identify the cut that has the lowest conductance. There are two possible cuts - horizontal (where the cut disconnected two adjacent levels) or vertical (where the cut disconnects the graph into subgraphs, each with h levels).

After some algebraic manipulations, we can notice that the conductance of the horizontal cut is $\varphi(G')_{S_h} = \frac{1}{h-1}$. Similarly, the conductance of vertical cut is:

$$\varphi(G')_{S_v} = \begin{cases} \frac{h}{nd(h-1)} & \text{if } d \leq \frac{n}{2h} \\ min\left(\frac{2hd-n}{nd}, \frac{1}{h-1}\right) & \text{if } \frac{n}{2h} < d < \frac{n}{h}. \end{cases} \tag{4.4}$$

The conductance of the graph is $min(\varphi(G')_{S_h}, \varphi(G')_{S_v})$. In order to analyze the impact of intra level edges, we assume a simple model where each node has k randomly chosen intra-layer edges. We can see that the horizontal cut for this graph: $\phi(G)_{S_h} = 1/(h - 1 + hk/(2d))$. i.e. the horizontal cut with intra level edges decrease the conductance. There are four possible cases for vertical cut depending on the value of d and h. The second argument for min in Equation 4.2 provides the value for $\phi(G)_{S_v}$. Comparing the equations, we can notice that the additional factor of k (introduced due to intra level edges) actually reduces conductance. □

One can see from the theorem, specifically the comparison between (4.2) and (4.3) that the removal of intra-level edges significantly increases the graph conductance and thereby make the random walk process more efficient. Our experiments on the Twitter prototype verified this finding. Figure 4.4 shows, for various keywords , how the removal of 10% to 100% randomly chosen intra-level edges affect the query cost of simple random walks to achieve a relative error of $\leq 5\%$ on estimating the average number of followers for all users who tweeted the keyword in 2013. One can observe from the figure that as the query cost decreases dramatically when intra level edges are removed. Even removal of subset of such edges is actually helpful. We observed that this modification, on average, reduces the query cost for most keywords by at least 20%.

Without intra-level edges: There are two possible cuts in this lattice structure (level-by-level subgraph) that may lead to a balanced cut: a horizontal and a

vertical cut. When having a horizontal cut (S_h) in the middle of the lattice, the conductance of the lattice would be:

$$\begin{aligned}\varphi(G)_{S_h} &= \frac{d\frac{n}{h}}{2\frac{h-2}{2}d\frac{n}{h} + d\frac{n}{h}} \\ &= \frac{1}{h-1}\end{aligned} \tag{4.5}$$

For a vertical cut (S_v), there are two cases based on the degree of the nodes.

- $d \leq \frac{n}{2h}$:

$$\begin{aligned}\varphi(G)_{S_v} &= \frac{1}{2d\frac{n}{2h} + 2d(h-2)\frac{n}{2h}} \\ &= \frac{1}{d(h-1)\frac{n}{h}}\end{aligned} \tag{4.6}$$

It is clear that $\frac{1}{d(h-1)\frac{n}{h}} < \frac{1}{h-1}$ and vertical cut results in lower conductance than the horizontal cut.

- $\frac{n}{2h} < d < \frac{n}{h}$:

$$\begin{aligned}\varphi(G)_{S_v} &= \frac{2(h-1)(d-\frac{n}{2h})}{d(h-1)\frac{n}{h}} \\ &= \frac{2hd-n}{nd}\end{aligned} \tag{4.7}$$

Thus

$$\varphi(G) = \begin{cases} \frac{h}{nd(h-1)} & \text{if } d \leq \frac{n}{2h} \\ min\left(\frac{2hd-n}{nd}, \frac{1}{h-1}\right) & \text{if } \frac{n}{2h} < d < \frac{n}{h}. \end{cases} \tag{4.8}$$

Note that

$$\frac{n}{2h} < d < \frac{n}{h} \Rightarrow 0 < \frac{2hd-n}{nd} \leq \frac{h}{n} \tag{4.9}$$

- If $1 < h < \frac{(1+\sqrt{1+4n})}{2}$ or $0 < n < h^2 - h$:

$$\begin{aligned}&\Rightarrow 0 < \frac{2hd-n}{nd} \leq \frac{h}{n} < \frac{1}{h-1} \\ &\Rightarrow min\left(\frac{2hd-n}{nd}, \frac{1}{h-1}\right) = \frac{2hd-n}{nd}\end{aligned} \tag{4.10}$$

In other words, vertical cut is better.

- But if $h > \frac{(1+\sqrt{1+4n})}{2}$ or $n > h^2 - h$ horizontal cut is better and:

$$min\left(\frac{2hd-n}{nd}, \frac{1}{h-1}\right) = \frac{1}{h-1} \tag{4.11}$$

With intra-level edges: Now let's assume there are k intra-level edges for every node in the lattice structure, where $k >> d$. Then Equations 4.5 to 4.8 will change as fallows: By the horizontal cut the conductance would be:

$$\begin{aligned} \varphi(G)_{S_h} &= \frac{d\frac{n}{h}}{\frac{h-2}{2}(2d+k)\frac{n}{h} + (d+k)\frac{n}{h}} \\ &= \frac{1}{h-1+\frac{hk}{2d}} \end{aligned} \tag{4.12}$$

and it is clear that since the $\frac{hk}{2d} > 0$, the conductance of the horizontal cut in lattice with intra-edges is decreasing.

In vertical cut (S_v) there are four cases based on the degree of the nodes.

- $d \leq \frac{n}{2h}$, $k \leq \frac{n}{2h}$:

$$\begin{aligned} \varphi(G)_{S_v} &= \frac{1}{2(d+k)\frac{n}{2h} + 2(d+k)(h-2)\frac{n}{2h}} \\ &= \frac{1}{(k+d)(h-1)\frac{n}{h}} \end{aligned} \tag{4.13}$$

  And it is clear that $\frac{1}{(k+d)(h-1)\frac{n}{h}} < \frac{1}{h-1+\frac{hk}{2d}}$ and vertical cut results in lower conductance than the horizontal cut.

- $d \leq \frac{n}{2h}$, $\frac{n}{2h} < k < \frac{n}{h}$:

$$\begin{aligned} \varphi(G)_{S_v} &= \frac{2(h-1)(k-\frac{n}{2h})}{(k+d)(h-1)\frac{n}{h}} \\ &= \frac{2kh-n}{kh+dn} \end{aligned} \tag{4.14}$$

- $\frac{n}{2h} < d < \frac{n}{h}$, $k \leq \frac{n}{2h}$:

$$\begin{aligned} \varphi(G)_{S_v} &= \frac{2(h-1)(d-\frac{n}{2h})}{(k+d)(h-1)\frac{n}{h}} \\ &= \frac{2dh-n}{kh+dn} \end{aligned} \tag{4.15}$$

- $\frac{n}{2h} < d < \frac{n}{h}$, $\frac{n}{2h} < k < \frac{n}{h}$:

$$\begin{aligned} \varphi(G)_{S_v} &= \frac{2(h-1)(d-\frac{n}{2h})(k-\frac{n}{2h})}{(k+d)(h-1)\frac{n}{h}} \\ &= (k-\frac{n}{2h})\frac{2dh-n}{kh+dn} \end{aligned} \tag{4.16}$$

Thus the conductance of the lattice structure with intra-level edges woud be:

$$\varphi(G) = \begin{cases} \frac{1}{(k+d)(h-1)\frac{n}{h}} & \text{if } d \leq \frac{n}{2h} \text{ , } k \leq \frac{n}{2h} \\ min\left(\frac{2kh-n}{kh+dn}, \frac{1}{h-1+\frac{hk}{2d}}\right) & \text{if } d \leq \frac{n}{2h} \text{ , } \frac{n}{2h} < k < \frac{n}{h} \\ min\left(\frac{2dh-n}{kh+dn}, \frac{1}{h-1+\frac{hk}{2d}}\right) & \text{if } \frac{n}{2h} < d < \frac{n}{h} \text{ , } k \leq \frac{n}{2h} \\ min\left((k-\frac{n}{2h})\frac{2dh-n}{kh+dn}, \frac{1}{h-1+\frac{hk}{2d}}\right) & \text{if } \frac{n}{2h} < d < \frac{n}{h} \text{ , } \frac{n}{2h} < k < \frac{n}{h}. \end{cases} \tag{4.17}$$

4.4.2.3 Time Interval in Level-by-Level Subgraph

We now address the second issue - how to properly set the time interval T which directly affects edge classification. Once again, we start with theoretical analysis on optimal T based on the simple example of level-by-level graph described in Section 4.4.2.2. Then, we verify the analysis with experimental findings over Twitter.

Theoretical Analysis: Note that the setting of T affects two parameters in this simple model: (1) the number of levels h - the longer T is, the smaller h, and (2) d, the number of (randomly chosen) Level $i+1$ nodes a Level i node is connected with. Here the relationship between T and d is not as clear: While a longer T will in general lead to more nodes on Level $i+1$, it *might* actually reduce d if most followers of the Level i node already responded within the time interval corresponding to Level i. The following corollary to Theorem 5 illustrates the relationship between h and d in order to maximize conductance of the level-by-level subgraph.

Corollary 3. *To maximize the conductance of G', there is*

$$d = \frac{(2h-1)(2h-2)}{h(2h-9)} \tag{4.18}$$

The conductance of such a lattice structure is

$$\varphi(S) = \min\left(\frac{d \cdot n/h}{\frac{n}{2} \cdot \frac{h-1}{h} \cdot 2d + \frac{n}{2} \cdot \frac{1}{h} \cdot d}, \frac{\frac{n}{2h} \cdot \frac{d-1}{d} \cdot \frac{d}{2} \cdot (h-1)}{\frac{n}{2} \cdot \frac{h-2}{h} \cdot 2d + \frac{n}{2} \cdot \frac{2}{h} \cdot d}\right) \tag{4.19}$$

$$= \min\left(\frac{2}{2h-1}, \frac{d-1}{4d}\right). \tag{4.20}$$

Roughly speaking, the larger h is, the smaller d will be. Thus, the optimal setting should satisfy

$$d = \frac{2h-1}{2h-9}. \tag{4.21}$$

Note that the average degree of a node, δ, satisfies

$$\delta = \frac{2d \cdot (h-1)}{h} \tag{4.22}$$

Thus, the average degree δ and the number of levels h should satisfy

$$\delta = \frac{(2h-1)(2h-2)}{h(2h-9)} \tag{4.23}$$

The proof follows directly from Theorem 5. Intuitively, this means that instead of setting the T to a fixed value, we should adjust it according to the propagation pattern of the query term or hashtag. Specifically, the average number of followers who "pick up" the hashtag after the current time interval should be close to its optimal value d as shown in (4.18). For example, if the average degree is around $d = 14$, then there should be around $h \approx 5$ levels in the lattice structure. Of course, the real-world scenario is more complex. For example, the average number of "pick ups" tends to decline over time - indicating that the time interval should be dynamically changed throughout the duration of propagation [56, 48].

Another interesting observation from the corollary is that the optimal value of d becomes very close to 2 (i.e., its limit when $h \to \infty$) when h is reasonably large. For example, we have $d = 2.13$ and 2.06 when $h = 50$ and 100, respectively. This

means that when the keyword of interest has been propagated for a long time (e.g., `privacy`), we can set T according to a simple rule of $d = 2$.

Practical Design: Recall that GRAPH-BUILDER constructs the best subgraph *on-the-fly* during aggregate estimation. First, we discuss a simpler problem where we are given a set of candidate values for T and aim is to identify which is best for estimating an aggregate. Constructing the term-induced subgraph for each value and comparing them is not ideal as it would require a prohibitive query cost. Instead, we perform a *pilot* random walk using each of the time intervals. Each of the pilot random walks uses a smaller budget (e.g., 50 samples) and terminates quickly. Using the partial topology revealed by each walk, we compute h and d and estimate the value of conductance using (4.3). The time interval with the highest conductance is selected and used for the rest of the process.

We evaluated the effectiveness of this over Twitter. Specifically, we identified a set of diverse time intervals varying from 1-hour to 1-month. For each time interval, we estimated its efficacy in sampling as against the theoretical value of the conductance. In other words, we ordered the time intervals in the horizontal x axis based on their conductance. We then performed random walk for each of these time intervals and compared the query cost to achieve a relative error of less than 5%. Figure 4.5 shows the results for three keywords. One can see that the orders based on theoretical conductance and experimental performance are consistent.

Algorithm MA-SRW: Recall from Section 4.3 the two key components of MICROBLOG-ANALYZER: GRAPH-BUILDER and GRAPH-WALKER. In this section, we developed a level-by-level subgraph for GRAPH-BUILDER. We now combine it with simple random walk in GRAPH-BUILDER to produce Algorithm MICROBLOG-ANALYZER-Simple Random Walker (MA-SRW). The samples obtained are then

used for aggregate estimation in the same way as simple random walk [46]. Algorithm 10 depicts the pseudocode for MA-SRW.

Algorithm 10 MA-SRW

1: Retrieve seed users
2: **while** Remaining query budget > 0 **do**
3: Retrieve samples using simple random walk. At each transition, only select from edges that belong to the level-by-level graph according to time interval T.
4: **end while**
5: Perform aggregate estimation as in simple random walk [46].

4.5 Topology-Aware Random Walks

To understand the key ideas of our Topology-Aware random walk algorithm, we start by briefly discussing the deficiencies of existing techniques, specifically the direct application of simple random walk or Metropolis-Hastings random walk to the level-by-level subgraph we constructed in Section 4.4. Then, we develop the key ideas for a novel topology-aware, level-by-level, random walk and present our MA-TARW algorithm.

4.5.1 Deficiencies of Traditional Random Walks

As mentioned in the introduction, the existing techniques have two main problems: (1) although they produce asymptotically unbiased (according to their respective stationary distributions) samples after a burn-in period, the number of transitions required for the burn-in is usually high [49]; and (2) while they can be combined with

mark-and-recapture [52] to estimate SUM and COUNT queries based on the samples, the query cost often rises to a prohibitively high level for practical purposes.

We note here that the fundamental reason underlying these problems is the *inability* of traditional random walk techniques to estimate the probability for a node u to be chosen as a sample. Note that while simple random walk is known to have a stationary distribution that assigns probability proportional to a node's degree $d(u)$, it is still impossible to compute the exact probability for a node to be accessed (i.e., $d(u)/(2|E|)$ where E is the set of all edges) unless one knows the total number of edges in the graph. Similarly, to know the exact probability for a node to be accessed by Metropolis-Hastings random walk (i.e., $1/|V|$ where V is the set of all vertices), one has to know the total number of nodes in the graph. Clearly, neither piece of knowledge is available *a priori* in our case - and estimating them (e.g., by using mark-and-recapture) requires a very high query cost.

To understand the importance of knowing the exact probability for a node to be accessed, note that such knowledge indeed addresses both deficiencies outlined above. First, with knowledge of $p(u)$, the probability for a node to be taken as a sample, one can simply apply the Hansen-Hurwitz estimator [58] to generate an *unbiased* estimation for any SUM or COUNT query defined in Section 4.2 as $f(u)/p(u)$, where $f(u)$ is the result of applying the SUM or COUNT query over u itself[4]. This avoids the usage of mark-and-recapture and, as a result, significantly reduces the query cost required for answering SUM and COUNT queries[5].

Similarly, the efficiency problem - i.e., the long burn-in period required - is also (at least partially) caused by the lack of knowledge on the probability for a node to

[4]e.g., if the aggregate is the number of posts containing `privacy`, then $f(u)$ is the number of u's posts containing `privacy`.

[5]Note that AVG queries can be simply estimated as SUM/COUNT.

be sampled at a certain step of the random walk. Specifically, the lack of knowledge mandates a long burn-in period for the sampling probability to converge to its target stationary distribution. If one can compute $p(u)$ during each step of the random walk process, then an unbiased aggregate estimation can be generated as long as $p(u) > 0$ for all u in the graph[6] - potentially saving significant query cost for the sampling process.

