Rank-Based Inference Over Web Databases

Md Farhadur Rahman †, Weimo Liu †, Saravanan Thirumuruganathan †, Nan Zhang †, Gautam Das †
The George Washington University †, University of Texas at Arlington ‡
†{mdfarhadur.rahman@mavs., saravanan.thirumuruganathan@mavs., gdas}@uta.edu,
‡{wliu,nzhang10}@gwu.edu

ABSTRACT
A fundamental virtue of social media is to build virtual communities between users. As such, it is no surprise that almost all social media sites provide web interfaces for the search and/or recommendation of other users who share similar attributes, interests, etc., with results being the top-k users selected according to a ranking function. Our studies of real-world websites unveil a novel yet serious privacy leakage caused by the design of such interfaces and ranking functions. Specifically, we find that many such websites feature private attributes only visible to a user him/herself, but not to other users (and therefore will not be visible in the query answer). Nonetheless, some websites also take into account such private attributes in the design of the ranking function, understandable for improving the effectiveness of search/recommendation. While the conventional belief might be that tuple ranks alone are not enough to reveal the private attribute values, our investigation shows that this is not the case in reality.

Specifically, we define a novel problem of rank based inference, and introduce a taxonomy of the problem space according to two dimensions, (1) the type of query interfaces widely used in practice and (2) the capability of adversaries. For each subspace of the problem, we develop a novel technique which either guarantees the successful inference of private attributes, or (when such an inference is provably infeasible in the worst-case scenario) accomplishes such an inference attack for a significant portion of real-world tuples. We demonstrate the effectiveness and efficiency of our techniques through theoretical analysis and extensive experiments over real-world datasets, including successful online attacks over popular social media such as Amazon Goodreads and Catch22dating.

1. INTRODUCTION
1.1 Motivation
Given that the most valuable assets of many social media websites are contributions (e.g., tweets, blogs, posts) from users, there should be no surprise that many of them strive to provide effective web interfaces for a visitor to search for users of interest and/or for existing users to get recommendations on other users they might be interested in. If we consider the search/recommendation to be performed over a backend database that has users as tuples, then what these social media websites offer is indeed a top-k query interface which, upon given a query (either explicitly specified in search or implicitly formed by user information in recommendation), returns the top-k nearest neighboring tuples, ranked by a predetermined ranking function.

While the design of such a ranking function is an interesting area that warrants and received a lot of attention [4][7][25], the focus of this paper is to unveil a never-before-seen privacy leakage (through the top-k interface) that is caused by a seemingly innocent design of the ranking function. To understand how the privacy leakage occurs, note that many social media websites feature both public and private attributes of a user - with private ones being visible only to the user him/herself or a selected subset of users (e.g., the user’s friends), but not to anonymous visitors or other users. Some common private attributes include demographics (e.g., income), location, past posts, etc. The private nature of these attributes naturally prevents the website from displaying them in the returned results of a top-k query. For example, Facebook friend recommendation only returns public attributes such as name, photo, and selected favorites, but not private ones such as number of friends, games played, frequency of status updates, etc.

The problem here, however, is that many websites indeed include these private attributes as input to the ranking function. The purpose of doing so is, understandably, to improve the effectiveness of ranking - e.g., common sense indicates that two users with similar demographics or behavior patterns (e.g., posting with similar frequencies) are more likely to be interested in each other.

From the privacy perspective, this design might look harmless as well - after all, while a ranking function might take as input a large number of attributes, its output is merely the (relative) rank of a tuple among returned results - not even the actual ranking score! Naturally, the traditional belief here is that it is impossible to infer private attribute values from just the ranking of a returned tuple.

In our investigation of real-world social media websites, however, we found this traditional belief to be wrong. Specifically, we develop a novel technique which, by asking a carefully constructed sequence of top-k queries and observing the corresponding change in tuple ranks among the returned results, may successfully infer the value of a private attribute.

Before introducing our technical results, we would like to first illustrate the real-world impact of this privacy leakage by briefly demonstrating a very simple attack one can deploy using this technique on Amazon Goodreads, a social cataloging site with more than 30 million users. Amazon Goodreads allows users to specify their location (specified through ZIP code) as a private attribute. Meanwhile, it also provides a web interface for searching users based on their names. When there are multiple matching users, this interface sorts them according to an increasing order of the geographic distance between the location of the querying user and users in the returned result.

Figure 1 illustrates how the rank of a tuple in the search results of Amazon Goodreads can reveal the ZIP code of a user even when the user sets it as private. Suppose our goal here (as an adversary) is to infer the ZIP code of “Philip Johnson” with a blue icon in Figure 1. In order to do so, we create two accounts u1 and u2 where u1 has any arbitrary name and u2 is named “Philip Johnson”. At the beginning, we set the ZIP code of u1 and u2 to 10035 (near New York, NY) and 20052 (near Washington DC), respectively, and search for “Philip Johnson” while signing in as u1. As shown on the left hand side of Figure 1 our own u2 shows up at No. 2 (i.e., the one with white-background icon), while the victim “Philip Johnson” is ranked No. 3. From this result we can infer that...
the victim “Philip Johnson” must be more than 380km (i.e., distance between 10035 and 20052) away from New York - already a privacy leakage. Now consider an even more dangerous situation. Suppose that we change the ZIP codes of both $u_1$ and $u_2$ to 76019 (Arlington, TX) and issue the same query (while signing in as $u_1$) again. This time, as shown on the right hand side of Figure 1, the victim tuple becomes top-ranked while our $u_2$ is ranked second. One can see from this query answer that we are now certain that the location of victim “Philip Johnson” must be in 76019 - i.e., the victim tuple’s rank reveals its location.

1.2 Novel Problem: Rank-Based Inference

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

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

Given a top-k query $q$, the database computes the distance $s(t|q)$ between $q$ and each tuple $t$ in the database using a predetermined ranking function, and returns the $k$ tuples with the smallest $s(t|q)$. Of course, only the $m$ public attributes are displayed on the return interface. In this paper, we consider a broad class of ranking functions, subject only to two conditions, monotonicity and additivity, which we shall define in §2 and demonstrate that they hold for almost all ranking functions used in the real-world.

The objective of an adversary is to compromise the privacy of a pre-determined victim tuple $v$. Of course, the adversary can readily acquire the public attributes of $v$. Thus, the technical challenge for the adversary is to unveil the private attribute values, e.g., $v[B_1]$, by issuing a small number of queries through the web interface. Note that an important goal for the adversary is to keep the number of queries as small as possible, because almost all websites enforce a limit on the number of queries one can issue through the web interface for a given time period (e.g., from one IP address or one user account each day), in order to prevent overburdening its backend database or to thwart third-party crawling of its contents.

To the best of our knowledge, the above problem of inferring sensitive data from the ranking of a tuple is very novel. While top-k querying has been extensively studied by the database community [11,13,25], much of the efforts were focused on (1) developing techniques to answer such queries efficiently [24,27], and (2) designing proper distance/ranking functions for various applications [6,8,10,23]. There have been prior work on data privacy in the general area of query inferencing [8,16,19], but most focus was on learning individual values from aggregates such as SUM, MIN, MAX, etc. We discuss related work in more details in §10.

1.3 Overview of Technical Results

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

Consider the first dimension. We distinguish between interfaces which only support “point queries” (i.e., a single value must be specified for each attribute in the query), and those that also support “IN queries” (i.e., where a subset/range of values can be specified for an attribute). For the second dimension, we distinguish between two types of adversaries: (1) those who are “query-only” (Q-only adversaries) - i.e., they only issue queries and observe query answers, but never tamper with (e.g., insert fake tuples into) the database; and (2) adversaries who “query-and-insert” (Q&I-adversaries), i.e., they only issue queries but also insert fake tuples into the database (e.g., by registering for fictitious user accounts on a social media website). As we shall further elaborate in §3, while some web databases have no restriction on the registration of new accounts (i.e., anyone can insert tuples into the database - Amazon Goodreads belongs to this category), others like Catch22Dating makes it difficult for users to create fictitious accounts by enforcing the authentication of real-world identities. In the latter case, most adversaries would be Q-only, while only those who have the resources to acquire multiple real-world identities can become Q&I.

| Table 1: Feasibility, Worst- and Practical Query Cost |
|-------------------------------------------------------|
| **Feasibility** | **Q&I-Point** | **Q-Point** | **Q&I-IN** | **Q-IN** |
| Feasibility     | Yes          | Maybe       | Yes        | Maybe       |
| Worst-case (Bool) | $\Omega(2^{m'})$ | N/A         | $\Omega(2^{m'})$ | N/A         |
| In Practice     | High         | Highest     | Lowest     | Low         |

We have carefully investigated the four problem subspaces arising out of this taxonomy, and developed four novel attacks: Q&I-Point, Q-Point, Q&I-IN, and Q-IN. The fundamental ideas behind these attacks include two critical reductions: One establishes the equivalency between compromising $v[B_1]$ and finding two queries which differ only on $B_1$, yet returns $v$ at different ranks - this reduction holds for all four cases. The second reduction further reduces the problem to finding one query which returns $v$ - this reduction only holds for Q&I-adversaries. The differences on the applicability of these reductions lead to different feasibility results for the attack, as illustrated in Table 1. Specifically, we find that while Q&I adversaries are always able to accomplish the attack, there are cases where Q-only ones will fail. In terms of query cost, while the worst-case cost for even Q&I adversaries can be exponential, the query cost in practice is very reasonable - and can be significantly reduced when IN queries are available, even though IN has no impact on
the (theoretical) worst-case query cost.

Our technical results also include an extensive set of experiments over real-world datasets, including successful online attacks over popular services such as Amazon Goodreads and Catch22Dating, which we shall further demonstrate in [7].

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

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

2. PRELIMINARIES

2.1 Model of Web Databases

As discussed in the introduction, many web databases store both public and private attributes of a user. Consider an n-tuple (i.e., n-user) database D with a total of \( m + m' \) attributes, including \( m \) public ones \( A_1, \ldots, A_m \) and \( m' \) private ones \( B_1, \ldots, B_{m'} \). Let \( V_i^A \) and \( V_i^{B_i} \) be the attribute domain (i.e., set of all attribute values) for \( A_i \) and \( B_i \), respectively. We use \( t[A_i] \) and \( t[B_i] \) to denote the value of a tuple \( t \in D \) on attributes \( A_i \) and \( B_i \), respectively. For the purpose of this paper, we assume there is no duplicate tuple - i.e., every tuple has a unique value combination for the \( m + m' \) attributes.

Recall from the introduction that the database allows top-\( k \) queries where \( k \) is a small number such as 10 or 50. Given a supported query \( q \) defined below, the database computes the ranking function \( s(t(q)) \) for each tuple \( t \in D \), and selects/returns \( k \) tuples with the minimum \( s(t(q)) \). Of course, only the public attribute values, i.e., \( t[A_1], \ldots, t[A_m] \), will be returned for each of the \( k \) tuples.

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

In this paper, we consider two types of predicates that can be specified on an attribute: point and IN. A point predicate assigns a single value in the domain, i.e., \( q[A_i] \in V_i^A \), while an IN predicate assigns a subset of values, i.e., \( q[A_i] \subseteq V_i^A \). Consider a dating website as an example. While gender is often specified as a point predicate (i.e., male or female), interests and age can be considered IN ones (i.e., find users who most closely match the interest set [reading, travel, cycling, cooking] or age range [25, 30]).

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

Most web databases enforce certain query-rate limits, i.e., limits on the number of queries one can issue (e.g., from an IP address or a user account) per time period (e.g., each day), in order to prevent overburdening of the backend database and/or third-party crawling of its contents. Hence an adversary must aim to minimize the query cost of a rank-based inference attack, as otherwise it would have to acquire more resources (e.g., more IP addresses, registering more accounts) in order to issue all queries required by the attack.

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

2.2 Properties of Ranking Function

There has been significant research in database ranking (e.g., [20, 25, 27]) which studies the design of ranking function \( s(t(q)) \), including in cases where the query has IN predicates (e.g., [23, 25]). While this paper aims to study generic rank-based inferences that work for a broad class of ranking functions, it is important to note that no attack will work without assuming certain properties of the ranking function. To understand why, consider a simple example where \( s(t(q)) \) is generated uniformly at random from \([1, n]\). Since the rank of a tuple has nothing to do with the tuple’s (private) attribute values, no adversary can compromise any private information from the returned ranks. Thus, it is the objective of this subsection to define a minimum set of conditions that are satisfied by most if not all ranking functions used in practice. Specifically, we consider monotonicity and additivity, respectively as follows.

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

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

Additivity Condition: Intuitively, the additivity condition states that, if two tuples share the same value on \( A_i \) (resp. \( B_j \)), then changing both to another (still common) value should not change the relative rank of these two tuples in any query answer. More formally, we have the following definition:

\[ \text{Additivity}: \forall q \text{ and } t, t' \in D, \text{ if } t[A_i] = t'[A_i] \text{ (resp. } t[B_j] = t'[B_j] \text{), then there must be } s(t(q)) - s(t'[q]) - (s(h(t)[q]) - s(h(t')[q])) \geq 0, \text{ where } h : D \rightarrow D \text{ is a function that changes the value of } A_i \text{ (resp. } B_j \text{) of the input tuple to a fixed value}. \]

One can see that both monotonicity and additivity are commonsense conditions that should be reasonably expected of a ranking
function. Our studies of real-world web databases (in [3] verified this observation, as all websites considered satisfy both conditions.

3. PROBLEM SPACE

In this section, we define the novel rank-based inference problem studied in the paper. Specifically, we start with defining the objectives of an adversary. Then, we partition the entire problem space into four quadrants along two dimensions: the type of queries supported (i.e., point or IN), and the type of operations an adversary can perform (i.e., whether the aforementioned tuple insertion constraint applies). Finally, we introduce the problem definition and a running example used throughout the paper.

3.1 Adversary Model

The objective of an adversary is two-fold: compromising privacy and minimizing query cost. Privacy-wise, an adversary aims to compromise a private attribute of a victim tuple \( v \). Without loss of generality, we assume that the adversary aims to compromise the value of \( v[B_1] \) based on prior knowledge of all public attributes of \( v \), i.e., \( v[A_1], \ldots, v[A_m] \). An adversary has no prior knowledge of the ranking function other than the fact that it satisfies the monotonicity and additivity conditions defined above.

Given the query-rate limitation discussed in [4], an important goal of the adversary is to minimize the query cost for compromising \( v[B_1] \), as otherwise the website-enforced limit on the number of queries from each user (e.g., IP-address) may stop the attack from being completed. To this end, it is important to note that our key efficiency measure here is the number of queries issued to the web database - while other measures such as local (CPU or I/O) processing overhead are all secondary.

3.2 Two Dimensions

The first dimension we use for partitioning the problem space is the type of queries supported. There are two different cases: (1) Point-Query Interface which requires a point predicate defined in [4] to be specified for every attribute. An example here is the friend recommendation offered by many social media websites - each user has to complete his/her own profile to enable the feature, essentially requiring the user to specify a point predicate on every public and private attribute. (2) IN-Query Interface which supports IN queries over all attributes. Clearly, here a user can choose “do not care” for an attribute by assigning its entire value domain to the IN condition. Since point queries are special cases of IN, all queries supported by the point-query interface are also supported here.

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

- **Query-only (Q-only) Adversary** can query the web database but cannot change it. This is the case when the website enforces the tuple insertion constraint (see [4]).
- **Query-and-Insert (Q&I) Adversary** can not only issue queries but also insert tuples to the database. It can also update or delete any tuple it inserted. These adversaries exist for websites which do not enforce the tuple insertion constraint.

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

3.3 Problem Definition

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

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

\[
\delta(q_1(D \cup T_1), q_2(D \cup T_2), \ldots, q_k(D \cup T_k)) = v[B_1].
\]

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

To better illustrate our fundamental ideas and to significantly simplify the notations, in the theoretical discussions in this paper, we focus on a Boolean database with \( k = 1 \) (note that \( k = 1 \) is actually a conservative worst-case assumption for the attack design). We do, however, note that the experimental results section focuses exclusively on real-world datasets and websites (which, naturally, are categorical).

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

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

where \( w_i^A, w_i^B \in \{0, 1\} \) are the ranking weight for attribute \( A_i \) and \( B_i \), respectively. Note that due to the monotonicity condition, there must be \( w_i^A > 0 \) and \( w_i^B \) for all \( i \). We would like to note, once again, that the adversary has no knowledge of the ranking function design whatsoever (other than its monotonicity and additivity). This linear ranking function based running example merely provides a concrete basis for the analysis of attack performance in practice.

4. POINT QUERY INTERFACE

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

4.1 Equivalence with Finding Two Queries

The focus of this subsection is to show that, for the running example of compromising \( v[B_1] \) from a Boolean database with \( k = 1 \), the problem is equivalent with finding two queries \( q_1 \) and \( q_2 \) that share the same predicate on all attributes but \( B_1 \), yet one returns the victim tuple \( v \) while the other does not - i.e.,

\[
\forall t \in D \text{ where } t \neq v, s(t|q_1) > s(v|q_1),
\]

\[
\exists t \in D \text{ with } t \neq v \text{ such that } s(t|q_2) < s(v|q_2).
\]

The sufficiency of constructing \( q_1, q_2 \) is straightforward: Due to the monotonicity and additivity conditions, we can infer from the above conditions that the value of victim tuple \( v \) on \( B_1 \) must be the
The queries an adversary issues before inferring the value of $v_{[B_1]}$ be $\epsilon_1, \ldots, \epsilon_e$. Note that in order for the inference to hold, the following condition must be true: if we change (i.e., flip) the value of $v_{[B_1]}$, then at least one of the queries $\epsilon_1, \ldots, \epsilon_e$ must have a different answer. To understand why, note that if all query answers the adversary received remain unchanged with a different value of $v_{[B_1]}$, then there is no way for the adversary to always infer $v_{[B_1]}$ correctly from the query answers, because any deterministic algorithm that takes the answers to $\epsilon_1, \ldots, \epsilon_e$ as input will output the same value no matter whether $v_{[B_1]} = 0$ or 1.