Admittedly, if one has no knowledge of the global graph topology, it is impossible to compute or make any meaningful estimation[7] of $p(u)$ without incurring as high a query cost as mark-and-recapture [44, 52]. The reason is simple - without "recapturing" at least some nodes accessed before, it is impossible to determine the scale of the graph as, theoretically speaking, it is entirely possible that the access cost we have incurred so far is still smaller than the average pairwise distance between nodes in the graph (one can always construct such an extreme-case scenario), making it impossible to guarantee or even estimate the error of aggregate estimations.

Fortunately, the subgraph construction technique described in Section 4.4 affords us substantial knowledge of the graph topology - not the entire node/edge sets - but knowledge of the level-by-level structure all nodes and edges are organized by, and which level a node falls into. As we shall show in the following subsection, such knowledge gives us the ability to efficiently compute an unbiased estimation of $p(u)$, which in turns enables a significantly more efficient (topology-aware) sampling process than the traditional random walk techniques.

[6]Note that the requirement $p(u) > 0$ is here to ensure that the sampling process can reach all nodes covered by the aggregate.

[7]Here we use "meaningful" to refer to estimations with statistical guarantees on bias and/or variance.

4.5.2 Key Idea: Level-by-Level Random Walk

In this section, we first develop a novel level-by-level random walk process by leveraging knowledge of the subgraph topology we constructed in Section 4.4. We also explain why this process requires far fewer queries than traditional (simple or Metropolis-Hastings) random walks. Then, we discuss how to estimate $p(u)$ in a level-by-level random walk - which in turn enables accurate aggregate estimations.

Description of Level-by-Level Random Walk: To understand the level-by-level random walk process, we start by considering a simple example where a level-by-level subgraph constructed for a given keyword has h levels and only edges between nodes of adjacent levels. As shown in Figure 4.6, the top level consists of users who mentioned the keyword earliest, while users at the bottom one or few levels are guaranteed to be returned by Twitter's search API (which has a time limit of about 1 week [38]) - i.e., our random walk process starts from these bottom levels. Note that every edge in the graph is directed from top to bottom.

Our topology-aware, level-by-level, random walk follows a bottom-top-bottom flow on the subgraph - i.e., a *random walk instance* starts from the bottom level and moves up one level at a time, by following the inverse direction of edges, until reaching a node with no incoming edge. Then, it reverses traversal direction and starts following the original edge directions to transit down, again one level at a time, until it reaches a node with no outgoing edge - at which time (this instance of) the random walk terminates. At each transition during the random walk, a branch is chosen uniformly at random. Note that *all nodes* we pass through during a random walk will be used to generate one aggregate estimation - and one can execute multiple instances of the random walk and average out the results to produce more accurate estimations - the details of these issues shall be described in the next subsection.

Before discussing the probability for each node to be chosen by such a level-by-level random walk, we first note that the query cost required by each instance of the random walk is much smaller than that for traditional topology-oblivious random walks. Specifically, our walk instance requires at most $2(h-1)$ transitions, orders of magnitude fewer than simple and Metropolis-Hastings random walks, according to the results in Section 4.6.2.

There is a simple reason behind this advantage: by leveraging knowledge of the level-by-level topology, our random walk process is capable of transiting between different "clusters" of nodes much faster than traditional topology-oblivious random walks. Specifically, for a $2(h-1)$-step level-by-level random walk instance over the above-described h-level graph, each of the first (or last) $h-1$ steps is guaranteed to draw from mutually exclusive subsets of nodes. This makes the random walk process reach (with a positive probability) all nodes in the graph much faster than traditional random walks which, despite improved subgraph designs, still have a fairly high probability to return to their origin point after a small number of transition steps[59].

Unbiased Estimation of $p(u)$: We now consider the estimation of $p(u)$ - the probability for a level-by-level random walk instance to reach a node u in the subgraph. To do so, we first define some notation. We use $\acute{p}(u)$ and $\grave{p}(u)$ to represent the probability for a random walk to reach u during the bottom-top and top-bottom phases, respectively. Also, we use $\nabla(u)$ and $\Delta(u)$ to denote the set of neighbors of u on the levels above and below u, respectively. The key observation for estimating $\acute{p}(u)$ and $\grave{p}(u)$ is

$$\acute{p}(u) = \sum_{v \in \Delta(u)} \frac{\acute{p}(v)}{|\nabla(v)|}, \quad \grave{p}(u) = \sum_{v \in \nabla(u)} \frac{\grave{p}(v)}{|\Delta(v)|}, \tag{4.24}$$

which holds for all but two exceptions: (1) for a node u with no incoming edges - i.e., when $\nabla(u) = \varnothing$ - we have $\grave{p}(u) = \acute{p}(u)$, and (2) for a node u with no outgoing edges - i.e., when $\Delta(u) = \varnothing$ - it is either $\acute{p}(u) = 1/s$ - where s is the number of seed nodes[8] the random walk might start from - if u is one of the seed nodes, or $\acute{p}(u) = 0$ otherwise.

Equation 4.24 illustrates a simple recursive process for producing an unbiased estimation of $p(u)$: Note that if we choose a node v uniformly at random from $\Delta(u)$, then

$$\omega(\acute{p}(u)) = \frac{|\Delta(u)| \cdot \acute{p}(v)}{|\nabla(v)|} \tag{4.25}$$

is an unbiased estimation for $\acute{p}(u)$ (same[9] applies to $\grave{p}(u)$). In addition, if we replace $\acute{p}(v)$ in (4.25) with an unbiased estimation of it, say $\omega(\acute{p}(v))$, then $|\Delta(U)| \cdot \omega(\acute{p}(v))/|\nabla(v)|$ remains an unbiased estimation of $\acute{p}(u)$ as long as the random selection of v from $\nabla(u)$ is independent of the estimation of $\omega(\acute{p}(v))$.

As such, the recursive process works as follows: After each instance of the level-by-level random walk terminates, we take $\acute{U}$ and $\grave{U}$, the sets of nodes the instance passes through during the bottom-top and top-bottom phases, respectively. Then, for each node $u \in \grave{U}$, we start a bottom-top, level-by-level random walk starting from u, this time for the sole purpose of recursively estimating $\grave{p}(u)$. On the other hand, for each node $u \in \acute{U}$, we start a top-bottom level-by-level random walk to estimate $\acute{p}(u)$ in a recursive fashion. Algorithm 11 depicts the pseudocode for estimating $\acute{p}(u)$ (the algorithm for $\grave{p}(u)$ is similar). One can see that this process can produce unbiased

[8]Recall from Section 4.3 that seed nodes consist of users returned by the limited search interface - e.g., for Twitter, those who tweeted the keyword within the last week and thus returned by the Search API.

[9]i.e., if we choose a node v uniformly at random from $\nabla(u)$, then $|\nabla(u)| \cdot \grave{p}(v)/|\Delta(v)|$ is an unbiased estimation for $\grave{p}(u)$.

Algorithm 11 ESTIMATE-$\acute{p}$

Input: u

if $\Delta(u) == \varnothing$ **then**

  $//u$ is a bottom level node

  $\acute{p}(u) = \begin{cases} 1/s & \text{If } u \text{ is one of the } s \text{ seeds} \\ 0 & \text{Otherwise} \end{cases}$

else if $\nabla(u) == \varnothing$ **then**

  $//u$ is a top level node

  $\acute{p}(u) = \grave{p}(u)$

else

  Pick a node v randomly from $\Delta(u)$

  $\acute{p}(v) =$ ESTIMATE-$\acute{p}$ (v)

  $\acute{p}(u) = \frac{|\Delta(U)| \cdot \acute{p}(v)}{|\nabla(v)|}$

end if

estimations of $\grave{p}(u)$ or $\acute{p}(u)$ for every node that the random walk instance passes through - i.e., every node that will be used in the aggregate estimation process, as explained in the next subsection.

Since the above discussions have established the unbiasedness of $f(u)/\grave{p}(u)$ on SUM and COUNT estimations as well as the unbiasedness of $\omega(\grave{p}(u))$ on estimating $\grave{p}(u)$, we now consider the other important factor affecting the error of aggregate estimation: *variance*. Specifically, the following theorem illustrates the estimation variance produced by topology aware random walk for SUM aggregates. Note that since COUNT can be considered as a special case of SUM (when $f(u) = 1$), estimation errors of COUNT and AVG (i.e., SUM/COUNT) aggregates can be derived accordingly.

Theorem 6. *For aggregate* Q_{A}*:* `SELECT SUM(`$f(u)$`) FROM` U `WHERE` *cond, after* r *random walk instances, topology aware random walk generates an estimation variance*

$$\sigma^2 = \left(\sum_{u \in cond} \frac{(V+1) \cdot f(u)^2}{r \cdot \grave{p}(u)} \right) - \frac{Q_{\mathrm{A}}^2}{r}, \text{ where} \tag{4.26}$$

$$V = \sum_{u \in cond} \sum_{\rho \in \mathcal{P}(u)} \grave{p}(u) \cdot p(\rho) \cdot \left(\frac{\grave{p}(u)}{\omega(\rho)} - 1 \right)^2 \tag{4.27}$$

when r *is sufficiently large, where* Q_{A} *is the real aggregate value,* $\mathcal{P}(u)$ *is the set of all bottom-top-bottom paths from* u *to one of the seed nodes,* $\omega(\rho)$ *is the estimation of* $\grave{p}(u)$ *produced by Algorithm 11 when path* ρ *is taken for estimating* $\grave{p}(u)$*, and* $p(\rho)$ *is the probability for* ρ *to be taken.*

Proof. For each node u passed through by a random walk instance, we can generate an estimation of Q_{A} from u as

$$\epsilon_u = \frac{f(u)}{\omega(\grave{p}(u))} = \frac{f(u)}{p(u)} \cdot \frac{\grave{p}(u)}{\omega(\grave{p}(u))}. \tag{4.28}$$

We denote $f(u)/\grave{p}(u)$ and $\grave{p}(u)/\omega(\grave{p}(u))$ as α_u and β_u, respectively. As we discussed above, $\omega(\grave{p}(u))$ is an unbiased estimation of $\grave{p}(u)$. Thus, $1/\omega(\grave{p}(u))$ is an asymptotically unbiased estimator for $1/\grave{p}(u)$ (when $r \to \infty$ leads to a more accurate estimation of $\omega(\grave{p}(u))$). Since the bottom-top-bottom random walks used for estimating $\grave{p}(u)$ are independent of those used for reaching u from the seed nodes, one can see that the covariance of α_u and β_u tend to 0 when $r \to \infty$.

As such, the variance of ϵ_u is

$$\sigma_u^2 = \mathrm{var}(\alpha_u) \cdot \mathrm{var}(\beta_u) + \mathrm{var}(\alpha_u) \cdot E(\beta_u)^2 + \mathrm{var}(\beta_u) \cdot E(\alpha_u)^2 \tag{4.29}$$

$$= \mathrm{var}(\alpha_u) \cdot (\mathrm{var}(\beta_u) + 1) + \mathrm{var}(\beta_u) \cdot Q_{\mathrm{A}}^2 \tag{4.30}$$

The derivation is because of the fact that $f(u)/\grave{p}(u)$ is an unbiased estimation for Q_{A}. According to Theorem 2 in [3], we have

$$\mathrm{var}(\alpha_u) = \left(\sum_{u \in cond} \frac{f(u)^2}{\grave{p}(u)} \right) - Q_{\mathrm{A}}^2. \tag{4.31}$$

The variance of β_u can be directly derived from the definition of variance, i.e.,

$$\mathrm{var}(\beta_u) = \sum_{u \in cond} \sum_{\rho \in \mathcal{P}(u)} \grave{p}(u) \cdot p(\rho) \cdot \left(\frac{\grave{p}(u)}{\omega(\rho)} - 1 \right)^2. \tag{4.32}$$

The combination of (4.30), (4.31) and (4.32) proves the theorem. □

Note that an intuitive explanation for V in the theorem is the variance of $\grave{p}(u)/\omega(\grave{p}(u))$, where $\omega(\grave{p}(u))$ is the estimation of $\grave{p}(u)$ produced by our algorithm, taken over the randomness of $\omega(\grave{p}(u))$. One can observe from the theorem that a key factor determining the estimation variance is the values of $\grave{p}(u)$ for nodes in the subgraph. To understand why, note from (4.26) that, given V, σ^2 is in general inversely proportional to $\grave{p}(u)$. Thus, if the subgraph happens to be highly skewed so as to have a node u with an extremely small $\grave{p}(u)$, then the estimation variance σ^2 (and thereby the aggregate estimation error) can be very large. Fortunately, as we shall show in Section 4.6, the variance is indeed fairly small in practice for the wide variety of keywords we tested.

Before concluding this subsection, we would like to briefly discuss the additional query cost introduced by the probability estimation process. One can see that, in order to estimate $\grave{p}(\cdot)$ or $\acute{p}(\cdot)$ for the (at most) $2h-1$ nodes the random walk passes through, this process requires at most $(2h-1) \cdot (h-1)$ additional transitions. While such $O(h^2)$ query cost surpasses that required by the level-by-level random walk itself, it is unlikely to cause any efficiency concern in practice because of the following two reasons.

First, as one can see from the results in Section 4.6.2, even a query cost of $O(h^2)$ is still an order of magnitude lower than topology-oblivious random walks, and second, the real-world query cost for estimating $\grave{p}(\cdot)$ or $\acute{p}(\cdot)$ is often lower than the worst-case scenario. To understand why, consider a common scenario where the level-by-level subgraph has one or a small number of roots at the top. Let there be one root v_{r}. Note that once we produce an estimation of $\grave{p}(v_{\mathrm{r}})$ (which is equal to $\acute{p}(v_{\mathrm{r}})$), we can reuse it for estimating $\grave{p}(\cdot)$ for *all* nodes in the top-bottom phase of *all* random walk instances - i.e., for these nodes, the probability estimation process only needs to walk bottom-up and not top-bottom anymore - saving about half of the query cost because of a single cache.

4.5.3 Algorithm MA-TARW

In this subsection, we put together the previous discussions of level-by-level subgraph, topology aware random walk and the unbiased estimation of selection probability $\grave{p}(u)$ to develop Algorithm MA-TARW, which can be used to estimate SUM, COUNT and AVG aggregates with or without selection conditions.

Algorithm 12 depicts the pseudocode for MA-TARW. First, it uses a small number of bootstrapping transitions to identify the best time interval T for the level-by-level subgraph (see Section 4.4.2.3 for details). It randomly picks a bottom level node (a user who has recently tweeted about the hashtag) and performs a bottom-top-bottom random walk instance $\mathcal{R}_i$ as described in previous subsection. For each node u in the walk $\mathcal{R}_i$, it computes the selection probability ($\acute{p}(u)$ or $\grave{p}(u)$). All nodes in $\mathcal{R}_i$ are used in computing a single estimate of the aggregate query. This random walk process is then repeated for multiple times - with the average estimate being outputted as the final aggregate estimation.