Without loss of generality, let $\epsilon_1$ be the point query that returns different answers when $v_{[B_1]} = 0$ and 1. Since the only difference between the two cases is the value of $v$, one can see that the change of answer to $\epsilon_1$ must be on whether it returns $v$ or not. Let $\epsilon_1'$ be the query that shares the exact same value as $\epsilon_1$ on all attributes but $B_1$. Note that, if by changing $v_{[B_1]}$, the answer to $\epsilon_1$ changes from returning $v$ to not returning it, then exactly one of $\epsilon_1$ and $\epsilon_1'$ must return $v$, while the other does not. In other words, by simply testing every query that has been issued, the adversary can find $\epsilon_1$ and $\epsilon_1'$ to be $q_1$ and $q_2$ that satisfy the above criteria.

### 4.2 Q&I adversary

We develop Algorithm Q&I-Point in this subsection. Specifically, we start with a somewhat surprising finding - for a Q&I adversary, as long as it learns one query which returns the victim tuple $v$, then it can always successfully compromise $v_{[B_1]}$ after issuing only $O(m^2)$ queries, where $m'$ is the number of private attributes. After explaining this finding, we present Algorithm Q&I-Point and analyze its worst- and average-case query costs.

#### 4.2.1 Reduction to finding one query that returns $v$

We first show that, when an adversary has the ability to insert tuples, the problem of finding $q_1, q_2$ becomes equivalent with finding one query that returns $v$ subject to a $O(m^2)$ difference on query cost. This time, the necessity is obvious - since $q_1$ has to return $v$, the construction of $q_1, q_2$ requires finding at least one query that returns $v$. To prove sufficiency, we essentially have to develop an algorithm that, based on the input of one query $q$ which returns $v$, constructs $q_1$ and $q_2$ which satisfy both conditions outlined above. We describe such an algorithm, namely Algorithm Insert-Tuple-Search as depicted in Algorithm 1 as follows.

Without loss of generality, we assume the input query $q$ to have value 0 on all attributes. We start by inserting into the database a tuple $t$ that has $t[A_i] = 0$ for all attributes $i \in [0, n]$. Then, we issue (in order) $m' + 1$ queries $q_0, \ldots, q_{m'}$ defined as follows: all these $m'$ queries have public attributes $A_i = 0$ ($i \in [1, m]$). In addition, $\forall h$, query $q_h$ has $B_i = 1$ if $i \leq h$ and $B_i = 0$ otherwise (i.e., if $i > h$). One can see that, at the two extremes, $q_0 = q$ while $q_{m'}$ has the same value combination as $t$. The following example demonstrates the values of $q_0, \ldots, q_3$ when $m = 2$ and $m' = 3$.

| $A_1$ | $A_2$ | $B_1$ | $B_2$ | $B_3$ |
|-------|-------|-------|-------|-------|
| $q_0$ | 0     | 0     | 0     | 0     |
| $q_1$ | 0     | 1     | 0     | 0     |
| $q_2$ | 0     | 0     | 1     | 1     |
| $q_3$ | 0     | 0     | 1     | 1     |

and then accomplish the attack by inferring $v_{[B_2]} = t[B_2]$. Second, every pair of adjacent queries in the sequence differ exactly one attribute in each q. $q_0$ and $q_0+1$ differs on $B_{h+1}$. These two observations immediately leads to the conclusion that there must exist a pair of adjacent queries $q_0$ and $q_0+1$ such that while $q_0$ returns $v$, $q_0+1$ does not. In other words, we have found two queries which satisfy Condition 3 - on which we can safely infer that $v_{[B_{h+1}]} = 0$ i.e., one private attribute of the victim tuple $v$ is now compromised.

Of course, compromising $v_{[B_{h+1}]}$ does not yet achieve the adversarial objective of compromising $v_{[B_1]}$. Nonetheless, note that once $v_{[B_{h+1}]}$ is compromised, the adversary can essentially consider it a public attribute. In other words, the adversary can repeat the above process, with the only difference that this time $t$ (and correspondingly every query in the query sequence) has $B_{h+1} = 0$. Again, by issuing at most $m'$ queries (note that the sequence is now $t$ query shorter), the adversary can compromise at least one other private attribute. This process can be repeated until all private attributes of $v$ are compromised - costing a total of $O(m'^2)$ queries. Considering that most real-world websites have only a few private attributes, this result indicates that, as long as an adversary can learn one query which returns the victim tuple $v$, the rest of the privacy compromising process can be done very efficiently.

#### 4.2.2 Algorithm Q&I-Point

Algorithm 2 depicts the pseudocode for Q&I-Point, our algorithm for a Q&I-adversary to perform rank-based inference of $v_{[B_1]}$ over a point query interface. The algorithm consists of two steps, the first of which finds a query $q$ that returns $v$ while the second calls upon the aforementioned Insert-Tuple-Search to compromise $v_{[B_1]}$. The design of the first step is straightforward - we randomly generate and issue a query $q$ with $q[A_i] = v[A_i]$ for all $i \in [1, m]$ and each $q[B_j]$ ($j \in [1, m']$) drawn i.i.d uniformly at random from $\{0, 1\}$ - and repeat this process until finding $q$ that returns $v$. One can see that the attack is guaranteed to be successful, with a worst-case query cost of $O(2m^2)$ - which, as we shall show below, matches the theoretical lower bound of $\Omega(2m^2)$.
In addition, there exists a database \( D \) such that no Q&I-adversary can find a query returning \( v \) without incurring a query cost of \( \Omega(2^m) \).

**Theorem 1.** Given any ranking function and victim tuple \( v \), there exists a database \( D \) such that no Q&I-adversary can find a query returning \( v \) without incurring a query cost of \( \Omega(2^m) \).

**Proof.** Let there be \( m' \) tuples in the database, \( v_1, \ldots, v_{m'} \in D \), such that \( v_i \) shares the same value as \( v \) on all attributes but \( B_i \). Due to the existence of \( v_1, \ldots, v_{m'} \in D \), any query \( q \) which differs from \( v \) on at least one attribute will not return \( v \) - because, according to the monotonicity condition, there must exist at least one tuple in \( v_1, \ldots, v_{m'} \) with a smaller distance from \( q \). Since the adversary was given the prior knowledge of \( v[A_1], \ldots, v[A_m] \) but no other information about the private attribute values, the optimal adversarial strategy is to issue queries \( q \) with \( q[A_i] = v[A_i] \) for all \( i \in [1, m] \) and \( q[B_i] \) chosen uniformly at random from \{0, 1\}. One can see that, in this worst-case scenario, the expected query cost required for finding a query that returns \( v \) is \( \Omega(2^{m'}) \).

To further illustrate the various factors that impact the query cost, we consider an example of how Q&I-Point performs over the linear-combination ranking function defined in \( \mathcal{F} \) and a database where each tuple is generated i.i.d. randomly according to a fixed distribution, while the victim \( v \) is chosen uniformly from the database.

**Theorem 2.** In the above scenario, the expected number of queries Q&I-Point requires for finding a query that returns \( v \) is

\[
\frac{1}{\prod_{t \in D, t \neq v} \left( \frac{1}{2} + \frac{1}{2} \cdot \text{erf} \left( \frac{\sqrt{2} \cdot d^A(v,t)}{\sqrt{\sum_{i=1}^{m} w_i^2}} \right) \right)}.
\]

where \( \text{erf}(\cdot) \) is the standard error function and \( d^A(v,t) = \frac{w_i \cdot |v[A_i] - t[A_i]| + \cdots + w_m \cdot |v[A_m] - t[A_m]|}{w_i + \cdots + w_m} \).

**Proof.** Note that \( q \) generated in the above-described random process has \( d(q,v) \) following Binomial distribution with mean \( \mu \) and variance \( \sigma^2 \) as follows,

\[
\mu = \frac{1}{2} \sum_{i=1}^{m'} w_i^2; \quad \sigma^2 = \frac{1}{4} \sum_{i=1}^{m'} w_i^2.
\]

In addition, \( \forall t \in D \), given \( d^A(q,t) \), the overall distance \( d(q,t) \) follows the binomial distribution with mean \( \mu_0 = d^A(q,t) + \mu \) and the same variance \( \sigma^2 \) as above. Note that since \( q \) shares the same attribute values as \( v \) on all public attributes, we have \( d^A(q,t) = d^A(v,t) \). As such, the probability for a query \( q \) with \( q[A_i] = v[A_i] \) for all \( i \in [1, m] \) and \( q[B_i] \) chosen uniformly at random from \{0, 1\} to return \( v \) is

\[
p = \prod_{t \in D, t \neq v} \left( \frac{1}{2} + \frac{1}{2} \cdot \text{erf} \left( \frac{\sqrt{2} \cdot d^A(v,t)}{\sqrt{\sum_{i=1}^{m} w_i^2}} \right) \right).
\]
search over the tree structure depicted in Figure 2. In the tree, each node is corresponding to a revision of \( q \), and a node at Level \( i \) (the root is at Level 0) has \( i \) attribute values differing from \( q \), with the differing attributes noted on the path from the node to the root. For example, the bottom-left corner node in the tree is corresponding to a query that differs from \( q \) on two attributes: \( A_1 \) and \( A_2 \).