Algorithm 12 MA-TARW

1: Estimate best value of T using bootstrapping transitions

2: **while** Remaining query budget > 0 **do**

3: Perform a bottom-top-bottom random walk $\mathcal{R}_i$

4: $\acute{p}(u)$ = ESTIMATE-$\acute{p}$ (u) $\forall\ u \in \acute{U}$ of $\mathcal{R}_i$

5: $\grave{p}(u)$ = ESTIMATE-$\grave{p}$ (u) $\forall\ u \in \grave{U}$ of $\mathcal{R}_i$

6: // Remove nodes from $\acute{U}$, $\grave{U}$ that does not match input query

7: $\widetilde{f}(\mathcal{R}_i) = \frac{1}{|\mathcal{R}_i|}\left(\sum_{u\in\acute{U}} \frac{f(u)}{\acute{p}(u)} + \sum_{u\in\grave{U}} \frac{f(u)}{\grave{p}(u)}\right)$

8: **end while**

9: **Return** average of all previous estimates $\widetilde{f}(\mathcal{R}_i)$

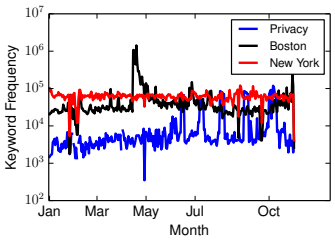


Figure 4.7. Microblog Analyzer: Frequencies of Chosen Keywords.

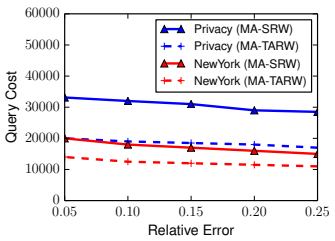


Figure 4.8. Microblog Analyzer: Twitter: AVG(followers).

4.6 Experimental Evaluation

In this section we evaluate the efficiency and accuracy of Algorithms MA-SRW and MA-TARW proposed in the paper, and compare them against a state-of-the-art baseline method.

4.6.1 Experimental Setup

Hardware and Platform: All our experiments were conducted on a computer with Intel Core(TM) i5 2.50 GHz CPU with 8 GB of RAM. The algorithms were implemented in Python 2.7.

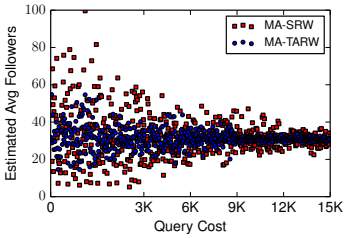


Figure 4.9. Microblog Analyzer: Twitter: Estimated AVG(followers).

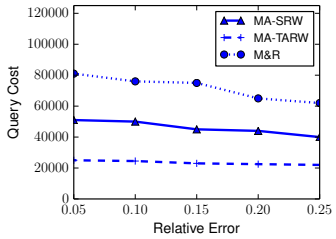


Figure 4.10. Microblog Analyzer: Twitter: Count(users).

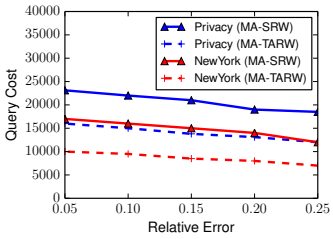


Figure 4.11. Microblog Analyzer: Twitter: AVG(Display Name).

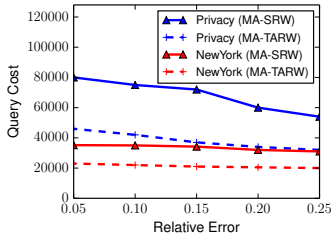


Figure 4.12. Microblog Analyzer: Google+: AVG(Display Name).

Datasets: We tested our algorithms on three real-world microblogging platforms - Twitter, Google+ and Tumblr. These were chosen due to their popularity and accessibility of their developer API. While the majority of our experiments were conducted over Twitter, we observed similar behavior on the other microblogs also. All our experiments were conducted by running it over the microblog in real-time.

Detailed discussion on how MICROBLOG-ANALYZER is instantiated for Twitter is found in Section 4.3.2. We now briefly describe how Google+ and Tumblr are instantiated. Google+ is an microblogging platform from Google that has more than 500 million users. It provides both API and web based interfaces. Google+ has an *Activity* API (equivalent to the Twitter search API) that allows us to search for posts that specify a particular keyword. It also has an API to retrieve user profile information, as allowed by the privacy setting of the user. However, some basic information such as display name are always available. Similar to Twitter, connections in

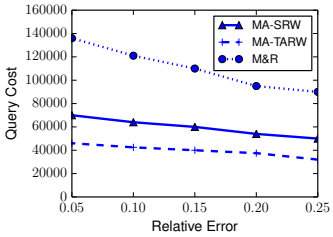


Figure 4.13. Microblog Analyzer: Google+: COUNT (male users who tweeted).

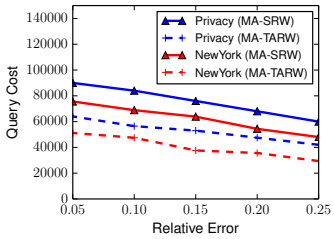


Figure 4.14. Microblog Analyzer: Tumblr: AVG(Likes).

Google+ are asymmetric. Connections are grouped into various groups, called *Circles*. Google+ has a *courtesy* rate limit of 10,000 queries per day and 5 per second. Due to the difficulty in retrieving connections (the API only provides the connections of an authenticated user), we define two users to be connected if they performed some activity together in last year, i.e., they liked, shared or commented the same post. Tumblr is another popular microblogging platform where users host multiple blogs and can follow blogs of other users. The posts in blogs correspond to tweets in Twitter which can then be liked or reblogged by other users. Tumblr has extensive API to retrieve various information about blogs. Requests are rate-limited to one every 10 seconds.

Aggregate Queries and Ground Truth: In our experiments, we focused on aggregate queries AVG, COUNT and SUM. We evaluated aggregate measures such as the number of followers, display name length, number of likes in the blog etc. For Twitter, we used the streaming API to collect all public tweets mentioning a diverse set of keywords (such as cities, celebrities, organizations etc) between Jan 1 - Nov 1, 2013. Since Twitter ensures that the stream returns all relevant tweets as long as their frequency is less than 1% of the Twitter Firehose and our keywords are not too frequent, this provides a reasonable ground truth over which aggregate estimation algorithms can be evaluated. Figure 4.7 shows the frequency of three keywords used

in the evaluation over time - `privacy` (a relatively low frequency term with occasional spikes), `New York` (a perpetually popular and high frequency keyword) and `Boston` (keyword that has medium frequency but had a singular spike on Apr 15, 2013 when the Marathon Bombing occurred). For Google+ and Tumblr, no convenient way to collect ground truth exists. To get a reasonable approximation, we instantiated multiple samplers that performed simultaneous random walks until they converged to their stationary distribution (with Geweke threshold $Z \leq 0.05$). The average estimate from all the walks serves as ground truth.

Performance Measures: Our aggregate estimation algorithms were evaluated according to two measures. Efficiency was measured as the number of API interface calls. Notice that multiple API calls could be required to obtain the result of a single query. For example, Twitter's followers API returns 5000 users per call and hence multiple calls are required to retrieve all followers of a celebrity. To measure accuracy, we use the relative error (see Section 4.2).

Algorithms Evaluated: We evaluated MA-SRW, MA-TARW and the state-of-art baseline M&R described next. Recall from Section 4.4 that MA-SRW outperformed SRW on the original or the term-induced social graph. Hence, to keep the presentation clear, we do not present any experiments on the original or the term-induced social graph. To the best our knowledge, we have not found any research that performs general aggregate estimation over microblogs. The closest is [44] that performs size (COUNT) estimation for (entire) social networks and does not directly support keyword-specific size estimation. We adapted [44] to only consider nodes that match the query and used it to measure the size of the term induced subgraph and refer this algorithm as M&R (for mark and recapture); we only include it for COUNT (as the algorithm was designed for).

4.6.2 Experimental Results

Efficacy of MA-TARW (Twitter): We conducted an extensive set of experiments to validate the efficacy of MA-TARW. Recall that experiments from Section 4.4 showed that performing random walk over level by level graph is more efficient than performing over original social graph or the term-induced subgraph. Hence, in this section, we do not compare MA-TARW over such graphs. Further, the level-by-level nature of the graph leveraged by MA-TARW is not available for the original social graph. We start with Twitter. Table 4.3 shows the average percentage improvement in Twitter query cost achieved by MA-TARW over MA-SRW and M&R for AVG(followers) and COUNT(users) queries (from Jan 1, 2013 to Oct 31, 2013) involving diverse keyword conditions to achieve a relative accuracy error of 5%. The results show that MA-TARW outperforms both competing algorithms and confirm our theoretical analysis.

Next, we study in more detail specific aggregate queries. We use MA-TARW to estimate the average number of followers of all users who tweeted `privacy`. Figure 4.8 shows that MA-TARW significantly outperforms MA-SRW. Figure 4.9 also validates this conclusion by showing that MA-TARW converges to the true estimate and has a lower variance in its estimate within few thousand queries.

We then perform a COUNT estimate of all users who tweeted `privacy`. Figure 4.10 shows that MA-TARW outperforms both MA-SRW and baseline M&R. Recall from Figure 4.3 that M&R requires lower query cost when evaluated on the level-by-level subgraph than on term induced subgraph; this is why we execute M&R on the level-by-level subgraph to better evaluate our topology-aware navigation algorithm. We next consider an aggregate query to estimate the average display name length of Twitter users who tweeted `privacy`. In contrast to AVG(#followers) shown above, this requires substantially smaller number of queries as this measure has a lower vari-

Table 4.3. Microblog Analyzer: Average Percent Improvement of MA-TARW

Keyword	MA-SRW (AVG)	MA-SRW (COUNT)	M&R (COUNT)
`Boston`	39	44	72
`Oprah`	27	37	67
`Simvastatin`	29	41	74
`$WMT`	33	51	78
`Lipitor`	24	47	76
`Tunisia`	33	31	53
`Tahrir`	41	55	61

ability than that of number of followers. Figure 4.11 shows that MA-TARW seems to leverage this aspect by essentially "skipping" such edges (which would have often been intra-level edges).

Next we evaluate our algorithms on Google+. Figures 4.12 and 4.13 show the performance of estimating the average display name length and count of male users (gender is generally missing from Twitter profiles, and hence we did not use it as a condition above) who posted `privacy` during the time period. We notice that MA-TARW outperforms the competing algorithms. It must be noted that the absolute query cost is much higher than in Twitter. This is to a large extent due to the fact that APIs of Google+ (such as Activity search) returns at most 20 results per invocation compared to 200 in Twitter's timeline API.

Finally, we evaluate our algorithms on Tumblr. Here, we evaluated the average number of likes obtained by posts with *textual* content containing the keyword `privacy`. Figure 4.14 shows that MA-TARW has the best performance.

4.7 Related Work

Graph Sampling Through Random Walks: A number of existing papers have studied the problem on sampling large graphs [46, 60, 42, 51] while [43, 44, 49] specifically focus on online social networks. Sampling techniques and the ground truth definition vary depending on whether the global topology is known [51, 43] or unknown. For the latter, [43, 51] compared the efficiency of various sampling techniques such as simple random walk (SRW), Metropolis-Hastings (MHRW), BFS and DFS. [43] also studied the problem of running multiple, parallel random walks. We used SRW as the basis of MA-SRW as [43] reported that SRW is typically 1.5-8 times faster than MHRW, which was observation as well.

Analytics of Twitter and Other Microblogs: While there has been plethora of work on using social media data from Twitter and other microblogs on specific analytics tasks (typically over current and future data), our paper is the first to study the problem of aggregate estimation over historic data. [61] provides an high level overview of possible analytics tasks over Twitter. Other analytics tasks include monitoring trends[62], predicting stock prices [63], topical expertise [64], measure information propagation in Twitter [65], such as in the context of natural disasters. There has been a set of paid and free third party services such as Sysomos, Topsy, Trendsmap etc that allow you to perform simple analytics tasks (such as monitor popular trends, analyze your tweeting/retweeting behavior, visualize your social network etc). However, none of the free ones allow analytics over historic data and even the paid ones offer simple, canned analytic options.

Search Engine Analytics: Another category of related research is analytics over a search engine's corpus (e.g., [66]) - simply because a microblog service can be considered as a search engine (collecting, indexing and publishing documents posted

by all users). However, search-engine-analytics techniques cannot be directly applied because of the limitation of search interface provided by microblogging services. Note that a key assumption made by all existing search-engine-analytics techniques is that the search interface can reveal *all* documents in the corpus (through answers to a very large set of search queries). This, unfortunately, is not the case for microblogging services. For example, Twitter search API is limited to tweets published in the last week[38]. These limitations prevent existing search-engine-analytics techniques from being applied.

4.8 Conclusions

We proposed novel solutions to perform aggregate query estimation on microblogging data that exploit the provided user timeline API calls. We showed how to define a conceptual level-by-level subgraph of the social graph that allows dramatically more efficient random walk-based sampling. Then, we further improved our solution by proposing a novel topology-aware navigation strategy on the level-by-level subgraph that significantly outperforms existing random walk sampling methods. Theoretical analysis and experiments over microblogs confirm the effectiveness of our solutions.

CHAPTER 5

Mining Frequent Featuresets over Structured Items

5.1 Introduction

5.1.1 Frequent Featureset Mining Problem

Technological progress has now enabled businesses to collect fine-grained information about how users interact with their websites and applications. Such information can include the articles read, movies watched, items purchased etc. We consider a database of *structured items* where each item can be described through a set of *features*. Intuitively, we can represent each feature as a boolean attribute whose presence or absence in an item can be observed. By interacting with such items, users generate *item transactions* (transactions over items.) For example, the set of articles read by the user on any given day forms a transaction.

Frequent itemset mining is an important first step in data analysis for a broad class of applications. It returns a collection of itemsets (set of items) that are most commonly consumed together. A huge body of research literature studies this problem under various scenarios. In contrast to prior work that tries to identify *itemsets* from item transactions (under various conditions), we consider a novel problem of discovering frequent *featuresets* (set of features) from item transactions.

> Frequent Featuresets Mining (FFM) problem: Given an database containing transactions over structured items, identify the set of frequent featuresets.

The frequent featureset mining problem has variety of applications and is general enough to handle any scenario where transactions over structured items are performed.

- Consider a music website such as *last.fm*. Here the items correspond to songs and the features are the artists associated with the song. The set of songs listened in a session can be treated as item transaction. FFM tries to identify the artists whose songs are listened together (without user mentioning why she listens to a specific song).
- Consider a news website such as *New York Times*. Here the items correspond to news articles, the features are the named entities (such as Obama or Egypt) in the article, while a transaction could be the set of articles read by a user in a session. Instead of identifying the most commonly read articles (as done in traditional itemset mining), FFM tries to identify the *features* that are consumed together.
- Consider a movie website such as *IMDB*. The movies correspond to items while the features could be actors, directors, genre etc. The set of movies watched (or rated) by a user in a given period time could be considered as a transaction. FFM tries to identify the set of features (such as an actor, director combination) that are consumed together.

Problem Novelty: In contrast to traditional itemset mining, we consider items that are structured with rich features. Further, while the user interaction with the item could be observed, the corresponding user interaction with features is typically hidden. For example, while *New York Times* knows *which* articles the user read, it does not know *why* (what features in the article led the user to read them). Our objective in this paper is to identify frequent patterns over these "hidden" interactions. Notice that identifying the set of articles most frequently read belong to

the traditional itemset mining problem. However, using the entire article transaction log to identify the user's favorite topic is a featureset mining problem. This problem is quite challenging as we have to identify the most *important* feature without any explicit feedback from the user.

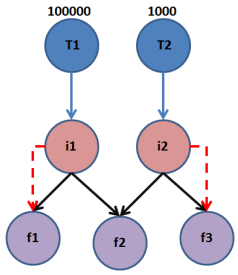


Figure 5.1. Hierarchical Representation of Transactions over Structured Items.

Consider Figure 1 that represents a small dataset of 2 item transactions $\{T_1, T_2\}$. For simplicity, we assume that each transaction contains only one item and that the user consumes an item due to a single feature. The frequency of the transaction is specified above it. T_1 has been observed 100,000 times while T_2 was observed 1000 times. The dataset has two items $\{i_1, i_2\}$ and three features $\{f_1, f_2, f_3\}$. The set of items in a transaction and the set of features present in an item are both observable. The blue arrows connect the transactions to the set of items while black arrows connect structured items to their features. The red dotted arrows highlight the feature which caused the user to consume the corresponding item. For example, the user consumed i_1 due to f_1. While the solid arrows are observable, the dotted ones are not. Hence the major challenge of FFM is to identify that f_1 is a frequent feature instead of f_2 without any explicit feedback from user.