During the breadth-first search process, for each node \( q' \) we encounter, we issue \( f_0(q') \) and \( f_1(q') \) and determine if exactly one of them returns \( v \) - if so, the attack is accomplished. Note that the enumeration can be made more efficient with a pruning-based optimization: If for a node \( q' \), neither \( f_0(q') \) nor \( f_1(q') \) returns \( v \), then we can cut off from the search all nodes in the subtree of \( q' \) that only contains attributes in \( A_1, \ldots, A_m \) (beyond those in \( q' \)) - as obviously no node in this category could possibly return \( v \). Algorithm 3 summarizes the pseudocode for Algorithm Q-Point.

**Performance Analysis:** One can see from the design of Q-Point that it enumerates all possible pairs of queries \( q_1, q_2 \) supported by the point-query interface that differ only on \( B_1 \). Thus, due to the equivalence proof in \([4, 1]\) Q-Point always accomplishes the attack as long as such an attack is at all feasible over the point-query interface. Nonetheless, the query complexity of Q-Point is \( O(2^{m+m'}) \) as there are \( 2^{m+m'} - 1 \) nodes in the tree - higher than Q&I-Point.

For the linear ranking function in the running example, we have the following results for Q-Point:

**Theorem 4.** Among all queries \( q \) which return \( v \), the expected ratio (taken over the randomness of \( D \)) which, upon flipping the value of \( q[B_1] \), returns another tuple in the database is at least

\[
1 - \prod_{i \in D, i \neq v} \left( 1 + \frac{\text{erf} \left( \frac{\sqrt{2} \cdot d^A(v, t) - w_i}{\sqrt{\sum_{i=1}^m w_i^2}} \right)}{1 + \text{erf} \left( \frac{\sqrt{2} \cdot d^A(v, t)}{\sqrt{\sum_{i=1}^m w_i^2}} \right)} \right),
\]

Proof. Following the results from the proof of Theorem 2, the expected ratio of point queries \( q \) which has \( q[A_i] = v[A_i] \) for all \( i \in [1, m] \) and returns \( v \) is

\[
p = \prod_{i \in D, i \neq v} \left( \frac{1}{2} + \frac{1}{2} \cdot \text{erf} \left( \frac{\sqrt{2} \cdot d^A(v, t) - w_i}{\sqrt{\sum_{i=1}^m w_i^2}} \right) \right),
\]

because \( d^A(v, t) \) is the extra distance a tuple \( t \neq v \) has compared with \( v \) - and such a distance has to be “covered” by the private attributes in order for \( t \) to be returned. A key observation here is that, the flip of \( q[B_1] \) changes the distance by at most \( w'_i \). Thus, in order for a tuple \( t \) to be returned after the flip, the private attributes of \( t \) still have to “cover” a distance of at least \( d^A(v, t) - w'_i \). In other words, the expected ratio of point queries \( q \) which (1) has \( q[A_i] = v[A_i] \) for all \( i \in [1, m] \), (2) returns \( v \), and (3) still returns \( v \) after the flip of \( q[B_1] \) is at most

\[
p' \leq \prod_{i \in D, i \neq v} \left( \frac{1}{2} + \frac{1}{2} \cdot \text{erf} \left( \frac{\sqrt{2} \cdot (d^A(v, t) - w'_i)}{\sqrt{\sum_{i=1}^m w_i^2}} \right) \right).
\]

Thus, among all point queries \( q \) which have \( q[A_i] = v[A_i] \) for all \( i \in [1, m] \) and return \( v \), the ratio that, upon flipping the value of \( q[B_1] \), return another tuple in the database is at least

\[
1 - p' / p = 1 - \prod_{i \in D, i \neq v} \left( 1 + \frac{\text{erf} \left( \frac{\sqrt{2} \cdot (d^A(v, t) - w'_i)}{\sqrt{\sum_{i=1}^m w_i^2}} \right)}{1 + \text{erf} \left( \frac{\sqrt{2} \cdot d^A(v, t)}{\sqrt{\sum_{i=1}^m w_i^2}} \right)} \right).
\]

One can see that if a point query \( q \) has \( q[A_i] \neq v[A_i] \) for certain \( i \in [1, m] \), then this ratio must be even higher because of the now smaller distance \( d^A(v, t) \) a tuple \( t \neq v \) needs to “cover” with the private attributes. Thus, \([11]\) is indeed a lower bound on the expected ratio for all point queries which return \( v \).

An interesting observation from the theorem is that, exact opposite to the Q&I-Point case, now the larger \( w'_i \) is, the easier it is for a Q-only adversary to launch the attack (i.e., the larger this ratio will be). On the other hand, the ratio also increases with a larger database size \( |D| \) and a smaller weight on other private attributes \( w'_i \) (as \( \text{erf}(x) \) has a larger derivative when \( x \) is close to 0). We shall verify these observations in the experimental results in \([7]\).

**Algorithm 3 Q-Point**

1. **Input:** \( v \)  \[ Output:** \( v[B_1] \)
2. Identify query \( q \) that returns \( v \)
3. Construct enumeration tree \( T_q \) for \( q \)
4. \( i \leftarrow 0 \)  /\ Level 0 corresponds to root
5. while \( i < m + m' \)
   6. for each query node \( q' \) in level \( i \) of \( T_q \)
      7. \( f_0(q') = f_1(q') = q' \); \( f_0(q')[B_1] = 0 \); \( f_1(q')[B_1] = 1 \)
     8. if \( f_0(q') \) and \( f_1(q') \) didn’t return \( v \) then prune subtree(\( q' \))
     9. if only \( f_0(q') \) (resp. \( f_1(q') \)) return \( v \) then return \( f_0(q')[B_1] \) (resp. \( f_1(q')[B_1] \))
10. end for
11. \( i \leftarrow i + 1 \)
12. end while
13. return failure

5. **IN QUERY INTERFACE**

5.1 **Q&I Adversary**

For Q&I adversaries, the feasibility of rank-based inference attack is established for point-query interface in \([4, 1]\). Since point-query interface is a special case of IN, the attack feasibility here is already established. Thus, our focus here is to study how the additional power of IN queries further empowers Q&I adversaries.

Recall from \([4, 1]\) that, for Q&I adversaries, rank-based inference can be reduced to identifying one query which returns victim \( v \). In the following discussions, we shall first show that, despite of the now larger space of queries, the reduction still holds. Then, we develop Algorithm Q&I-IN, the attack algorithm for Q&I-adversary over IN query interfaces.

5.1.1 **Reduction to finding one query that returns \( v \)**

We start by showing that, so long as a Q&I adversary can find one IN query \( q \) that returns the victim \( v \), it can always accomplish the attack fairly efficiently within \( O(m^2) \) queries. To enable this reduction in the IN case, the only difference from point-query case (\([12, 1]\)) is that now the input \( q \) might have ranges like \( \{0, 1\} \) specified as predicates on \( B_1 \), instead of a single value as in the point-query case (which we denoted as 0). Fortunately, this change does
not affect the reduction construction and correctness at all - we still “flip” one private attribute at a time to construct the query sequence $q_0, \ldots, q_m$, and “flip” all private attributes to construct $\bar{t}$ - just this time “flipping” is from the original value or range specified in $q$ to any (individual) value that is different. One can see that everything else in the algorithm remains the same - if we find that by changing $B_i = \{0, 1\}$ to $B_i = 0$, the query now returns $\bar{t}$ instead of $t$, then we can safely conclude that $v[B_i] = 1$.

5.1.2 Algorithm Q&I-IN

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

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

The first observation from this example is that an attack over a point-query interface actually requires an extremely large number of queries. Specifically, note from Theorem 2 that, for a given $m$, the query cost can be made arbitrarily large with an increasing $m'$.

On the other hand, if IN queries are available, the attack query cost - more specifically, the number of queries required to find one query returning $v$ - is exactly 1 because an IN query $q$ with $A_i = v[A_i]$ for $i \in [1, m]$ and $B_j = \{0, 1\}$ for $j \in [1, m']$ always returns $v$.

One can see from the example that the usage of IN queries significantly reduces the attack query cost because of a simple reason: the ability for an adversary to eliminate all private attributes from consideration makes it much easier for the adversary to unveil the victim tuple from the database (through the query interface), so that the adversary can compromise the private attributes one at a time. More intuitively, IN queries enable an adversary to “divide-and-conquer” the private attributes, instead of having to get lucky and guess multiple private attribute values at once (e.g., in order to have the victim tuple $v$ returned in a point-query interface).

Design of Q&I-IN: Algorithm 4 depicts the pseudocode for Algorithm Q&I-IN, which enables a Q&I-adversary to launch our rank-based inference attack on $v[B_i]$ over an IN query interface. With the algorithm, we start with a query $q$ which has $q[A_i] = v[A_i]$ for all $i \in [1, m]$ and $q[B_j] = \{0, 1\}$ for all $j \in [1, m']$. Then, if $q$ does not return $v$, we gradually replace predicates on $B_i$ with point predicates (i.e., $B_i = 0$ or $B_i = 1$). Specifically, we perform what is essentially a breadth-first search process which enumerates all value combinations for $\{B_1, B_2, \ldots, B_{m'}\}$. In order words, the queries we issue are $B_1 = 0$, $B_1 = 1$, $B_2 = 0$, $B_2 = 1$ AND $B_2 = 0$, $B_1 = 1$ AND $B_2 = 1$, $\ldots$, where each query also includes $q[A_i] = v[A_i]$ for all $i \in [1, m]$; and $q[B_j] = \{0, 1\}$ for all unspecified $B_j$. When we find a query that returns $v$, we launch the binary search process in Algorithm Insert-Tuple-Search to complete the attack of $v[B_i]$.