Challenges: We now consider the various challenges in the general FFM problem where, an item could be consumed by a user due to any arbitrary subset of its

features. Further, the rationale for different users to consume the same item could be very different. For instance, on *New York Times*, one user may read an Sports section article due to of its coverage of games while another user may read the same article for its coverage of the players involved. Of course, user transactions on *New York Times* do not provide any information about why an item was consumed. This hidden mapping between items and features makes identifying frequent featuresets extremely challenging.

It might seem that running an existing frequent itemset mining algorithm by aggregating the features could identify frequent featuresets. For example, item i_1 in transaction T_1 is replaced $\{f_1, f_2\}$ while i_2 in T_2 is replaced by $\{f_2, f_3\}$. This approach, while intuitive, results in potentially incorrect frequent featureset due to a subtle pitfall - *the most frequent feature is not necessarily the most important one.*

Frequency Vs Importance. Using the dataset from Figure 1, this results in f_2 being the most frequent featureset followed by f_1. However, we can make a simple argument to negate this conclusion. If feature f_2 was indeed the dominant feature, then more users would have also consumed item i_2. Instead, it languishes with only 1000 transactions. Hence it is clear that the feature f_1 must have a higher importance than f_2 and must have been declared the most frequent featureset. The root cause for the incorrect conclusion is due to the fact that traditional itemset mining approaches considers *all* the features of the items in transaction instead of considering only the subset for which the user picked those items. We highlight other pitfalls of straightforward adaptations of existing deterministic and probabilistic frequent itemset mining algorithms in Section 5.4.

Combinatorial Explosion. Another major problem is that of *combinatorial explosion* of potential featuresets that could have generated the item transaction database. Consider for instance a single transaction of ten items with 5 features

each. If the features of items in the transaction do not overlap, then are there are 50 different features. Even if we assumed that a user picked an item for a single feature, there are 5^{10} potential featuresets that could have generated this item transaction. If we allow the user to choose an item over any subset of its features, then there are exactly 2^5 feature combinations to pick it. Thus, there are $(2^5)^{10} = 2^{50}$ hidden feature transactions that would generated the single item transaction.

5.1.2 Outline of Our Approach

Intuitively, any deterministic approach could not be used to solve the FFM problem. Unlike deterministic frequent itemset mining where the presence of an item in a transaction is known with certainty, it is hard to say which subset of features of that item lead the user to pick that item in the transaction. The difficulty comes from the fact that the presence of a feature in a transaction depends on whether the user was interested in that feature while generating the transaction rather than on the presence of that feature in one of the items of the transaction.

We tackle this problem in two stages. First, we identify the potential reasons (featuresets) for which an item was consumed in the context of the transaction/user. Due to the limited user-item interaction information, it is unlikely that we could identify the featureset that generated the transaction with any certainty. Given an item transaction T, we enumerate the various featuresets that could have generated it. We introduce a novel *featureset uncertainty* model to represent the likelihood for each of the candidate featuresets to have generated T. Identifying this likelihood is the *first* fundamental problem we solve in this paper. We use constrained least squares based approach to estimate the likelihood. Our *second* problem seeks to mine the frequent featuresets under the featureset uncertain model. We propose an efficient dynamic programming based approach for this purpose. In an effort to improve

the performance, we also designed a scalable "approximation" algorithm with only a marginal decrease in accuracy.

Our experimental results over a number of large datasets show that the frequent featuresets identified by our algorithms have high qualitative score under a number of popular interestingness measures. Additionally, in contrast to traditional itemset mining, the availability of user information allows us to perform personalized featureset mining.

Summary of Contributions.

- We introduce and motivate the novel problem of *frequent featureset mining* for transactions over structured items.
- We describe various approaches to map item transactions to feature transactions and highlight their pros and cons. We also make observations about the subtle issues that render adaptations of traditional itemset mining inapplicable.
- We introduce a novel featureset uncertainty model and develop an efficient algorithm to estimate the likelihood that a featureset generated an item transaction.
- We develop diverse algorithms to mine frequent featuresets under featureset uncertainty model .
- We present a thorough experimental evaluation of our algorithms and study their scalability using large synthetic datasets.

5.2 Framework and Problem Definition

In this section, we define the data model and the necessary background. We formally state the two central problems addressed in the paper - estimating the likelihood that a featureset generated an item transaction and using this information

to mine frequent featuresets. We then describe two variants that occur in practice depending on whether the user who made the transactions is identifiable.

5.2.1 Framework

Structured Items and Features. Let $\mathcal{I} = \{I_1, I_2, \ldots, I_n\}$ be a universe of structured items. Each item $I \in \mathcal{I}$ could be described as a set of features. A *feature* is a property of the item whose presence or absence can be observed. Let $\mathcal{F} = \{f_1, f_2, \ldots, f_m\}$ be the universe of all features. An item can be equivalently considered as a tuple over boolean attributes $\{f_1, f_2, \ldots, f_m\}$. Given an arbitrary item I and feature f, $I[f] = 1$ if I contains f and 0 otherwise. We represent the set of features of an item I by $features(I)$.

As an example, consider an online news website such as New York Times. Each news article can be considered as a structured item. $\mathcal{I}$ is the entire news article catalog. The set of named entities present in an article can be considered as its features. $\mathcal{F}$ is the set of all the named entities covered by the catalog.

A non empty set of items, $X \subseteq \mathcal{I}$ is called an *itemset*. An itemset is called as *l-itemset* if it has exactly l items. Similarly, a *featureset* $F \subseteq \mathcal{F}$ is a non empty set of features. A *l-featureset* has exactly l features. An item I is said to contain a featureset F, iff $\forall f \in F, I[f] = 1$. We will henceforth use the word *item* to refer to structured item.

Item Transactions. Let $\mathcal{IT} = \{T_1, T_2, \ldots, T_N\}$ be a database of item transactions, $|\mathcal{IT}|$=N. We represent each transaction as a triple $< tid, uid, X >$, where *tid* is the transaction identifier, *uid* is the user identifier who made the transaction and $X \subseteq \mathcal{I}$ is the set of items in the transaction. A transaction is said to contain an itemset if all the items in the itemset are also present in the transaction. Further, the notation

$features(T)$ for a transaction T returns the set of features that are present in at least one item in T.

User-Transaction Interaction Information. We consider the database of item transactions $\mathcal{IT}$ to be available at two levels of granularity - aggregate and fine-grained. Under the *aggregate* granularity, the only information available are the transaction identifier and the items contained in the transaction. Under the *fine-grained* granularity, we also know the user id that identifies the user who made the transaction. This additional information allows us to group the transactions by users. We hasten to add that our algorithms do not require any additional profile information about the user. Under aggregate granularity, we would consider all the items to be performed by a single "average" user and use the same user identifier for all item transactions.

Such interaction can be represented via an *aggregate interaction vector* v where each component corresponds to the number of times a particular transaction was made. If the user who made the transaction is known, then we could compute, for each user u, an *individual interaction vector* v_u where each component provides the number of times u performed a given transaction. The interaction vector is normalized and has non-negative numbers that add upto 1.

Frequent Itemset Mining. The support of an itemset X, denoted by $sup(X)$, is the number of transactions in which X appears in the transaction database. Let $minSup \in (0, N]$ be an integer, where N is the number of transactions. An itemset X is considered frequent if $sup(X) > minSup$.

Typically, items occurring in a transaction are considered to be certain. In other words, an item exists in a transaction or it doesn't. However, there are scenarios where the presence of an item is uncertain and quantified probabilistically [67, 68].

For example, in our paper, we observe only the item transaction created by the user. However, the underlying featureset that generated the item transaction is typically unknown. A natural model to manage such uncertainty is to assign values to various featuresets based on the likelihood that they have created the item transaction.

Uncertain Transaction Databases. An *uncertain* item I [67] is one whose presence in an item transaction T is provided by its *existential probability* $P(I \in T) \in [0, 1]$. In contrast, a *certain* item either occurs or does not in a transaction. i.e. $P(I \in T) \in \{0, 1\}$. An *uncertain transaction* contains uncertain items. Finally, an *uncertain transaction database* [68] contains uncertain transactions.

Probabilistic Frequent Itemset Mining. We can see that the support of an itemset X, $sup(X)$ is a random variable in uncertain databases and no longer has a constant value. There are multiple candidate definitions that extend support of an itemset to uncertain scenario [69, 70]. However, [71] showed the two most popular definitions (based on expected support and frequent probability) have a tight correlation between them. Given an uncertain database, a minimum support *minSup* and an itemset X, the *frequent probability* of X is defined as:

$$Pr(X) = Pr\{sup(X) \geq minSup\} \tag{5.1}$$

X is considered to be a *probabilistic frequent itemset* if $Pr(X) \geq minSup$ is at least a constant probabilistic frequent threshold of pft .

Running Example. Table 5.1 describes a simple dataset with $n = 3$ items, $m = 4$ features and $N = 3$ transactions that will serve as a running example to highlight our various algorithms.

Table 5.1. Hidden Itemset Mining: Running Example

TId	**Item {features}**	**Count**
T_1	$I_1\{f_1, f_3\}, I_2\{f_3, f_4\}$	50
T_2	$I_3\{f_2, f_3\}, I_2\{f_3, f_4\}$	100
T_3	$I_1\{f_1, f_3\}, I_2\{f_3, f_4\}, I_3\{f_2, f_3\}$	1000

5.2.2 Featureset Uncertainty Model

Model for Generating Item Transactions We first present an intuitive generative model for item transaction using their component features. As we argued in the introduction, a user picks an item due to a subset of its features. Such a behavior extends for each item in a transaction. To generate an item transaction T, the user first picks the size of the transaction, $|T|$. Each item in transaction is chosen as follows : the user first picks a featureset $F \subseteq \mathcal{F}$ according to a probability distribution. Then among the items that contain all the features in featureset F, she selects an item uniformly at random. This process is repeated $|T|$ times to generate the item transaction.

Generating Featureset. Recall that each item in a transaction was chosen by the user due to a subset of its features. Given a transaction $T = \{I_1, I_2, \ldots, I_{|T|}\}$, let F_i be the subset of features for which item I_i was chosen. We refer to the union of these featuresets $G_T = \cup_{i=1}^{|T|} F_i$ as the generating featureset for the item transaction T. In other words, G_T is the "ground-truth" that generated T. Ideally, if our aim is to identify the frequent featuresets, the frequent mining algorithm must be run over G_T for each item transaction $T \in \mathcal{IT}$.

Featureset Uncertainty Model. However, in practice, it is unlikely that we will be able to ascertain the featureset that generated a transaction deterministically. Further, for any given item transaction T, there are numerous possible featuresets

that could have generated T. The set of all featuresets that could have generated T is given by, $\mathcal{G}_T = \{G_T | G_T \subseteq \mathcal{F} \wedge |G_T \cap I| > 0 \; \forall I \in T\}$.

Given a single transaction T and no additional information, we cannot ascertain which of the featureset $F_T \in \mathcal{G}_T$ could have generated T. In our paper, we associate with each featureset the probability that it could have generated the transaction given other item transactions. Of course, given a single item transaction T, each generating featureset in $\mathcal{G}_T$ has a uniform probability (as we do not possess adequate information to claim otherwise). However, given multiple item transactions, it is possible to assign different likelihoods to each generating featureset. Hence, frequent featuresets can be obtained by running uncertain frequent mining algorithms over $\mathcal{FT}$.

Featureset Likelihood. The likelihood of a featureset F for a given transaction T is defined as the conditional probability that F was the generating featureset for T given the entire item transaction dataset $\mathcal{I}$ and all possible generating featureset G_T for T. The featureset likelihood of transaction is a probability distribution over all featuresets in its generating featureset. Of course, any featureset not in $\mathcal{G}_T$ will have a likelihood of 0. We would like to note that the likelihood of a featureset formalizes the notion of importance first described in Introduction.

Problem 1 (**Featureset Likelihood Estimation: FLE).** Given an item transaction database $\mathcal{IT}$ containing user transactions over structured items and aggregate interaction vector v, estimate the featureset likelihood for each transaction $T \in \mathcal{IT}$.

5.2.3 Frequent Featureset Mining

Once the relative likelihood of the generating featuresets of an item transaction are identified, the next step is to use this information to mine frequent featuresets.

Problem 2 (**Frequent Featureset Mining : FFM).** Given an item transaction database $\mathcal{IT}$ containing user transactions over structured items, the corresponding

featureset likelihood and minimum support threshold *minSup*, identify the set of frequent featuresets with expected support of at least *minSup*.

5.3 Overview of Our Approach

In this subsection, we describe the high level components of our approach that also provide a roadmap to the rest of the technical sections.

Challenge I: Enumerating Generating Featuresets. The first major challenge is to generate all possible featuresets that could have generated a given item transaction. However, even for small item transaction, this number could be exponential. Hence we make a simplifying assumption by noting that most users consume an item due to a small subset of its features. Recall that each item in the transaction must have at least one feature in the generating featureset. If we consider each item as a set with its feature as the elements, then a candidate generating featureset is a *hitting set* (a set of elements that has non empty intersection with all the other sets [72]). Section 4 talks about generating featuresets in more detail.

Challenge II: Identifying Likelihood of Generating Featuresets. Once we have identified the generating featuresets for each item transaction in the dataset, we represent them as a bipartite graph where one partition corresponds to item transactions while the other partition corresponds to featuresets. An edge exists between a transaction T and a featureset F if F was a generating featureset for T. Given this bipartite graph, the next challenge is to identify the relative likelihood of each featureset to generate the item transaction. We treat this as a constrained optimization problem, the solution of which allows us to order the generating featuresets of an item transaction based on the likelihood that it generated the transaction. Section 4 talks about identifying the likelihood of featuresets in more detail.

Challenge III: Mining Frequent Featuresets. Once the relative likelihood of the generating featuresets of an item transaction are identified, the next step is to identify the frequent featuresets. We first describe an exact but inefficient algorithm followed a much more efficient but approximate variant. Section 5 talks about mining featuresets in more detail.

Challenge IV: Scaling Featureset Mining. Numerous hurdles exist in each of the prior stages that hinder our ability to develop scalable solution for frequent featureset mining. We utilize sampling as the key technique to achieve scalability. We propose two different algorithms based on sampling the item transaction database and sampling the item transaction-featureset bipartite graph that provide a nice tradeoff between performance and quality . Section 6 talks about our scalability approaches.

5.4 Baseline Techniques

In this section, we describe two intuitive baseline ideas for solving the frequent featureset mining problem and point out their respective pitfalls which motivate our proposed approach. Both techniques solve the problem by transforming the item transaction database into a feature transaction database and utilize existing itemset mining algorithms.

5.4.1 Adapting Frequent Itemset Mining Algorithms.

In this subsection, we describe two approaches that transform the *certain* item transaction database into another *certain* feature transaction database.

Consider an obvious approach that transforms item transactions into transactions over features by using the union of features present in all the items in the transaction. This approach could lead to inaccurate results as it ignores two important facts: (a) The frequency of features within a transaction. (b) The combination

of features for which a user picked the item. If multiple items in a given transaction contain the same feature (for example, multiple articles read by a user contains the topic *Egypt*), then it is highly likely that this feature must be given a higher weight. Taking the union of features fails in capturing this valuable information. Second, this approach considers *all* the features of the items in transaction instead of considering only the subset for which the user picked those items.

However, even assigning a weight based on its frequency does not solve the issue as we show below. Consider a more sophisticated approach where each item transaction T is converted to a feature transaction F_T by taking the union of the features of all items in T. In other words, $F_T = \{features(I) | I \in T\}$. Once the feature transaction database is obtained, we can apply classical frequent itemset mining algorithms such as APriori[73] with appropriately chosen threshold. However, as pointed out in the example from Section 5.1, this also suffers from a subtle pitfall of mistaking frequency for importance.

5.4.2 Adapting Probabilistic Itemset Mining Algorithms.

In this subsection, we describe an intuitive approach that transform the *certain* item transaction database into an *uncertain* feature transaction database.

Attribute Uncertainty Model. The most basic model for describing an uncertain transaction database is the *attribute uncertainty model* [74]. Under this model, each attribute of the tuple is associated with a probability. If we consider a transaction as a tuple where the attributes are $\{I_1, I_2, \ldots, I_k\}$, then each item $I \in \mathcal{I}$ exists in T with an existential probability $P(I \in T)$. This model assumes that the presence of an item is independent of other items in the transaction.

Tag-Cloud Approach. This approach works by transforming the *certain* item transaction database into an *uncertain* feature transaction database. This is an adaptation of the technique used in [70]. We call this as a *tag cloud* [75] based approach (as tag clouds in social media are often generated this way). Each item transaction T is converted to an *uncertain* feature transaction F_T where the existential probability of a feature is computed as the ratio of number of items in which it is present to the total number of all items in the transaction. Formally, $F_T = \{f : \frac{\sum_{I \in T} |\{f\} \cap features(I)|}{|T|} | f \in features(T)\}$. Once the feature transaction database is obtained, we can apply any probabilistic frequent itemset mining algorithm such as UAPriori[69] with appropriately chosen threshold.

While more sophisticated that the previous approach, this algorithm still suffers from the flaws described in the introduction. It treats the most frequent feature(set) as also the most important one. As our counter example pointed out this may not always be valid.

5.5 Computing Featureset Likelihood

Recall from Section 5.2 that each item in a transaction was chosen by a user due to a subset of its features. The collection of all such subsets for the entire transaction is its generating featureset. If we have access to this information, frequent featuresets can be obtained by running traditional frequent mining over it. However, since this information is not available, we have to generate the possible featuresets and then evaluate their likelihood to be a generating featureset. We tackle this problem in two stages. We first generate all possible candidate featuresets that could have generated the transaction and then using this information estimate the likelihood for each transaction.

5.5.1 Generating Candidate Featuresets

In this subsection, our objective is to enumerate all possible featuresets that could have generated the transaction. Since each transaction could have an exponential number of generating featuresets, we use the idea of minimal hitting sets to substantially reduce this number.

Hitting Sets and Generating Featuresets. A key idea in reducing the combinatorial explosion comes from observing the close relationship between generating featuresets and hitting sets. Notice that for a featureset to be a potential generating featureset, it has to necessarily have at least one feature from each item in the transaction.

Given a finite set S (also called as universal set) and a collection C of subsets of S, the *hitting set* for C is a subset $S' \subseteq S$ such that it contains at least one element from each subset in C [72]. A hitting set is *minimal* if none of its proper subsets are also hitting sets. If we treat each item as a set of features and the union of all item features of the transaction as the universal set, then it is easy to see that any generating featuresets is also a hitting set. A further reduction in search space of featuresets can be achieved by observing that typically, users choose an item due to a small number of its features. This parsimonious behavior has been exploited in multiple prior work to generate concise models. This observation motivates us to identify *minimal generating featuresets* by using their relation to minimal hitting sets[76, 77].

Running Example. We can express the relation between transactions and the featuresets as a bipartite graph. The transactions form one partition while featuresets form another. An edge exists between a featureset F and a transaction T when F is a generating featureset for T. Using our running example, this bipartite graph would

have 3 transactions and 11 featuresets. However, if we use the minimal hitting set requirement, the graph has just 4 featuresets. Figure 5.2 shows the resulting graph.

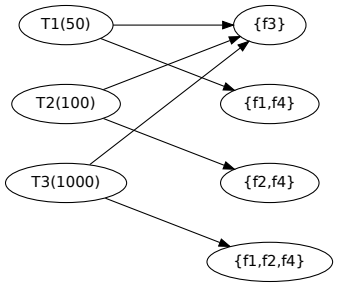


Figure 5.2. Bipartite Graph with Minimal Generating Featuresets.

Enumerating Minimal Generating Featuresets.

We describe a simple randomized algorithm to identify minimal hitting sets containing frequent features with high probability. The algorithm starts with an empty candidate hitting set. It then randomly picks a feature and adds it to the candidate. This process is repeated till all items in the transaction are covered resulting in a minimal generating featureset. Each feature is picked with probability proportional to its frequency in uncovered items. In other words, a feature that is present in multiple items has a higher likelihood of being chosen. This enumeration algorithm could be terminated after each featureset in the collection is returned by at least two different invocation of the randomized algorithm This heuristic is also referred to as Good-Turing test. Consider a simple invocation over transaction $T_3 = I_1\{f_1, f_3\}, I_2\{f_3, f_4\}, I_3\{f_2, f_3\}$. The algorithm starts with an empty set. Suppose it picked feature f_3 - it will immediately terminate as all items are covered. Suppose it picked feature f_1. Then items I_2, I_3 are not covered. Since f_3 occurs in both items it is picked with probability $\frac{2}{4} = \frac{1}{2}$ while f_2 and f_4 are picked with probability $\frac{1}{4}$ in the next iteration.

Complexity of Enumerating Minimal Hitting Sets. Identifying a single minimal hitting set requires a time complexity that is linear in the size of all features. However, *counting* all minimal hitting sets is #P-Complete [78]. It is possible to derive tighter bounds with additional information such as the maximum number of features in any item and the maximum transaction size in the dataset by using the idea of parameterized complexity [78].

5.5.2 Estimating Featureset Likelihood

Given an item transaction T, Section 5.5.1 identifies the set of featuresets $\mathcal{G}_T$, that could have generated it. However, given the limited information, we do not know the actual generating featureset. We used the featureset uncertainty model to express the probability that a given featureset generated the transaction T. In this section, we provide an optimization formulation to compute these values.

User-Transaction Interaction Information. Users generate transactions while interacting with items. Such interactions could be represented differently at aggregate or individual levels. At the aggregate level, we do not have access to the id of user who created the transaction. Instead, we treat all transactions to be performed by an "average" user. Such interaction can be represented via an *aggregate interaction vector* v where each component corresponds to the number of times a particular transaction was made. If the user who made the transaction is known, then we could compute, for each user u, an *individual interaction vector* v_u where each component provides the number of times u performed a given transaction. The interaction vector is normalized and has non-negative numbers that add upto 1. For the rest of the section, we assume the average user case while in Section 5.6, we describe how to customize the optimization formulation in the presence of user identity.

Transaction-Featureset Transition Matrix. The output of Section 5.5.1 can be succinctly summarized as a bipartite graph where transactions form one partition while featuresets form another. An edge exists between a featureset F and a transaction T, if F could have generated T. Figure 5.2 shows the graph for our running example.

We assume a column stochastic matrix $\mathcal{T}$ in which rows correspond to transactions while columns are the featuresets. Each cell $\mathcal{T}_{ij}$ provides the probability that an average user would generate the transaction T_i if she is interested in featureset F_j. We can construct a boolean matrix $\overline{\mathcal{T}}$ that represents the transaction-featureset bipartite graph where $\overline{\mathcal{T}}_{ij} = 1$ if transaction T_i could have been generated by featureset F_j. We also assume that if a featureset F could not have generated a transaction T, then the probability that a user would have generated T to be 0. In other words, $\mathcal{T} \leq \overline{\mathcal{T}}$. There are multiple ways to construct $\mathcal{T}$ from $\overline{\mathcal{T}}$. For example, we can assume a uniform distribution where, given a featureset F, all the transaction that could have been generated by F are chosen uniformly at random. Of course, it is possible to have a non-uniform distribution if we have additional domain knowledge (for example - given an actor, users are more likely to see a movie where the actor starred than one where he doesnt). Our likelihood estimation method is oblivious to the distribution of $\mathcal{T}$.

Featureset Likelihood Vector. As we described in Section 5.4, it is not possible to accurately identify the likelihood of featureset for a transaction in isolation. It is necessary to utilize the featuresets of other transactions for this purpose. As an example, given a single movie, it is not possible to confidently ascertain which feature caused an user to watch it. However, if we see other movies by the same actor (and with high transaction count), our confidence increases. Due to this reason, we

identify the featureset likelihood globally (across all transactions) instead of a single transaction. Intuitively, the global featureset likelihood vector w can be thought of as the "featureset preference vector" of a typical user. In other words, this provides a probability distribution over various featuresets. Frequent featuresets will have a higher value than less frequent featuresets. This vector is stochastic (all entries are non-zero and sum upto 1).

Given the prior notations, we can now formally respecify our generative model for item transactions from Section 5.2. A typical user has a "featureset-preference" vector that describes a distribution over the featuresets. The user chooses a featureset using that distribution. Once the featureset F_j is choosen, the user looks at the corresponding column in $\mathcal{T}$ and chooses a transaction with probability proportional to $\mathcal{T}_{ij}$. Intuitively, the relationship between various entities can be summarized by:

$$\mathcal{T}w = v \tag{5.2}$$

Computing Global Featureset Likelihood. From Section 5.5.1, we obtained the transaction-featureset bipartite graph. Using this information, we can compute the transaction-feaureset transition matrix $\mathcal{T}$ by assuming uniform distribution. We are also provided with the aggregate interaction vector v. Our aim now is to identify the featureset likelihood vector w.

Notice that typically, the number of featuresets far outnumber the number of trasactions. Hence the linear system of equations expressed by Equation 5.2 is overdetermined and has no solution. We can define an error metric based on the reconstruction error, $Error(v - \mathcal{T}w)$. For the purpose of our paper, we use the L_2 error metric. Our problem boils to finding the best w vector that minimizes $||v - \mathcal{T}w||_2$.

Algorithm 13 FFM-AVG

1: **Input:** $\mathcal{IT}$

2: Compute aggregate interaction vector v from $\mathcal{IT}$

3: $\forall T \in \mathcal{IT}$, generate candidate featuresets

4: Form transction-featureset bipartite graph $\overline{\mathcal{T}}$ and estimate $\mathcal{T}$

5: $constraints = \{ \forall i \, w_i \geq 0, \, ||w||_1 = 1 \}$

6: $w = \underset{w}{\text{argmin}} \, ||v - \mathcal{T}w||_2$ subject to $constraints$

7: **return** w

The solution vector w must minimize the reconstruction error and must also satisfy some constraints. Algorithm 13 provides a pseudocode for the problem. We model this problem as a constrained optimization problem with non negativity and stochastic constraints. Specifically, the optimization with L_2 metric corresponds to a constrained least squares problem with stochasticity constraints. Due to how the objective function is defined, this falls under a subset of convex optimization problem for which optimal solutions can be computed efficiently.

Complexity. The constrained optimization problem has a worst case complexity of $O(N^3)$. However, there exist very efficient iterative algorithms that can obtain the optimal solution in few iterations [79].

5.6 Mining Frequent Featuresets

Let us recap what we have achieved so far. We started with a database of item transactions with the objective of identifying frequent featuresets. Using a novel uncertainty model, we expressed each item transaction as a collection of featuresets that could have generated it. Using this transaction-featureset bipartite graph as a base, we used a constrained least squares approach to identify the global likelihood

for each featureset. In this section, we use this information to identify the frequent featuresets. We first describe an algorithm that performs mining directly over the featureset uncertainty model. We then design an efficient yet approximate algorithm that transforms the featuresets and their likelihoods into feature transactions which can then passed to any state of the art probabilistic itemset mining algorithms.

5.6.1 Exact Algorithm for Mining Frequent Featuresets

In this subsection, we describe an exact algorithm *FFM-EXACT* for identifying all frequent featuresets. This algorithm takes as input a single featureset F and computes whether F is a probabilistic frequent featureset. Our algorithm is based on dynamic programming and runs in polynomial time and is adapted from the approach first described in [80].

Frequent Featuresets and Probabilistic Heavy Hitters. There exists a close parallel between the concept of frequent featureset and that of probabilistic heavy hitters in uncertain data [80]. Given an uncertain database, an item t is considered as a (ϕ, τ)-probabilistic heavy hitter if the probability that t is heavy hitter (i.e. occurs more than fraction of ϕ in a possible world) is greater than τ. We can immediately see that if we treat each featureset F as an item (and compress other featuresets of a transaction to $\overline{F}$), then verifying whether F is a (*sup(X), pft*)-heavy hitter corresponds to finding if the featureset is probabilistically frequent. Our exact algorithm *FFM-EXACT* takes this approach.

This algorithm takes as input a single featureset F and the per-transaction featureset likelihood estimated from Section 5.5. It then converts each item transaction T into an uncertain transaction with two possible items F and $\overline{F}$. The existential probability (for a featureset F and for a transaction T) $Pr_T(F)$ is set to estimated

featureset likelihood if F was a generating featureset of T. Else the value was set to 0. The existential probability of $\overline{F}$ is computed as $1 - Pr_T(F)$.

Given this information, we could use the algorithm in [80] to estimate the probability that F is a frequent itemset. This algorithm is based on dynamic programming. We create a two dimensional table B_F with N rows and N columns. The cell $B_F(i, j)$ is interpreted as the probability that F was the generating featureset in exactly i item transactions out of the first j item transactions. Using this interpretation, the table can be filled as follows.

Base Case:

$$\begin{aligned} B_F[0,0] &= 1 \\ B_F[i,0] &= 0 \quad (i \geq 1) \\ B_F[0,j] &= \begin{cases} B_F[0,j-1] & \text{if } F \in T, j \geq 1 \\ B_F[0,j-1](1 - Pr_{T_j}(F)) & \text{if } F \notin T, j \geq 1 \end{cases} \end{aligned} \tag{5.3}$$

Induction Step:

$$B_F[i,j] = \begin{cases} B_F[i,j-1] & \text{if } F \notin T \\ B_F[i,j-1](1 - Pr_{T_j}(F)) + & \\ \quad B_F[i-1,j-1]Pr_{T_j}(F) & \text{if } F \in T \end{cases} \tag{5.4}$$

Once the table is filled the probability that F is a frequent featureset can be computed as in [80]. This step is repeated for each featureset and runs in $O(N^2)$ per featureset.

5.6.2 Approximate Algorithm for Mining Frequent Featuresets

While the previous algorithm is exact and produces accurate results, it is prohibitively expensive. In this subsection, we design an *approximate* algorithm *FFM-APPROX* by sacrificing some accuracy in favor of dramatically improved efficiency.

From Generating Featuresets to Feature Transactions. Our next step is to use this collection of transactions and the featuresets with likelihood weights to identify the frequent featuresets. Unfortunately, there exist no algorithm to identify the frequent featuresets. Most uncertain frequent mining problems work over attribute uncertainty model necessitating a transformation from featureset uncertainty model to attribute uncertainty model. This transformation might seem counterintuitive - we are moving from featureset uncertainty model (which is quite expressive), to a simpler uncertainty model. Further, it might seem that we are abandoning all the results of our expensive pre-processing. However, as out later experiments show, performing frequent featureset mining over this relaxed dataset provides a higher quality results than transforming to attribute uncertainty model directly.

In this section, we will focus on taking item transactions along with their candidate generating featuresets into a simpler feature transaction where each feature has an existential probability associated with it. Higher the probability, the more likely its corresponding feature to generate more items in the item transaction.

It is possible to generate feature transactions at multiple levels of granularity.

Average User - Per Transaction. This is the simplest transformation and the most generic. This is applicable in the case where the original item transaction database did not have any mechanism to identify the user who made the transaction. Further, in this approach we convert each item transaction into a corresponding feature transaction.