One can see from the algorithm design that, just like in the point-query case, we guarantee a successful attack. The worst-case query cost for Q&I-IN, just like Q&I-Point, is $O(2^{m'})$. As we shall demonstrate in the following worst-case analysis, this query cost cannot be improved beyond a constant factor.

5.1.3 Query cost analysis

Algorithm 4 Q&I-IN

1: **Input:** $v$ **Output:** $v[B_i]$
2: Initialize starting query $q: q[A_i] = v[A_i] \forall i \in [1, m]$ and $q[B_j] = \{0, 1\} \forall j \in [1, m']$
3: Iteratively convert $q$ to a point query till it returns $v$
4: $q' \leftarrow$ Insert-Tuple-Search$(q, v)$
5: return $q'[B_i]$ as the inferred value of $v[B_i]$

A surprising result here is that, while the availability of an IN query interface does not help a Q&I adversary at all in the worst-case scenario, it does have the potential to significantly reduce the query cost in practice - especially when the number of public attributes is large - as we shall show through the running example.

Worst-case scenario remains unchanged: To understand why, consider any worst-case database from Theorem 2 and insert to it $m'$ tuples $v_1, \ldots, v_{m'}$, such that $v_i$ shares the same public value $v$ with $v_j$ on all attributes but $B_i$. Suppose when there is a draw (i.e., $s(t_1|q) = s(t_2|q)$), any $v_i$ will be returned before $v$. A key observation here is that the adversary cannot get any additional help from IN queries. Note that as long as a query $q$ contains an IN predicate on $B_i$ ($i \in [1, m']$), it is impossible for $q$ to return the victim tuple $v$ because there must be another tuple in $v_1, \ldots, v_{m'}$ which exactly matches $q$ and therefore will be returned. Thus, the cost for Q&I-IN in this case remains the same as in Theorem 2, $\Omega(2^{m'})$.

Running Example: For the linear-combination ranking function, we have the following theorem.

**Theorem 5.** In the running example, the expected number of queries Q&I-IN requires for finding a query that returns $v = 1$ if $d(t, v) > 0$, and at most $\sum_{h=1}^{m'}((2^{h+1} - 1) \cdot (1 - (1 - p(h))^2))$ otherwise, where

$$p(h) = \prod_{t \in D, t \neq v} \left(\frac{1}{2} + \frac{1}{2} \cdot \text{erf} \left(\frac{\sqrt{2} \cdot d^h(v, t)}{h^{1/4}}\right)\right).$$

**Proof.** First, when no other tuple $t \in D$ shares the same public-attribute value-combination as $v$ (i.e., $\min_{t \in D, t \neq v} d^h(v, t) > 0$), then as we discussed in the design of Q&I-IN, only one query (with point-predicates and IN-predicates on all public and private attributes, respectively) is required. For other cases, in analogy to the proof of Theorem 2, one can see that the probability for a randomly generated query $q$ with $q[A_i] = v[A_i]$ for all $i \in [1, m]$ and $h$ point-predicate queries specified on private attributes to return $v$ is

$$p(h) = \prod_{t \in D, t \neq v} \left(\frac{1}{2} + \frac{1}{2} \cdot \text{erf} \left(\frac{\sqrt{2} \cdot d^h(v, t)}{h^{1/4}}\right)\right).$$

Since the total number of such queries is $2^h$, and the overall query cost after enumerating queries with $h$ or fewer predicates is (i.e., on $B_1, \{B_1, B_2\}, \ldots, \{B_1, \ldots, B_{m'}\}$, as specified in Q&I-IN) $2^{h+1} - 1$, the expected number of queries required by Q&I-IN is $\sum_{h=1}^{m'}((2^{h+1} - 1) \cdot (1 - (1 - p(h))^2))$. □

One can see from the theorem the substantial promise for IN queries to significantly reduce the query cost - not only the query cost can be cut to 1 when no other tuple shares the same public-attribute value-combination as $v$, but the value of $p(h) - i.e.,$ the probability for a query with $h$ point-predicates on private attributes
5.2 Q-only adversary

Just like the availability of IN queries does not help reduce the worst-case query cost for Q&I-adversaries, it cannot change the (in)feasibility result for Q-only adversaries either. Indeed, one can see from the proof of Theorem 4 that the construction there readily applies to IN-query interfaces as well - proving that Q-only adversaries cannot guarantee the access of rank-based inference even for IN query interfaces. Nonetheless, as we shall show in this subsection and in the experimental results, the availability of IN queries does help with reducing the query cost in practice, especially when the number of public attributes is large.

Algorithm 5 depicts the pseudocode for Algorithm Q-IN, which enables a Q-only adversary to launch our rank-based inference attack on \( v[B] \) over an IN query interface. With the algorithm, we start with calling Algorithm Q&I-IN to find one query \( q \) which returns \( v \). Note that, according to the design of Q&I-IN, \( q \) always has \( q[A] = v[A] \) for all \( i \in [1, m] \).

After obtaining \( q \), Algorithm Q-IN issues both \( f_0(q) \) and \( f_1(q) \) (recall the definition of \( f_0(\cdot) \) and \( f_1(\cdot) \) from \( \text{Algorithm 4} \)). Similar to the discussion in Algorithm Q-Point, one can see that at least one of the two queries must return \( v \), and the attack is already successful if only one of them does. In case both \( f_0(q) \) and \( f_1(q) \) return \( v \), we start the process of gradually removing public attributes from \( q \) (i.e., by setting \( q[A] \) to \{0, 1\}).

Specifically, we start with revising \( q \) to \( q' \) by setting \( A_1, \ldots, A_m \), respectively, to \{0, 1\} one attribute at a time. For each \( q' \), we test whether exactly one of \( f_0(q') \) and \( f_1(q') \) returns \( v \). If none of these \( m \) queries accomplishes the attack, we continue the revision by setting one more public attribute in each \( q' \) to \{0, 1\} (leading to \( \binom{m}{2} \) revisions). This process continues until we find a successful attack or exhaust all possible revisions. A pruning-based optimization here is that, if neither \( f_0(q') \) nor \( f_1(q') \) returns \( v \), then we do not need to continue the revision of \( q' \) (i.e., settings any more public attributes in it to \{0, 1\}), because no such revised query will return \( v \) anyway due to the monotonicity condition.

If we have exhausted all possible revisions of \( q \), without completing the attack, then we move back to Algorithm Q&I-IN, find another query \( q \) which returns \( v \), and attempt the revision process again. We repeat this process until exhausting all queries found by Algorithm Q&I-IN which return \( v \).

Algorithm 5 Q-IN

1: Input: \( v \) \hspace{1cm} Output: \( v[B] \)
2: \hspace{1cm} while some query \( q \) returns \( v \) do
3: \hspace{2cm} \( f_0(q) = f_1(q) = q; \hspace{0.5cm} f_0(q)[B_1] = 0; \hspace{0.5cm} f_1(q)[B_1] = 1 \)
4: \hspace{2cm} if only \( f_0(q) \) (resp. \( f_1(q) \)) returns \( v \), return 0 (resp. 1)
5: \hspace{1cm} \hspace{1cm} \( i \leftarrow 1 \)
6: \hspace{2cm} \hspace{1cm} for each \( \binom{m}{2} \) possible combination \( C \in \{A_1, \ldots, A_m\} \) do
7: \hspace{3cm} \( q' \leftarrow q; \hspace{0.5cm} q'[C] = \{0, 1\} \)
8: \hspace{3cm} if \( f_0(q') \) and \( f_1(q') \) didn’t return \( v \) then prune subtree(\( q' \))
9: \hspace{2cm} \hspace{1cm} if only \( f_0(q') \) (resp. \( f_1(q') \)) return \( v \), return 0 (resp. 1)
10: \hspace{1cm} \hspace{1cm} \( i \leftarrow i + 1 \)
12: \hspace{1cm} \hspace{1cm} end while
13: \hspace{1cm} return failure

One can see from the design of Algorithm Q-IN that, since the maximum possible number of revisions for each input \( q \), from Q&I-IN is \( \binom{m}{2} + \cdots + \binom{m}{m} = 2^m \), the maximum number of queries we might get from Q&I-IN is \( 2^m \), and the overall query cost (for finding all queries \( q \)) of Q&I-IN is \( O(2^m) \), the query complexity for Q-IN is \( O(2^{m^2} + m) \). For the running example, we have the following results for Q-IN:

\[
1 - \prod_{t \in D, t \neq v} \left( 1 + \text{erf} \left( \frac{\sqrt{2} d(v, t)}{\sqrt{\sum_{j \in B_1 \in S^t} w_{ij}^v}} \right) \right). \tag{14}
\]

The corollary follows directly from the proof of Theorem 4. Just like in the Q&I-IN case, one can observe from the theorem the substantial promise for IN queries to significantly reduce the query cost - specifically, note that the smaller \( S \) or \( S' \) is, the higher this expected ratio will be. As such, the overall query cost is likely much smaller than Q-Point - as we shall verify in the experiments.