Algorithm 14 FFM-APPROX

1: **Input:** $\mathcal{G}$:< $\mathcal{T}$,$\mathcal{F}$,$\mathcal{E}$,$\mathcal{P}$ > - bipartite graph where $\mathcal{P}$ represents the likelihood distribution on $\mathcal{F}$

2: $\mathcal{FT} = \{\}$

3: **for all** $T \in \mathcal{IT}$ **do**

4: Collect generating featuresets GF_T=$\{F_i \in \mathcal{F}$ — $(F, T) \in \mathcal{E}\}$

5: Normalize the weights of the generating featuresets, $\forall F_i \in GF_T, P(F_i) = P(F_i)/\sum_{F_j \in GF_T} P(F_j)$

6: Initialize a feature transaction, $TmpF_T = \cup F_i | F_i \in GF_T$

7: Update weights of individual features, $\forall f_i \in TmpF_T$, $f_i.weight = \sum_{F_j \in GF_T} P(F_j) * F_j(f_i)$ if $f_i \in F_j$, 0 otherwise

8: $\mathcal{FT} = \mathcal{FT} \cup \{TmpF_T\}$

9: **end for**

10: Invoke uncertain frequent mining on $\mathcal{F}$

Algorithm 14 works as follows. First, for each item transaction T, we isolate all its generating featuresets from the transaction-featureset bipartite graph. We then normalize the featureset likelihoods so that they sum up to 1. The updated values provide, for each featureset, the probability that it could have generated T given other transactions. We then convert this to a feature transaction as follows. For each feature f in T, we identify all the generating featuresets of T in which it is part of. The existential probability of the feature f is given by the summation of the normalized likelihoods of the relevant featuresets.

Running Example. Recall that Figure 5.2 gave the minimal generating transactions for the dataset. After running the constrained optimization, the global likelihood for

each featureset is computed. Figure 5.3 shows an arbitrary subgraph for illustration purposes.

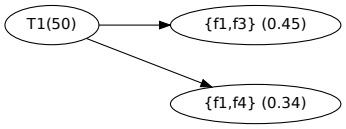


Figure 5.3. Bipartite Graph with Featureset Probabilities.

Let us convert item transaction T_1 to a feature transaction. The featuresets associated with T_1 are $\{f_1, f3\}$ and $\{f1, f4\}$ with values 0.45 and 0.34 respectively. We first normalize the values so that updated values are 0.569 and 0.431. In other words, the first featureset had a likelihood of 0.569 to have generated T_1. We now create the feature transaction from these normalized values. In this case, $\{f_1\}$ was present in both featuresets, its existential probability is $0.569 + 0.431 = 1$. For the other features, their existential probabilities are computed based on the single featureset they each belonged to. The final feature transaction is $\{f_1 : 1.0,\ f_3 : 0.569,\ f_4 : 0.431\}$.

Per User - Per Transaction. If we have information that could identify the user who made the transactions, we could design a user-specific feature transaction. In this approach, we start with the subset of item transactions performed by the user. Then we generate the candidate featuresets, identify their likelihood and use the previous method to convert to feature transaction. The major advantage is that if there is an item transactions made by two different users, the previous method would generate identical feature transaction. However, in this case, they could potentially generate different feature transactions.

Per User. Traditional frequent featuresets are executed over the entire dataset. However, it is also possible to identify frequent featuresets across users. A major

advantage is that this approach provides featuresets popular among a large fraction of the user population as opposed to featurests popular among a large fraction of transactions. This approach alleviates the effect of users who make a lot of identical transactions hence generating irrelevant frequent featuresets. We start from the subset of item transactions performed by the user. We then identify the candidate featuresets for all transactions and identify their global likelihood. Notice that the likelihood Then, we create a *single* feature transaction for the user. For each feature that is present in some transaction made by the user, its existential probability is the sum of featuresets in which the featureset is present.

Frequent Featureset Mining. Once we have converted the item transactions into feature transactions, they could be passed to any state-of-the-art uncertain frequent itemset mining algorithms [69, 81, 82] to obtain frequent featuresets. As the experiment section will confirm, the featuresets obtained are of higher quality than direct adaptations of certain or uncertain frequent itemset algorithms.

5.7 Scaling Featureset Mining

The previous sections described the various stages involved in taking a dataset of item transactions and processing it to identify the frequent featuresets. These algorithms are feasible for datasets with a relatively small number of features or items. Also recall from Section 5.5 that the worst case complexity of finding the likelihood is cubic in the size of transactions. In this section, we describe how to overcome the two major scalability challenges - large number of candidate generating featuresets and large size of item transaction databases. Surprisingly, the solution to both the problem rely on a common trick - sampling.

5.7.1 Sampling Transaction-Featureset Bipartite Graph.

Recall that one of the major challenges in identifying frequent featuresets is to enumerate all the major candidate featuresets of each transaction. This was similar to enumerating all the minimal hitting sets. In the worst case, this number could be exponential. This issue is amplified by the fact that the same process has to repeat for each item transaction in the database. Hence, in order to scale the algorithm, we have to identify the most important featuresets and retain only them. Any featureset that does not have a high likelihood could be removed.

We use a combination of heuristics to scale the algorithm. First, we relax the constraint that the generating featureset has to be minimal. This might seem counterintuitive as the number of non minimal generating featuresets far outnumbers that of minimal ones. However, this relaxation allows us to build featuresets that are simultaneousaly related to multiple transactions. We adapt the existing randomized algorithms described in Section 5.5.1 for this purpose. Given the expected support threshold, we identify the minimum number of transactions that any candidate generating featureset has to cover. For example, it might be the case that for a featureset to be frequent, it has to cover at least 100 transactions. The heuristic starts with a hitting set containing all the features in $\mathcal{F}$. This is obviously a valid featureset touching all transactions. We randomly drop features from it till the number of transactions it covers reach the threshold (say 100). We repeat the process as long as necessary to cover all the transactions in the database. We can observe that each iteration of this approach results in a featureset that covers at least 100 transactions. These featuresets also contains features that are, intuitively, more likely to be in the final frequent featuresets. A bottom-up variant of this heuristic is also possible where we start with an empty set and add features till it covers at least 100 transactions.

Refer to Algorithm 15 for pseudocode. This approach, in the worst case, degenerates to the algorithms we described in previous sections.

Complexity of FFM-APPROX-SAM1. From the pseudocode in Algorithm 15, it is possible to analyze its worst case complexity. From Section 5.5, recall that the complexity of FFM-APPROX is $O(N^3)$ where N is the number of transactions. Other operations in the algorithm can be ignored for asymptotic analysis. This implies that the algorithm FFM-APPROX-SAM1 has a time complexity of $O(N^3)$. In practice, the scalable variant is extremely efficient.

Algorithm 15 FFM-APPROX-SAM1

Input: $\mathcal{G}$: bipartite graph, r: max featuresets, t: threshold
Candidate featuresets $\mathcal{H} = \{\}$
for index=1 **to** r **do**
    Candidate hitting set $hs = \{\}$
    randomly add features from $\mathcal{F}$ until hs hits at least t item transactions
    $\mathcal{H} = \mathcal{H} \cup hs$
end for
$\mathcal{G}$ = Build-Bipartite-Graph($\mathcal{IT}, \mathcal{H}$)
w = FFM-APPROX($\mathcal{IT}$)
$\mathcal{FT}$ = FEATURE-TRAN-GEN($\mathcal{G}$)
FFS = frequent featuresets from ($\mathcal{FT}$)
return FFS

5.7.2 Sampling Item Transactions

An orthogonal approach to scale is based on sampling the item transaction database so as to get a smaller sample on which the algorithms are feasible. Intuitively, we pick a random sample of the database and run the algorithms described in previous sections over it. If the sample obtained is representative, then the frequent featuresets obtained with a scaled down threshold would also be frequent in the entire database. A key issue with this sampling strategy is the possibility of errors. Specifically, it is possible to get both *false negative* (when a featureset is frequent in the database but not in the sample) and *false positive* when a spurious frequentset is frequent in the sample but not in original dataset.

There are multiple strategies to reduce the number of false positives and negatives ranging from using a lower threshold, using multiple chunked samples instead of a single sample etc. One of the elegant algorithms to remove the false positives or negatives completely is the one proposed by Toivonen et al. [83]. We adapt a variant of this algorithm for our paper. We start with obtaining a random sample and run our algorithms over it. Once the frequent featuresets are identified, we collect the featuresets in the *negative border*. These are featuresets that are not frequent by themselves but *all* their immediate subsets are. We then make a pass over the entire dataset and verify the expected support of all the frequent featuresets identified from the sample and also their negative border. The set of frequent featuresets identified in the second pass are exactly the solution to our original problem. Refer to Algorithm 16 for pseudocode.

Complexity of FFM-APPROX-SAM2. The analysis of FFM-APPROX-SAM2 follows from that of FFM-APPROX-SAM1. The dominant factor is the procedure FFM-APPROX which takes time $O(N_{\mathcal{S}}^3)$. Unless the sample size is very small, this

Algorithm 16 FFM-APPROX-SAM2

1: **Input:** $\mathcal{IT}$

2: Generate sample $\mathcal{S}$ from $\mathcal{IT}$

3: $\mathcal{G}_{\mathcal{S}}$ = transaction-featureset bipartite graph for $\mathcal{S}$

4: w = FFM-APPROX($\mathcal{IT}$)

5: $\mathcal{FT}_{\mathcal{S}}$ = FEATURE-TRAN-GEN($\mathcal{G}_{\mathcal{S}}$)

6: FFS_S = frequent featuresets from ($\mathcal{FT}_{\mathcal{S}}$)

7: FFS_S^- = Compute negative border for FFS_S

8: FFS = Filter non frequent featuresets from $FFS_S \cup FFS_S^-$

9: **return** FFS

provides the time complexity. Of course, the scalable variant is in practice extremely efficient.

5.8 Experiments

In this section, we provide an extensive experimental evaluation of the efficiency and effectiveness of our proposals in mining hidden frequent featuresets on real and synthetic datasets. The experimental results confirm the superiority of our approach over direct adaptations of existing frequent itemset mining algorithms.

System Configuration. All our algorithms are implemented in C++. Experiments were conducted on Linux Ubuntu 13.04 machine, Intel Core i5 processor, 64 bit machine with 8 GB RAM. Timing values are taken by averaging over twenty runs.

5.8.1 Datasets

We used three different datasets to evaluate our algorithms. Two of them are real-world dataset while the third is a synthetic data allowing us to evaluate the scalability of our algorithms.

AJE dataset : This dataset consists of 2103 news articles published between April 2012 and February 2013 on Aljazeera english (AJE) website[1] - one of the most influential news media in the MENA region[2]. Each article comes with a set of comments (389k in total) posted by 35k different users from 179 different countries. We characterized each article by its features (i.e. topics, persons and locations) extracted using Open Calais[3]. We also evaluated other entity extraction libraries such as Alchemy and Stanford NER but found the results produced by all three services were remarkably similar. On average, each article has 7.5 features distributed as follows: 1.42 topics, 2.58 person names and 3.5 locations names (countries and/or cities).

An item transaction is defined as the set of articles commented by a user uid on the same date d, augmented with the comments a user posted on each article i.e. $T =< tid, uid, (a_1, c_1), \ldots (a_k, c_k), d >$, where c_i could be the concatenation of all comments posted by uid on a_i and data d. Grouping the data by users then by dates results in 15358 transactions. See Table 5.2 for further details.

AJE Ground Truth. We propose to use user comments as a proxy to uncover the hidden features that interested a user while reading an article. Given an article a with a set of features $features(a)$, a comment c posted by user u on article a, we assume that the features that most interested the user u are those of the intersection $features(a) \cap features(c)$. The ground truth for a transaction is the set of ground

[1] http://www.aljazeera.com

[2] MENA: Middle East and North Africa

[3] `http://www.opencalais.com/`

truth for all articles. While this ground truth is an approximation of the real ground truth (which ideally would be obtained by surveying users), we found that our approach is an efficient and automatic way to extract the ground truth. We conducted an user study where we evaluated different mechanisms to identify ground truths - our approach identified more entities than other alternate methods. Finally, the set of ground truth frequent featuresets (GT) is obtained by running a deterministic frequent itemset mining algorithm on the obtained feature transactions.

Synthetic dataset SD : The synthetic dataset was created to test the various facets of our algorithm. It was generated in two phases. In the first, we used IBM Quest Generator to generate 1000 structured items over 100 features with average feature size of 4. (here dataset transaction corresponded to items while dataset items corresponded to features). Once the structured items are known, we then create another dataset for the actual evaluation purposes. The output was a collection of feature transactions that also forms the ground truth. For each feature in the transaction, we chose the corresponding item (of item transaction) uniformly at random. For eg, suppose the feature transaction was $\{f_1, f_2\}$. We now look at all possible items with f_1 (resp f_2) and choose an item at random. items with their presence.

Last.fm Dataset LF: Last.fm [4] is a music website where users could listen to songs from internet radio stations or from their portable music devices. Songs corresponds to items. Each song is described using multitude of features including artist, genre, albums, record labels and semi-structured information via tags. The set of songs played by a user in a session correspond to the transaction. Last.fm provides a scrobbling API that allows programs to send and receive information about tracks listened. We built a Chrome extension that used Last.fm's API to monitor the set

[4] `http://www.last.fm/`

of tracks listened by the user. We recruited more than 2000 volunteers to install our extension. Our extension provides a gamified interface where it periodically inquires users about which of the track's features lead them to listen to it. The users were also allowed to enter free-form content using tags or other detailed description. These formed the ground truth over which our algorithms were evaluated.

Table 5.2. Hidden Itemset Mining: Characteristics of Dataset

Dataset	#Trans.	#Items	#Features	Avg Len
AJE	$15K$	2103	459	1.5667
SD(T10I4D100K)	$100K$	1000	100	10
LF	$40K$	5644	919	18

5.8.2 Evaluation metrics

The evaluation of our algorithms is quite tricky due to the limited amount of information available. Specifically, it is not makes sense to use traditional metrics used for exact itemset mining. Instead, we used measures that are commonly used in approximate frequent itemset mining [84]. A brief description is given below.

Recoverability. Recoverability measures how well a pattern mining algorithm recovers the ground truth featuresets (GT). For each ground truth featureset pattern GT_i, we first identify the frequent featureset FFS_i generated by our algorithm that best matches with it (based on the number of common features cf_i they share). In other words, the recoverability of GT_i is the largest percent of an featureset found by any pattern FFS_i that is associated with GT_i. This is performed as a weighted average as matching with a larger pattern counts much than matching with smaller patterns.

$$Recoverability \quad = \quad \frac{\sum_{i=1}^{|GT|} cf_i}{\sum_{i=1}^{|GT|} |GT_i|} \tag{5.5}$$

Spuriousness. It is possible for an algorithm to get a high recoverability by returning large featuresets. Spuriousness is a metric complementary to recoverability - it measures the number of features in the pattern that are not associated with original matching pattern. Further, precision can be computed as 1.0-spuriousness. For each FFS say FFS_i, we first identify the ground truth featureset GT_i, which share maximum number of common features (denoted as cf_i) between them. The number of spurious items for each FFS is, $|FFS_i|$ - $|cf_i|$. The equation to calculate the *spuriousness* over whole FFS is given below.

$$Spuriousness = \frac{\sum_{i=1}^{|FF|}(|FF_i| - cf_i)}{\sum_{i=1}^{|FF|} |FF_i|} \tag{5.6}$$

Significance. This metric combines both *recoverability* and *Spuriousness* in the same way F-measure does with *precision* and *recall*.

$$Significance = \frac{2 * (Recoverability * (1 - Spuriousness))}{(Recoverability + (1 - Spuriousness)} \tag{5.7}$$

Redundancy. This metric mitigates the effect of producing relevant but redundant FFS (i.e. frequent sets that are subsets of other frequent sets). Redundancy is measured by creating a matrix, M of size $|FFS|X|FFS|$. Each entry in the matrix,ffs_{ij} denotes the number of common items between FFS_i and FFS_j. Then we compute the sum of the upper triangular matrix of M and subtracting the diagonal entries to estimate the redundancy. This measure doesn't take the average over the number of FFS. Hence, it implicitly penalize if the number of FFS is too large.

$$Redundancy = \frac{\sum_{i,j=1...|FFS|, j \succ i} ffs_{ij} - \sum_{i=1...|FFS|} ffs_{ii}}{2} \tag{5.8}$$

5.8.3 Qualitative Evaluation

For the qualitative purposes, we test two algorithm. BASELINE is a simple tag cloud based approach defined in Section 5.4. We did not test the deterministic baseline as it consistently underperformed BASELINE. The frequent featureset mining algorithm *FFM-APPROX* treats all transactions are made by a single user. We evaluate the exact, approximate and sampling based variants of our algorithm. After identifying the frequent featuresets using baseline and our algorithms, we evaluate their quality using the evaluation metrics described previously.