6. DISCUSSIONS

6.1 Categorical and Numeric Attributes

Previously, we mostly focused on Boolean databases. In this subsection, we discuss the extensions to categorical and numeric attributes. Specifically, we shall start with discussing the extension to categorical attributes for each of the four problem subspaces, respectively, and then discuss at the end of this subsection a key additional issue concerning numeric attributes: If the private attribute of interest is a numeric one, can the attack be arbitrarily precise?

Point-Query Interface: We start with considering Q&I-Point. One can see from the design of Algorithm Q&I-Point that only two revisions are required for handling categorical attributes: (1) For finding a query \( q \) that returns \( v \), we should now draw the value of \( q[B] \) uniformly at random from the (categorical) domain of \( B \), denoted by \( V^B \), instead of from just \{0, 1\}. (2) For the Insert-Tuple-Search subroutine, we should now enumerate all possible values of \( B \) in the query (while keeping the other predicates of \( q \) - i.e., instead of issuing two queries \( q_1 \) and \( q_2 \), we now issue \( |V^B| \) queries with different values of \( B \). The search space for the tuple to insert also expands from \( 2^m \) queries to \( \prod_{b \in V^B} |V^b| \) - though the exact same binary search process can still be used. One can see that these changes lead to an overall worst-case query complexity of \( O(S \prod_{b \in V^B} |V^b|) \) for Q&I-Point.

We now shift our attention to Q-Point. Recall from Theorem 4 that, after calling Algorithm Q&I-Point to find one query which returns \( v \), Algorithm Q-Point performs a breadth-first search over a query tree (depicted in Figure 2). A key observation here is that, with the presence of categorical attributes, the tree structure remains exactly the same - only the definition queries corresponding to each node in the tree needs to be extended: in the Boolean case, each node is corresponding to a query that has the exact opposite values (to \( q \)) for all (and only) attributes on the path from the node to the root. In the categorical case, the node is corresponding to all queries that have different values on all (and only) attributes on the path. In other words, each node is now corresponding to not 1 but \( \prod_{i} |V_i| \) queries, where \( |V_i| \) is the domain size of each attribute from the node to the root. One can see that this change leads to an overall worst-case query complexity of \( O(S \prod_{b \in V^B} |V^b| \prod_{j=1}^{m'} |V_j^B|) \).
IN-Query Interface: The changes required for Q&I-IN and Q-IN to handle categorical attributes are similar to those for their counterparts in point-query interfaces - specifically, we replace \( \{0, 1\} \) with the corresponding attribute domain \( V^A_i \) or \( V^B_j \), and otherwise follow the same query enumeration process as in the Boolean case. Of course, this time the number of queries to enumerate is much larger, - leading to an overall query cost of \( O(\prod_{j=1}^{n'} |V^B_j|) \) and \( O(\prod_{i=1}^{m'} |V^A_i| - \prod_{j=1}^{m'} |V^B_j|) \) for Q&I-IN and Q-IN, respectively. Also as in the Boolean case, while a Q&I-adversary can always accomplish the attack, a Q-only adversary might not in the worst-case scenario.

Attack Precision for Numeric Private Attribute: In the original problem definition discussed in \( ^2 \) we consider an attack to succeed if and only if the adversary unveils the exact value of a (Boolean or categorical) private attribute. For a numeric (private) attribute, however, it becomes more complex to measure the success of an attack. Specifically, as we shall demonstrate as follows, while there are cases where an adversary can infer a numeric attribute value to an arbitrary precision, there are also cases where the precision is limited to a (small) fixed range. Nonetheless, either case still represents serious compromise of user privacy.

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

Given the wide variety of ranking functions a query interface might feature, and the fact that even a fairly wide interval on \( B_i \) (as long as it is significantly narrower than the entire domain) is usually a significant threat to privacy in practice, discussing the achievable precision for each type of interfaces is beyond the scope of this paper. Instead, we make an assumption that numeric attributes can be properly discretized (and treated as a categorical one) according to two principles: (1) the discretized range is narrow enough so each tuple has a unique value combination of all attributes, and (2) the range should be as wide as possible, so as to minimize \( |V^A_i| \) and \( |V^B_j| \), thereby minimizing the query cost of the attack.

6.2 Defense Against Rank-based Inference

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

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

Another category of defense is to adjust the assignment of public/private attributes and/or the design of ranking function. Recall from the discussions of Q&I-Point and Q-Point that the more attributes the database owner assigns to be private, and the higher weights the ranking function assigns on private attributes, the more difficult it is for an adversary to launch the attack, as the prior knowledge held by an adversary on the victim tuple (i.e., \( v[A_1], \ldots, v[A_{m_1}] \)) now plays a lesser role on determining the rank, making it harder for the adversary to efficiently locate the victim tuple.

Nonetheless, this strategy does not work as effectively on an IN-query interface. To understand why, note from the design of Q&I-IN that, as long as the public attributes are sufficient for uniquely identifying the victim tuple, a Q&I-adversary can succeed with \( O(m') \) queries no matter how much weight the ranking function places on the private attributes. In this case, the defender can choose to publicize fewer attributes (if doing so prevents an adversary from learning these attribute values for the victim tuple), or disabling IN-query predicates on certain attributes. As we discussed in \( ^2 \) the reduction of IN-query predicates may significantly delay the attack in the average-case scenario.

7. EXPERIMENTAL RESULTS

7.1 Experimental Setup

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

Real-World Datasets: We tested our algorithms locally over two real-world datasets. One is data from eHarmony (eH) \( ^{29} \), a prominent online dating service that matches users by personality traits. It consists of anonymized profile information of 500,000 real-world websites - Amazon Goodreads and Catch22Dating - and one is data from a social cataloging site where the users can connect to each other and share their experience/opinions about

Amazon Goodreads (GR): a social cataloging site where the users can connect to each other and share their experience/opinions about

\(^{1}\) http://www.eharmony.com

\(^{2}\) http://auto.yahoo.com

\(^{3}\) https://www.goodreads.com/
books. The user profile information consists of demographic information (public) and location (ZIP code) of the user which can be set to private, preventing it from being displayed to other users. We found that regardless of a user’s choice on location privacy, the ranking function used in the website’s “user search” interface always takes the user’s location into account when ranking it among other users in search results. Specifically, the system ranks each user according to its (geographic) distance from the location of the user performing the search. The search interface to find other similar users allows only a single attribute - user name.

Hence, we can consider each tuple in Goodreads database to have only two attributes: a public attribute - user name and a private attribute - zipcode. Both are categorical attributes with very large value domains. Goodreads allows free account registration - i.e., there is no tuple insertion constraint. Further, the ranking function only allows fully specified queries. Thus, we consider it to be a point-query interface with threat from Q&I-adversaries, and used our Q&I-Point algorithm to infer user locations.

**Catch22Dating:** Catch22Dating is an online dating website where users create profiles that are then matched to other users in the same site. It allows users to specify a set of public and private attributes. The public attributes are used to capture the demographic information of the user whereas the private attributes specify user’s matching preferences. It has a search interface option (dubbed “Both Perspective”) - selecting this option enables the ranking function to use the public and private attributes of all profiles in the website. While Catch22Dating has a number of private attributes, we focused on one Boolean attribute - Is it ok if your matches have been married before (henceforth referred to as Married). Catch22Dating does enforce the tuple insertion constraint (by requiring Student ID from selected universities during user registration). It also allows IN queries to be specified (e.g., one can set an attribute to be “do not care” in the query). Hence, we consider it to be an IN-query interface with threat from Q-only adversaries, and used our Q-IN algorithm to infer the value of private attribute Married.

**Algorithms:** For both real-world datasets eH and YA, we tested all the four algorithms - Q&I-Point, Q-Point, Q-IN, Q-IN - covering our entire problem space. For the online demonstration, we used a variant of the corresponding algorithm to take into account the fact that we do not know the proprietary ranking functions of GR and CD.

**Performance Measures:** For all algorithms, we measure efficiency through query cost, i.e., the number of queries required for each successful attack, which is consistent with prior work [13-14].

### 7.2 Experiments over Real-World Datasets

We start by describing the experimental results over real-world datasets eH and YA, to which we have full access. For both datasets, we set $k = 1$ unless otherwise specified. Our charts report the results for eHarmony. The equivalent charts for Yahoo! Autos can be found in §9.

**Query Cost versus $k$:** We first tested the performance of our algorithms over eHarmony dataset by investigating the query cost for different values of $k$. Figure 3 shows that query cost decreases with higher values of $k$ as expected. The query cost of our algorithms can be broadly categorized into two parts - the query cost to identify a query $q$ that returns the victim tuple and the query cost required to construct additional queries from $q$ through which the private attribute is inferred. When the value of $k$ increases, the former query cost falls dramatically. Further, the figure also shows that when IN-queries are available (Q&I-IN and Q-IN), the query cost is lower than the cases where only point queries are allowed (Q&I-Point and Q-Point), consistent with our discussions in §5.