Experimental Observations: Figures 5.4(a)-5.4(l) show how our algorithms perform against baseline for the three datasets. The major observations are as follows: (a) Our algorithms consistently out perform BASELINE for larger value of support (b) The exact algorithm `FFM-EXACT` have a higher score than the approximate versions. (c) The sampling based algorithms perform slightly worse than the approximate variants. `FFM-APPROX-SAM1` which is designed for speed has a lower accuracy than `FFM-APPROX-SAM2` that is optimized for accuracy.

We vary the minimum support from 0.1 to 0.5 and measure its impact over the evaluation metrics. Figures 5.4(a), 5.4(b) and 5.4(c) show the corresponding impact over recoverability. Higher values of recoverability is desirable. We can see that the recoverability of the algorithms increase as minimum support decreases. This expected behavior is due to the high number of FFS that are discovered at lower values of support, increasing the number of recovered GT. Our algorithms out performs BASELINE for higher values of support. Figures 5.4(d), 5.4(e) and 5.4(f) show the spuriousness scores achieved at different support values. A lower value of spuriousness

is desired. The figures show that our algorithms obtain a significantly lower value of spuriousness than BASELINE. Higher values are desirable for significance while lower values are desirable for redundancy. Figures 5.4(g)-5.4(l) show that our algorithms have a superior performance for both significance and redundancy.

5.8.4 Quantitative Evaluation and Scalability

Scalability. Figures 5.5(a)-5.5(c) show the runtime performance of all our algorithms. We measure three parameters - number of transactions (N), number of items (n) and number of features (m). As expected, the exact algorithm takes prohibitive amount of time and for large datasets, it took more than two days. The approximate variant is much faster than the exact version while the two sampling variants are orders of magnitude faster. We can also see that the number of transactions and features have a higher impact on running time than the number of items. This is to be expected as the major factor in the runtime is the number of featuresets which are directly impacted by the number of features.

Robustness of Sampling: In this set of experiments we measure the robustness of our sampling algorithms. We utilize the theoretical results from [83] as a heuristic to determine the minimum sample size. We seek to obtain a *representative* sampling by ensuring that, given an itemset S, the probability that the difference between S's relative frequency in the database and the sample varies by atmost a constant ϵ is less than a constant δ. Given ϵ, δ, [83] provides the minimum sample size that must be obtained regardless of database size. Figures 5.6(a)-5.6(c) show the robustness of our algorithms. Once the sample size exceeds 30K (minimum sufficient size for $\epsilon = \delta = 0.01$), the quality does not dramatically improve with higher sample size. Figures5.6(a), 5.6(b) show the results for recoverability and Significance respectively. The results of other metrics were similar and not included to conserve space.

Figure 5.6(c) shows that given a fixed sample size as suggested by [83] is sufficient to reach a good accuracy regardless of the size of the database.

5.9 Related Work

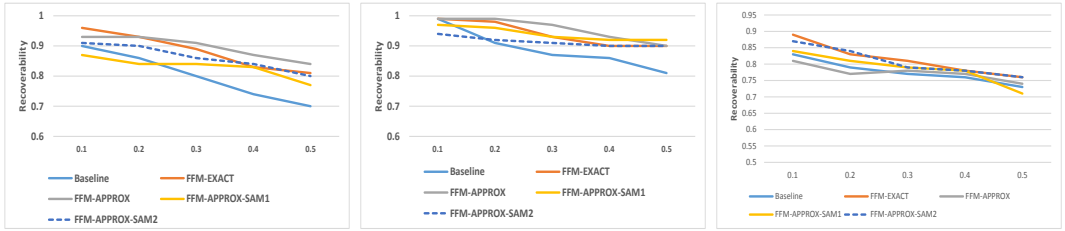
While there has been extensive work on Frequent itemset mining starting with [85], most of the proposed approaches and techniques assume the atomicity of items and hence aim at identifying frequent itemsets by mining transactions over items. To the best of our knowledge, we are the first to solve the problem of identifying hidden frequent featuresets by mining observed transactions using a novel featureset uncertainty model.

Uncertain Frequent Itemset Mining. Chui and al. [69] were the first to investigate mining frequent itemsets over uncertain transactions. Since then, an important amount of work has been done in this area (see [67] for a survey). There are two main approaches for mining uncertain frequent itemsets: expected support and probabilistic models, both of which consider the support as a random variable but with different interpretations. In expected support approaches [69, 81, 82], an itemset is frequent iff its expected support is greater or equal than a predefined threshold. Several well-known frequent itemset mining algorithms have been adapted to deal with the expected support such as UAPriori [69], UFP-Growth [81], and UH-Mine [82]. On the other hand, probabilistic frequent itemset mining approaches rely on the concept of *frequentness probability* which denotes the probability of an itemset support to be greater than a predefined minimum support [70, 86]. Only itemsets with frequentness probability greater than the minimum support are considered as frequent. This is typically solved using dynamic programming [70] or divide and conquer [86]. Tong et al. showed that these two definitions are equivalent [71].

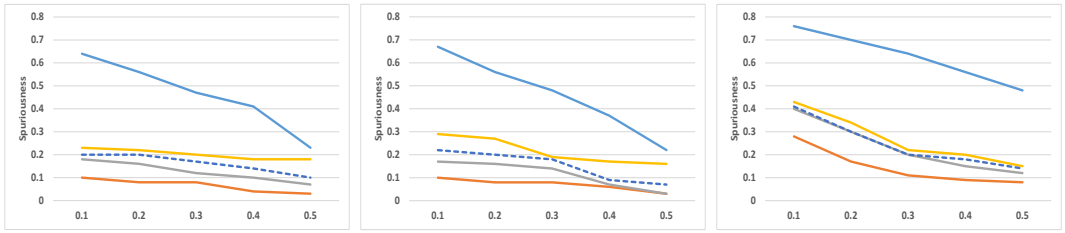
Uncertain Model in Probabilistic Database. There are three different models to represent uncertainty in relational databases: *tuple uncertainty*, *attribute uncertainty*, and *xtuple* uncertainty. In *tuple uncertainty* model, each tuple is associated with a score reflecting the probability of that tuple to exist in the database [87]. In *attribute uncertainty* model, the uncertainty is more fine grained where a probability score is assigned to each attribute value in a tuple [74]. Notice that both models are mapped into probability distribution over all the *Possible World* [88] where each *Possible* world is an deterministic instance of the database. The concept of *xtuples* is used in the ULDB model proposed in [68]. An xtuple could be seen as a probability distribution over a set of mutually exclusive tuples. Uncertain itemset mining approaches follow either the tuple uncertainty model [86] where a probability score is associated with each transaction, or attribute uncertainty model [70] where probabilities are associated to each item in a transaction.

5.10 Final Remarks

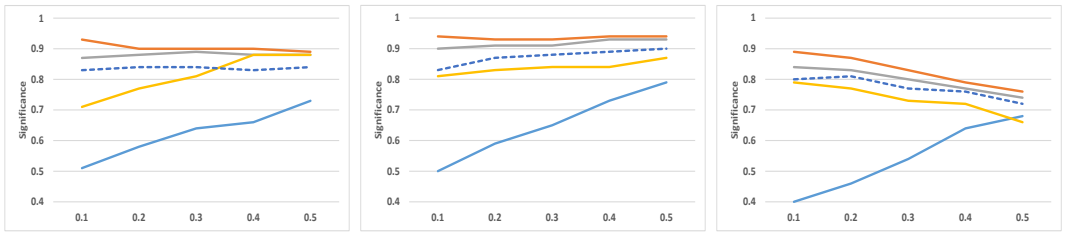
In this paper we study the novel problem of mining hidden frequent *featuresets* from item transactions. We motivated this novel problem with several illustrative examples and introduced a featureset uncertainty model. We developed a constrained least squares based approach to solve the problem of learning generating featureset likelihoods. We also developed two heuristics based on sampling to scale up our algorithm to real world problem sizes.



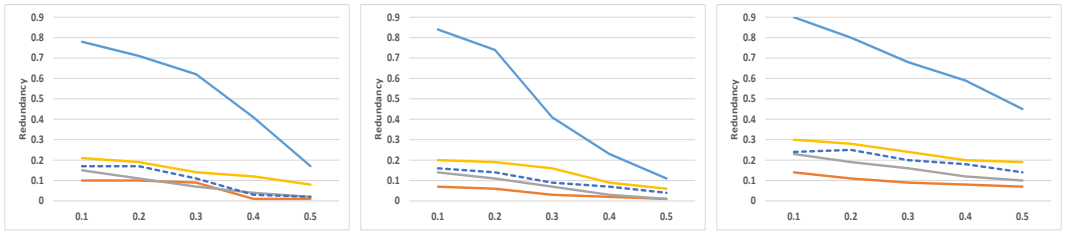
(a) Recoverability (AJE) (b) Recoverability (SD) (c) Recoverability at k (SD)



(d) Spuriousness (AJE) (e) Spuriousness (SD) (f) Spuriousness at k (SD)

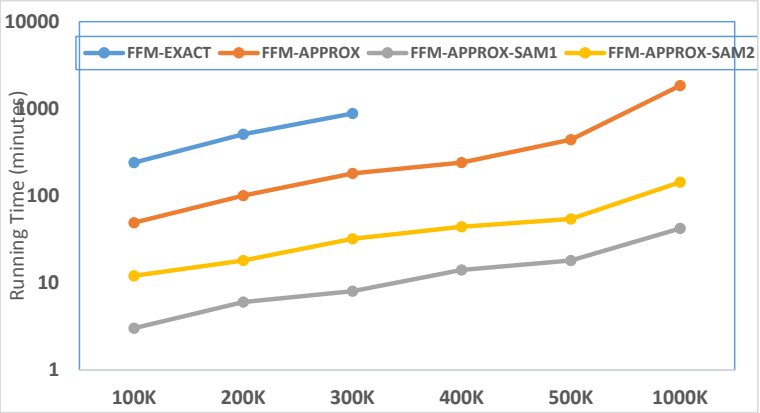


(g) Significance (AJE) (h) Significance (SD) (i) Significance at k (SD)

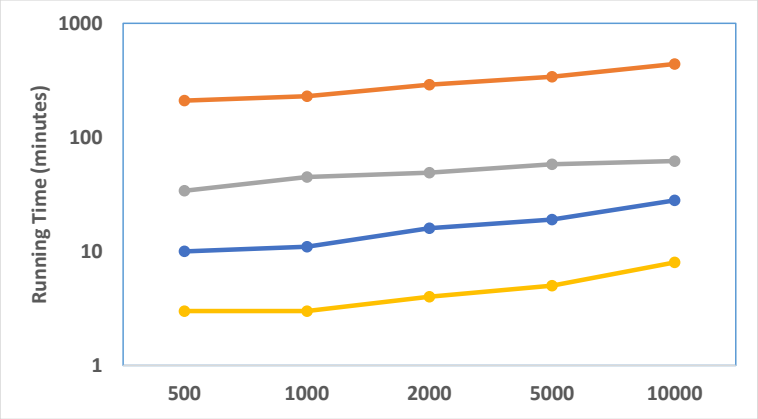


(j) Redundancy (AJE) (k) Redundancy (SD) (l) Redundancy at K (SD)

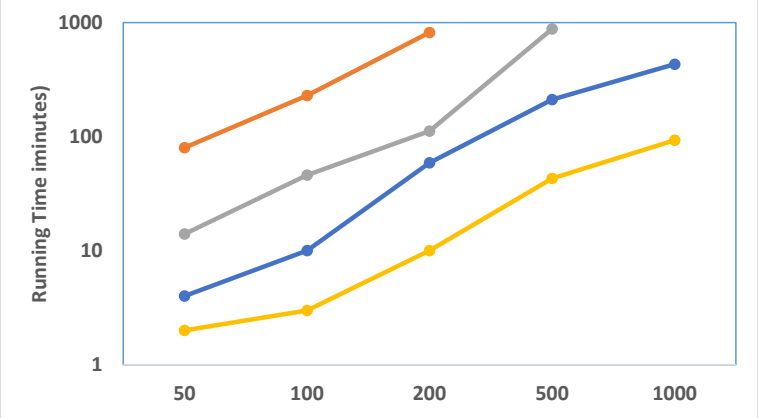
Figure 5.4. Qualitative Evaluation of Datasets with Varying Minimum Support.



(a) Number of Transactions(N)

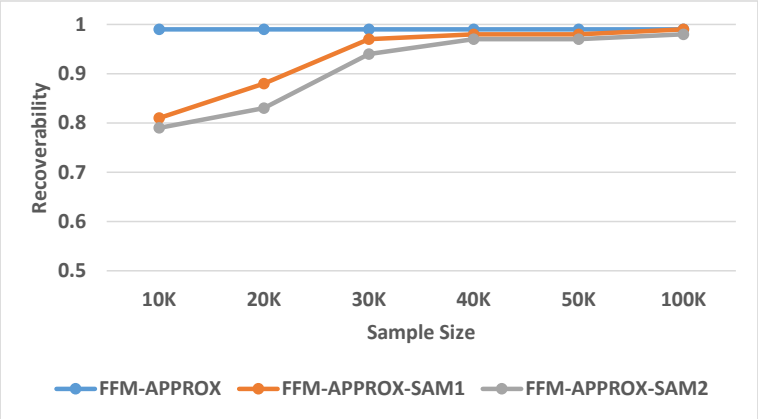


(b) Number of Items(n)

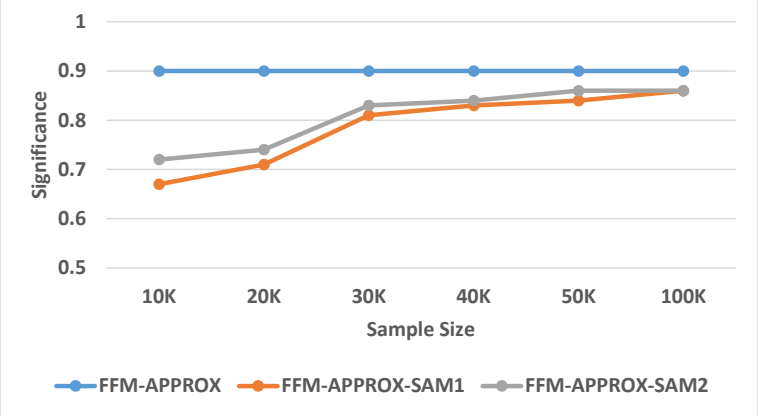


(c) Number of Features(m)

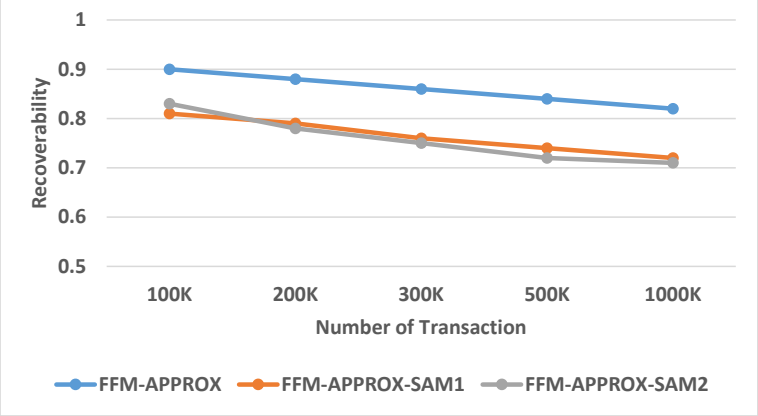
Figure 5.5. Hidden Itemset Mining: Effect on Running Time.