**Query Cost versus Database Size, $m$:** Figure 4 depicts the impact of database size on query cost. As expected, the increase in database size does not have any major impact and only results in a slight increase in overall query cost. This is due to the fact that the number of queries needed to identify a randomly chosen tuple increases much more slowly than the database size.

**Query Cost versus $m$, $m'/n$:** In our next experiments, we investigate how varying the number of public and private attributes affect the query cost. The results of these experiments are shown in Figures 5 and 6. As expected, when the number of public attributes increases, the query cost drops significantly. When the number of public at-
tributes are limited, their values are not adequate to distinctly identify a random tuple. Hence, we need to resort to using randomly chosen values for the private attributes which increases query cost. However, when \( m \) increases, most tuples become uniquely identified based on their public attributes only. For a fixed \( m \), the query cost increases with increasing \( m' \) - when the public attributes are not adequate for uniquely identifying the victim tuple, our algorithms resort to issuing queries where the private attributes are chosen randomly from \([0, 1] \). However, the number of such possible queries \( 2^m' \) increases with higher \( m' \).

**Query Cost versus Ranking Weights**: In this experiment, we fixed the weight of all public attributes to 1 and varied the weights of private attributes \( w_i \) between 0.01 and 100. The results shown in Figure 7 and 8 are consistent with our theoretical results from Sections 4 and 5. When the weights over private attributes decrease, the query cost for Q&I adversaries also decreases. This is due to the fact that identifying the query \( q \) that returns the victim tuple \( v \) becomes much easier for this case. The opposite holds for Q-only adversaries where increasing the weights decreases the query cost for compromise.

**Other Experiments**: We performed additional experiments to identify the fraction of tuples in a database that could be successfully compromised using our algorithms. For this experiment, we randomly chose 100K tuples and tried to compromise them. Recall that the Q&I adversary based algorithms are always guaranteed to succeed. Figure 9 shows that the Q-only algorithms are able to compromise almost all the tuples. Even with a restrictive interface where \( k = 1 \), Q-Point algorithm could compromise more than 99% of the tuples. This value increases even further for higher, but more realistic, values of \( k \). We adapted Algorithm 1 so that it seeks to infer all \( m' = 25 \) private attributes. Figure 10 shows the result. While the overall query cost seems high, the amortized query per private attribute varies between 35 and 60. Figure 2 shows how varying the domain size of the private attribute affects the query cost. As expected, boolean attributes require the least query cost which increases with larger domain size. This is consistent with our worst case analysis.

### 7.3 Online Demonstration

We now describe the results of the online demonstrations where we sought to compromise private attributes `zipcode` and `Married` for Amazon Goodreads (GR) and Catch22Dating (CD), respectively. A detailed description of the procedure we used and its correctness can be found in [8]. Note that since we have no connection with these websites and thus do not access to the ground truth, we limit the scope of these experiments to a small-scale proof-of-concept, while leaving the comprehensive experiments to the above-described real-world datasets.

**Experimental Details**: Since Amazon Goodreads enforces no limits on account creations, we started with registering 10 fake accounts with randomly generated ZIP codes, and launched Q&I-Point over it to verify the correctness of our algorithm. Then, to enable verification on real accounts, we identified 25 “special” users at Goodreads who have their ZIP code hidden but chose to reveal their city/state (in US). We launched Q&I-Point successfully on all these users, and then verified that every ZIP code we compromised indeed belongs to its corresponding city/state revealed by the user. The average query cost, as shown in Table 2, is 560 per victim.

Catch22Dating does enforce the tuple insertion constraint, making the verification much harder. Furthermore, the website also allows NULL values on almost all attributes. As a result, our Q-IN attack might fail because no query can reveal the private attribute value - but it might also fail simply because the user specified NULL as the attribute value. One can see from Table 2 that out of the 110 users we attacked, we were able to compromise the private attribute `Married` for 58 of them. For the other 52, either the user did not specify whether he/she would like to accept matches who have been married, or Q-IN attack fails on these users. The average query cost for the success and NULL/failure cases are 62 and 668, respectively, consistent with our prior discussions that failures generally consume many more queries than the successful cases.

| #Accounts | #Success | Avg Cost (Success) | Avg Cost (Failure) |
|-----------|----------|--------------------|--------------------|
| CD        | 120      | 61                 | 60                 |
| GR        | 25       | 25                 | 560                |
|           |          |                    | N/A                |

### 8. ADDITIONAL DETAILS FOR ONLINE EXPERIMENTS

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

**Example Attack over Catch22Dating**: Catch22dating (CD) is an online dating website with millions of users. CD allows users to create profiles containing public (such as demographics) and private (such as matching preferences). CD also has a search interface where users could specify a query (based on public information only) to search for other users. CD uses a ranking function that matches the profile using both public and private information. Suppose, we wish to infer a private information (Is it ok if your matches have been married before) of a user \( v \) (with screen name Anya). We first created a fake user profile \( u \) where we specified the marital status as ‘Never married’. Under these circumstances, our results in Anya as the best matching user. Figures 1 and 2 shows the result. We then change \( u \)’s profile to specify the marital status as ‘Previously Married’. When we issue the same query (but for the modified profile), we can see that the rank of Anya has dropped. We can now plausibly infer that Anya has specified that she prefers her matches not to be married before.

**Catch22Dating Inference**: Using the notations from the technical sections, let \( v \) be the victim tuple whose private attribute value \( v[A_1] \) we seek to infer. In the context of Catch22Dating, the private boolean attribute \( B_1 \) stores the user’s response to question: Is it ok if your matches have been married before. It can take two values - No or No Preference. The public attribute most relevant to \( B_1 \) is \( A_1 \) which stores the user’s response to the question: Have you married before. It takes two values - Yes and No. We construct a random point query by using the public attributes from \( v \)’s profile and chose the values for private attributes randomly. However, we set the value for the attribute Have you married before to No. If this randomly constructed query (say \( q_1 \)) returned \( v \), then we create an alternate query \( q_2 \). \( q_2 \) is identical to \( q_1 \) on all attributes except for the value of attribute \( A_1 \) - \( q_1[A_1] = \text{No} \) (not married before) and \( q_2[A_1] = \text{Yes} \) (had married before). Now if the rank of \( v \) is lower in \( q_2 \) than in \( q_1 \) (i.e., \( d(q_2, v) > d(q_1, v) \)), the attacker can infer that the target profile \( v \) has private attribute \( B_1 \) value set to No.

**Correctness Argument**: If the target profile \( v \) has \( B_1 \) value set to No Preference, then \( d(q_1, v) = d(q_2, v) \). This is because by
setting \(v[B_i]\) to No Preference, the target profile is accepting any value of \(A_1\) in the search query. On the other hand if \(v[B_1] = \text{No}\) then \(d_1(q_1, v) < d_1(q_2, v)\). When the attacker issues a query \(q_2\) followed by \(q_1\), one of the three scenarios can arise:

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

If \(v[B_1] = \text{No Preference}\), only (1) or (2) is possible. While scenario (1) is easy to understand, scenario (2) may appear if there exists a tuple \(t\) such that \(t[B_1] = \text{No}\), \(d_1(q_1, t) < d_1(q_1, v)\) and \(d_1(q_2, t) > d_1(q_2, v)\). Scenario (3) is impossible when \(v[B_1] = \text{No Preference}\) as it is not possible to find a tuple \(t\) that has \(d_1(q_1, t) < d_1(q_1, v)\) and \(d_1(q_2, t) > d_1(q_2, v)\). So when the attacker finds that the rank of the target profile \(v\) decreases after switching from \(q_1\) to \(q_2\), he/she can correctly infer that \(v[B_1] = \text{No}\), because the only assignment \(v[B_1]\) can have other than No Preference is No.

**Goodreads Inference:** Goodreads has a single private attribute zipcode. The search interface to find other similar users allows only a single attribute - user name. When displaying the results of a search query it ranks the user profiles (who have the user name from the query) according to a proprietar function from the location of the user performing the search. Based on our observations, Goodreads seems to use some proprietary variant of zipcode-distance function. We used a publicly available distance function - but to address the uncertainty of Goodreads’ ranking function we added an error margin.

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

9. **ADDITIONAL EXPERIMENTS**

Figures 13 and 20 show the results from experiments performed over Yahoo! Autos. We set \(k = 1\) and \(m = m' = 10\) unless otherwise specified.

**Experimental Observations:** We can notice that the results of the experiments over Yahoo Autos (YA) follows trends similar to that over eHarmony. For example, Figure 14 shows that the query cost decreases with increasing value of \(k\). This is due to the fact that for larger values of \(k\), few queries are required to identify a query that returns victim tuple \(v\). Figure 14 shows that increasing the database size do not have any major impact on query cost as the number of queries needed to identify a randomly chosen tuple increases much more slowly than the database size. Figures 15, 16 describe how changing the number of public and private attributes affect the query cost. Not surprisingly, the results follow a trend similar to one observed for eHarmony. When the number of public attributes increase, the query cost drops significantly. In contrast, for a fixed \(m\), increasing \(m'\) results in higher query cost. Figure 17 shows the result of varying the weights of private attributes. Consistent with our theoretical results, when the weights over private attributes decrease, the query cost for Q&I adversaries also decreases. However, for Q-only adversaries increasing the weights decreases the query cost needed for compromise. Figure 18 shows that our algorithms could compromise an overwhelming number of tuples. Figure 19 shows that the cost of compromising a categorical attribute increases with its domain size. In other words, boolean attributes have the smallest query cost which increases for attributes larger domain size. Finally, Figure 20 shows the cost for inferring all private attributes. Note that the amortized query cost per private attribute is very small - as also observed for eHarmony dataset.