(a) Recoverability Vs Sample Size



(b) Significance Vs Sample Size



(c) Recoverability Vs No. Of Transactions

Figure 5.6. Robustness of Sampling Algorithms.

REFERENCES

[1] "Bright planet, deep web faqs, 2010," *http://www.brightplanet.com/the-deep-web/.*

[2] F. N. Afrati, P. V. Lekeas, and C. Li, "Adaptive-sampling algorithms for answering aggregation queries on web sites," *DKE*, vol. 64, no. 2, pp. 462–490, 2008.

[3] A. Dasgupta, X. Jin, B. Jewell, N. Zhang, and G. Das, "Unbiased estimation of size and other aggregates over hidden web databases," in *SIGMOD*, 2010.

[4] F. Wang and G. Agrawal, "Effective and efficient sampling methods for deep web aggregation queries," in *EDBT*, 2011, pp. 425–436.

[5] C. Sheng, N. Zhang, Y. Tao, and X. Jin, "Optimal algorithms for crawling a hidden database in the web," in *VLDB*, 2012, pp. 1112–1123.

[6] A. Dasgupta, G. Das, and H. Mannila, "A random walk approach to sampling hidden databases," in *SIGMOD*, 2007.

[7] X. Jin, N. Zhang, and G. Das, "Attribute domain discovery for hidden web databases," in *SIGMOD*, 2011.

[8] M. Hu and B. Liu, "Mining and summarizing customer reviews," ser. KDD '04, pp. 168–177. [Online]. Available: http://doi.acm.org/10.1145/1014052.1014073

[9] ——, "Mining opinion features in customer reviews," ser. AAAI'04, 2004, pp. 755–760. [Online]. Available: http://dl.acm.org/citation.cfm?id=1597148.1597269

[10] M. Wu, http://lithosphere.lithium.com/t5/science-of-social-blog/The-Economics-of-90-9-1-The-Gini-Coefficient-with-Cross/ba-p/5466, 2010, [Online; accessed 12-Feb-2013].

[11] J. Madhavan, D. Ko, L. Kot, V. Ganapathy, A. Rasmussen, and A. Y. Halevy, "Google's Deep Web crawl," *Proceedings of The Vldb Endowment*, vol. 1, pp. 1241–1252, 2008.

[12] M. Álvarez, J. Raposo, A. Pan, F. Cacheda, F. Bellas, and V. Carneiro, "Crawling the content hidden behind web forms," in *Proceedings of the 2007 international conference on Computational science and Its applications - Volume Part II*, ser. ICCSA'07. Springer-Verlag, 2007, pp. 322–333.

[13] D. J. Haglin and A. M. Manning, "On minimal infrequent itemset mining," in *International Conference on Data Mining*, 2007.

[14] J. Han, H. Cheng, D. Xin, and X. Yan, "Frequent pattern mining: current status and future directions," *DMKD*, 2007.

[15] K. Chang and J. Cho, "Accessing the web: From search to integration," in *Tutorial, SIGMOD*, 2006.

[16] A. Doan, R. Ramakrishnan, and S. Vaithyanathan, "Managing information extraction," in *Tutorial, SIGMOD*, 2006.

[17] S. Raghavan and H. Garcia-Molina, "Crawling the hidden web," in *VLDB*, 2001.

[18] E. Dragut, T. Kabisch, C. Yu, and U. Leser, "A hierarchical approach to model web query interfaces for web source integration," in *VLDB*, 2009.

[19] Z. Zhang, B. He, and K. Chang, "Understanding web query interfaces: best-effort parsing with hidden syntax," in *SIGMOD*, 2004.

[20] B. He, K. Chang, and J. Han, "Discovering complex matchings across web query interfaces: A correlation mining approach," in *KDD*, 2004.

[21] E. Dragut, C. Yu, and W. Meng, "Meaningful labeling of integrated query interfaces," in *VLDB*, 2006.

[22] B. He and K. Chang, "Statistical schema matching across web query interfaces," in *SIGMOD*, 2003.

[23] Z. Bar-Yossef and M. Gurevich, "Mining search engine query logs via suggestion sampling," in *VLDB*, 2008.

[24] K. Bharat and A. Broder, "A technique for measuring the relative size and overlap of public web search engines," in *WWW*, 1998.

[25] K. Liu, C. Yu, and W. Meng, "Discovering the representative of a search engine," in *CIKM*, 2002.

[26] M. Shokouhi, J. Zobel, F. Scholer, and S. Tahaghoghi, "Capturing collection size for distributed non-cooperative retrieval," in *SIGIR*, 2006.

[27] Z. Bar-Yossef and M. Gurevich, "Efficient search engine measurements," in *WWW*, 2007.

[28] J. Callan and M. Connell, "Query-based sampling of text databases," *ACM TOIS*, vol. 19, no. 2, pp. 97–130, 2001.

[29] P. Ipeirotis and L. Gravano, "Distributed search over the hidden web: Hierarchical database sampling and selection," in *VLDB*, 2002.

[30] Y.-L. Hedley, M. Younas, A. E. James, and M. Sanderson, "Sampling, information extraction and summarisation of hidden web databases," *Data and Knowledge Engineering*, vol. 59, no. 2, pp. 213–230, 2006.

[31] N. Bruno, L. Gravano, and A. Marian, "Evaluating top-k queries over web-accessible databases," in *ICDE*, 2002.

[32] I. Ilyas, G. Beskales, and M. Soliman, "A survey of top-k query processing techniques in relational database systems," *ACM Computing Surveys*, vol. 40, 2008.

[33] A. Dasgupta, G. Das, and H. Mannila, "A random walk approach to sampling hidden databases," in *SIGMOD*, 2007.

[34] A. Dasgupta, N. Zhang, and G. Das, "Leveraging count information in sampling hidden databases," in *ICDE*, 2009.

[35] ——, "Turbo-charging hidden database samplers with overflowing queries and skew reduction," in *EDBT*, 2010.

[36] S. Thirumuruganathan, N. Zhang, and G. Das, "Breaking the top-k barrier of hidden web databases," in *ICDE*, 2013, pp. 1045–1056.

[37] D. Horvitz and D. Thompson, "A generalization of sampling without replacement from a finite universe," *Journal of the American Statistical Association*, vol. 47, pp. 663–685, 1952.

[38] "Twitter search," in *https://dev.twitter.com/docs/using-search*, 2013.

[39] "Gnip," in *http://gnip.com*, 2013.

[40] "Datasift pricing," in *http://datasift.com/pricing/*, 2013.

[41] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," ser. SIGCOMM '07, 2007, pp. 29–42. [Online]. Available: http://doi.acm.org/10.1145/1298306.1298311

[42] M. Kurant, M. Gjoka, C. T. Butts, and A. Markopoulou, "Walking on a graph with a magnifying glass: stratified sampling via weighted random walks," ser. SIGMETRICS '11. [Online]. Available: http://doi.acm.org/10.1145/1993744.1993773

[43] M. Gjoka, M. Kurant, C. T. Butts, and A. Markopoulou, "Walking in facebook: a case study of unbiased sampling of osns," ser. INFOCOM'10, 2010, pp. 2498–2506.

[44] L. Katzir, E. Liberty, and O. Somekh, "Estimating sizes of social networks via biased sampling," ser. WWW '11, 2011. [Online]. Available: http://doi.acm.org/10.1145/1963405.1963489

[45] W. R. Gilks, *Markov Chain Monte Carlo In Practice*. Chapman and Hall/CRC, 1999.

[46] L. Lovász, "Random walks on graphs: A survey," in *Combinatorics, Paul Erdős is Eighty*, 1996, vol. 2.

[47] "Sysomos twitter usage statistics," in *http://www.sysomos.com/insidetwitter/*, 2013.

[48] K. Lerman and R. Ghosh, "Information contagion: An empirical study of the spread of news on digg and twitter social networks." *ICWSM*, vol. 10, pp. 90–97, 2010.

[49] A. Mohaisen, A. Yun, and Y. Kim, "Measuring the mixing time of social graphs," in *SIGCOMM '10*, 2010. [Online]. Available: http://doi.acm.org/10.1145/1879141.1879191

[50] M. E. J. Newman, "Modularity and community structure in networks," *PNAS*, vol. 103, no. 23, 2006.

[51] J. Leskovec and C. Faloutsos, "Sampling from large graphs," ser. KDD '06, 2006, pp. 631–636. [Online]. Available: http://doi.acm.org/10.1145/1150402.1150479

[52] L. Cowen, "Handbook of Capture-Recapture Analysis." *The Quarterly Review of Biology*, vol. 81, no. 3, p. 310, 2006.

[53] "Twitter api," in *https://dev.twitter.com/docs/api/1.1*, 2013.

[54] "Twitter Streaming API," in *https://dev.twitter.com/docs/streaming-apis*, 2013.

[55] J. Geweke *et al.*, *Evaluating the accuracy of sampling-based approaches to the calculation of posterior moments*. Federal Reserve Bank of Minneapolis, Research Department, 1991.

[56] "Sysomos twitter retweet statistics," in *http://www.sysomos.com/insidetwitter/engagement/*, 2013.

[57] L. Lovasz and R. Kannan, "Faster mixing via average conductance," in *STOC*, 1999, pp. 282–287.

[58] M. H. Hansen and W. N. Hurwitz, "On the theory of sampling from finite populations," *AMS*, vol. 14, no. 4, pp. 333–362, 1943.

[59] C. Domb, "On multiple returns in the random-walk problem," in *Proc. Cambridge Philos. Soc*, vol. 50. Cambridge Univ Press, 1954, pp. 586–591.

[60] E. M. Airoldi and K. M. Carley, "Sampling algorithms for pure network topologies: a study on the stability and the separability of metric embeddings," *ACM SIGKDD Explorations Newsletter*, vol. 7, no. 2, pp. 13–22, 2005.

[61] S. Kumar, F. Morstatter, and H. Liu, "Twitter data analytics," 2013.

[62] M. Mathioudakis and N. Koudas, "Twittermonitor: Trend detection over the twitter stream," ser. SIGMOD '10. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807306

[63] E. J. Ruiz, V. Hristidis, C. Castillo, A. Gionis, and A. Jaimes, "Correlating financial time series with micro-blogging activity," in *WSDM*. ACM, 2012, pp. 513–522.

[64] A. Pal and S. Counts, "Identifying topical authorities in microblogs," in *WSDM*. ACM, 2011, pp. 45–54.

[65] M. Mendoza, B. Poblete, and C. Castillo, "Twitter under crisis: Can we trust what we rt?" in *SOMA*, 2010.

[66] M. Zhang, N. Zhang, and G. Das, "Mining a search engine's corpus: Efficient yet unbiased sampling and aggregate estimation," ser. SIGMOD '11. [Online]. Available: http://doi.acm.org/10.1145/1989323.1989406

[67] C. C. Aggarwal and P. S. Yu, "A survey of uncertain data algorithms and applications," *IEEE TKDE*, 2009.

[68] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom, "Uldbs: Databases with uncertainty and lineage," in *VLDB*, 2006.

[69] C. K. Chui, B. Kao, and E. Hung, "Mining frequent itemsets from uncertain data," in *PAKDD*, 2007, pp. 47–58.

[70] T. Bernecker, H.-P. Kriegel, M. Renz, F. Verhein, and A. Zuefle, "Probabilistic frequent itemset mining in uncertain databases," in *SIGKDD*. ACM, 2009, pp. 119–128.

[71] Y. Tong, L. Chen, Y. Cheng, and P. S. Yu, "Mining frequent itemsets over uncertain databases," *VLDB*, 2012.

[72] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[73] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *SIGMOD*, New York, NY, USA, 1993, pp. 207–216. [Online]. Available: http://doi.acm.org/10.1145/170035.170072

[74] P. Sen, A. Deshpande, and L. Getoor, "Representing tuple and attribute uncertainty in probabilistic databases," in *Data Mining Workshops, ICDM*. IEEE, 2007, pp. 507–512.

[75] P. Venetis, G. Koutrika, and H. Garcia-Molina, "On the selection of tags for tag clouds," in *WSDM*, 2011.

[76] R. Reiter, "A theory of diagnosis from first principles," *Artif. Intell.*, vol. 32, no. 1, pp. 57–95, Apr. 1987. [Online]. Available: http://dx.doi.org/10.1016/0004-3702(87)90062-2

[77] T. Hofmann, "Learning what people (don't) want," in *EMCL*, ser. EMCL '01, 2001, pp. 214–225. [Online]. Available: http://dl.acm.org/citation.cfm?id=645328.650012

[78] P. Damaschke, "The union of minimal hitting sets: parameterized combinatorial bounds and counting," in *STACS*, 2007, pp. 332–343. [Online]. Available: http://dl.acm.org/citation.cfm?id=1763424.1763465

[79] Å. Björck, *Numerical Methods for Least Squares Problems*. Philadelphia: SIAM, 1996.

[80] Q. Zhang, F. Li, and K. Yi, "Finding frequent items in probabilistic data," ser. SIGMOD '08, 2008. [Online]. Available: http://doi.acm.org/10.1145/1376616.1376698

[81] C. K.-S. Leung, M. A. F. Mateo, and D. A. Brajczuk, "A tree-based approach for frequent pattern mining from uncertain data," in *AKDDM*. Springer, 2008.

[82] C. C. Aggarwal, Y. Li, J. Wang, and J. Wang, "Frequent pattern mining with uncertain data," in *SIGKDD*. ACM, 2009, pp. 29–38.

[83] H. Toivonen, "Sampling large databases for association rules," in *VLDB*, 1996, pp. 134–145. [Online]. Available: http://dl.acm.org/citation.cfm?id=645922.673325

[84] R. Gupta, G. Fang, B. Field, M. Steinbach, and V. Kumar, "Quantitative evaluation of approximate frequent pattern mining algorithms," in *KDD*, 2008, pp. 301–309.

[85] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *VLDB*, 1994.

[86] L. Sun, R. Cheng, D. W. Cheung, and J. Cheng, "Mining uncertain data with probabilistic guarantees," in *SIGKDD*. ACM, 2010, pp. 273–282.

[87] N. Dalvi and D. Suciu, "Efficient query evaluation on probabilistic databases," *VLDBJ*, vol. 16, no. 4, pp. 523–544, 2007.

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

BIOGRAPHICAL STATEMENT

Saravanan Thirumuruganathan was born in Coimbatore, India. He received his Bachelor and Masters' degree in Computer Science and Engineering from Anna University, India, in 2005 and the University of Texas at Arlington in 2010 respectively. His current research interests include data exploration, hidden web databases, data mining, graph analytics, machine learning and crowdsourcing. He has interned at Microsoft Research (Redmond), Qatar Computing Research Institute and Yahoo! Labs (Bangalore). He was the recipient of various awards including Doctoral Dissertation Fellowship (2015), John S. Schuchman Outstanding PhD Student (2014), Outstanding Graduate Teaching Assistant (2012). His paper titled "Who Tags What? An Analysis Framework" has been selected for the Best Papers of VLDB 2012.