10. **RELATED WORK**

10.1 **Ranking of Tuples**

The area of ranking and top-\(k\) processing has been extensively studied as a way to identify most important tuples in the context of deterministic [25, 25], probabilistic [27] and incomplete [22] data in database systems. Our paper uses a top-\(k\) selection query model [25] where the scores are attached to the tuples. A top-\(k\) selection query produces a list of \(k\) tuples with the highest score computed possibly through a user defined function. Retrieving top-\(k\) tuples where the tuple score is a combination of scores of individual attributes has been studied in [18, 24]. There are many ways to categorize ranking functions used in databases. A ranking function \(f\) is said to be static if for a given tuple \(t\), \(f(q,t)\) is constant for all queries \(q\). A ranking function is query dependent if, for a given tuple \(t\), \(f(q,t)\) varies for different queries \(q\). A popular query dependent ranking function is that of nearest neighbor [24] where the tuples are ordered based on the distance between tuple \(t\) and the given query \(q\). Other categorizations such as monotone, generic or no ranking (such as Skyline queries) has also been studied [25].

Ranking functions are most useful when the query is under-specified and matches lot of tuples (many answers problem [7]) or when the query is over-specified and matches no tuples in the database (empty answer problem [4, 31]). The former problem is typically solved by proposing automated ranking methods that rely on work-
load or other additional information [7]. The latter is solved by iteratively relaxing the problem till we get some results [4]. Such query reformulation methods are in contrast to the ranked retrieval approach such as ours where the user or the system provide a ranking function that scores all tuples based on the input query. Recently, there has been some work [15] on estimating the rank of a tuple given a top-k static ranking interface. [34] described methods to retrieve the top-h tuples (where $h > k$) over a top-k interface. This could be construed as a rank based inference where the ranking function of the database is not known and the rank of a tuple is considered as the private attribute that must be inferred.

### 10.2 Inference Control

An inference attack [19] is said to occur when sensitive information can be inferred from non-sensitive data and metadata. The sensitive information thus inferred could be a value of a private attribute of a tuple or the value of an individual tuple from an aggregate. Prior work has studied the problem of inferring whether a tuple is present in a database [32], learning individual values from aggregates such as SUM/MIN/MAX [8], range SUM queries [9] etc. The field of inference control [2,16,19] seeks to prevent such attacks by using diverse techniques such as auditing, controlling the number of tuples that match a query or modify query responses using perturbation, distortion etc [10]. Researchers have also proposed multiple privacy preserving aggregate query processing techniques [3,17]. Recently, [28] has showed that it is possible to infer the location of a user in a Location based Social Network (LBSN) (which could be considered as a private attribute) if the ranking function returns the distance between the query and the victim tuple. However, we do not assume the availability of such information as most websites do not display the score of a tuple for a query.

### 10.3 Other Web Database Privacy Problems

[13] [14] studied the problem of estimating aggregates over a hidden web databases using sampling based approaches. [15] studied the problem of protecting sensitive aggregates from disclosed through individual tuples that are returned by search queries without affecting the usability of the hidden web databases. This is in contrast to traditional privacy scenarios that seek to protect individual information. [12] investigates potential privacy issues from disclosure of sensitive aggregates in health databases. [5][21][16] study the problem of sensitive aggregate protection in the context of frequent pattern mining.

### 11. FINAL REMARKS

In this paper, we identified an important and novel problem of rank-based inferencing over web databases. We introduced a taxonomy of the problem space into four important subspaces based on varying database interface limitations and adversarial capabilities. For each problem subspace, we developed nontrivial adversarial techniques. For each problem subspace, we developed nontrivial adversarial capabilities. For each problem subspace, we developed nontrivial adversarial capabilities.

### 12. REFERENCES

[1] Catch22dating frequently asked questions. http://www.catch22dating.com/faq. Accessed: 2014-11-10.

[2] N. R. Adam and J. C. Worthmann. Security-control methods for statistical databases: a comparative study. ACM Computing Surveys (CSUR), 21(4):515–556, 1989.

[3] R. Agrawal and R. Srikant. Privacy-preserving data mining. ACM Sigmod Record, 29(2):439–450, 2000.

[4] S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis. Automated ranking of database query results. In In CIDR. CIDR, 2003.

[5] M. Atallah, E. Bertino, A. Elmagarmid, M. Ibrahim, and V. Verykios. Disclosure limitation of sensitive rules. In KDEX. IEEE, 1999.

[6] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. ACM Transactions on Database Systems (TODS), 27(2):153–187, 2002.

[7] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic ranking of database query results. pages
888–899. VLDB Endowment, 2004.
[8] F. Chin. Security problems on inference control for sum, max, and min queries. *JACM*, 33(3):451–464, 1986.
[9] F. Y. Chin, P. Kosowski, and S. Loh. Efficient inference control for sum queries. *Theoretical Computer Science*, 32(1):77–86, 1984.
[10] F. Y. Chin and G. Ozsoyoglu. Auditing and inference control in statistical databases. *TSE*, 8(6):574–582, 1982.
[11] D. R. Cox. *Principles of statistical inference*. Cambridge University Press, 2006.
[12] G. Das and N. Zhang. Privacy risks in health databases from aggregate disclosure. In *PETRA*, page 74. ACM, 2009.
[13] A. Dasgupta, G. Das, and H. Mannila. A random walk approach to sampling hidden databases. In *SIGMOD*, 2007.
[14] A. Dasgupta, X. Jin, B. Jewell, N. Zhang, and G. Das. Unbiased estimation of size and other aggregates over hidden web databases. In *SIGMOD*, pages 855–866. ACM, 2010.
[15] A. Dasgupta, N. Zhang, G. Das, and S. Chaudhuri. Privacy preservation of aggregates in hidden databases: why and how? In *SIGMOD*, pages 153–164. ACM, 2009.
[16] J. Domingo-Ferrer. A survey of inference control methods for privacy-preserving data mining. In *Privacy-preserving data mining*, pages 53–80. Springer, 2008.
[17] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography*, pages 265–284. Springer, 2006.
[18] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *JCSS*, 66(4):614–656, 2003.
[19] C. Farkas and S. Jajodia. The inference problem: a survey. *ACM SIGKDD Explorations Newsletter*, 4(2):6–11, 2002.
[20] X. Geng, T.-Y. Liu, T. Qin, A. Arnold, H. Li, and H.-Y. Shum. Query dependent ranking using k-nearest neighbor. In *SIGIR*, pages 115–122. ACM, 2008.
[21] A. Gkoulalas-Divanis and V. S. Verykios. An integer programming approach for frequent itemset hiding. In *CIKM*, 2006.
[22] T. J. Green and V. Tannen. Models for incomplete and probabilistic information. In *Current Trends in Database Technology–EDBT* 2006, pages 278–296. Springer, 2006.
[23] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. *Range queries in OLAP data cubes*, volume 26. 1997.
[24] V. Hristidis, N. Koudas, and Y. Papakonstantinou. Prefer: A system for the efficient execution of multi-parametric ranked queries. In *ACM SIGMOD Record*, pages 259–270, 2001.
[25] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 40(4):11, 2008.
[26] W. Kießling. Foundations of preferences in database systems. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 311–322. VLDB Endowment, 2002.
[27] J. Li, B. Saha, and A. Deshpande. A unified approach to ranking in probabilistic databases. *VLDB*, 2009.
[28] M. Li, H. Zhu, Z. Gao, S. Chen, L. Yu, S. Hu, and K. Ren. All your location are belong to us: Breaking mobile social networks for automated user location tracking. MobiHoc ’14, 2014.
[29] B. McFee and G. R. Lanckriet. Metric learning to rank. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 775–782, 2010.
[30] A. Motro. Vague: A user interface to relational databases that permits vague queries. *ACM Transactions on Information Systems (TOIS)*, 6(3):187–234, 1988.
[31] D. Mottin, A. Marascu, S. B. Roy, G. Das, T. Palpanas, and Y. Velegrakis. A probabilistic optimization framework for the empty-answer problem. *VLDB*, 6(4):1762–1773, 2013.
[32] M. E. Nergiz, M. Atzori, and C. Clifton. Hiding the presence of individuals from shared databases. In *SIGMOD*, 2007.
[33] Y. Rui, T. S. Huang, and S. Mehrotra. Content-based image retrieval with relevance feedback in mars. In *Image Processing, 1997. Proceedings., International Conference on*, volume 2, pages 815–818. IEEE, 1997.
[34] S. Thirumuruganathan, N. Zhang, and G. Das. Breaking the top-k barrier of hidden web databases. In *ICDE*. IEEE, 2013.
[35] S. Thirumuruganathan, N. Zhang, and G. Das. Rank discovery from web databases. *VLDB*, 2013.
[36] V. S. Verykios, A. K. Elmagarmid, E. Bertino, Y. Saygin, and E. Dasseni. Association rule hiding. *TKDE*, 2004.