Aggregate Estimations over Location Based Services

Weimo Liu[†], Md Farhadur Rahman[‡], Saravanan Thirumuruganathan[‡], Nan Zhang[†], Gautam Das[‡]
The George Washington University[†], University of Texas at Arlington[‡]

ABSTRACT

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

1. INTRODUCTION

1.1 LBS with a kNN Interface

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

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

In addition, there are important differences among the services based on the type of information that is returned along with the k tuples. Some services (e.g., Google maps) return the locations (i.e., the x and y coordinates) of the k returned tuples. We refer to such services as *Location-Returned LBS* (LR-LBS). Other services (e.g., WeChat, Sina Weibo) return a ranked list of k nearest tuples, but suppress the location of each tuple, returning only the tuple ID and perhaps some other attributes (such as tuple name). We refer to such services as *Location-Not-Returned LBS* (LNR-LBS).

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

1.2 Aggregate Estimations

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

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

1.3 Outline of Technical Results

Results over LR-LBS Interfaces: We first describe our results over LR-LBS interfaces. Like [10], our approach is also based on generating random point queries and computing the area of Voronoi

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing info@vldb.org. Articles from this volume were invited to present their results at the 41st International Conference on Very Large Data Bases, August 31st - September 4th 2015, Kohala Coast, Hawaii.

Proceedings of the VLDB Endowment, Vol. 8, No. 12
Copyright 2015 VLDB Endowment 2150-8097/15/08.

cells of the returned tuples, but a key differentiator is that our approach provides an efficient yet *precise* computation of the area of Voronoi cells. This procedure is fundamentally different from the approximate procedure used in [10] for estimating the area of Voronoi cells, and is one of the significant contributions of our paper. This leads to *unbiased estimations* of SUM and COUNT aggregates over LR-LBS interfaces; in contrast, the estimations in [10] have inherent (and unknown) sampling bias.

Moreover, we also leverage the top-k returned tuples of a query (and not just the top-1) by generalizing to the concept of a *top-k Voronoi cell*, leading to significantly more efficient aggregate estimation algorithms. We also developed four different techniques for reducing the estimation error (and thereby estimation error) over LR-LBS interfaces: faster initialization, leveraging history, variance reduction through dynamic selection of query results, and a Monte Carlo method which leverages current knowledge of upper/lower bounds on the Voronoi cell without sacrificing the unbiasedness of estimations. We combine the above ideas to produce Algorithm LR-LBS-AGG, a completely unbiased estimator for COUNT and SUM queries with or without selection conditions. We note that AVG queries can be computed as SUM/COUNT.

Results over LNR-LBS Interfaces: We also consider the problem of aggregate estimations over LNR-LBS interfaces. To the best of our knowledge, this is a novel problem with no prior work. Recall that such type of kNN interfaces only return a ranked list of the top-k tuples in response to a point query, and location information for these tuples is suppressed. None of the prior work for LR-LBS interfaces can be extended to LNR-LBS interfaces. For such interfaces, we develop aggregate estimation algorithms that are not completely bias-free, but can guarantee an arbitrarily small sampling bias. The key idea here is the inference of a tuple's Voronoi cell to an arbitrary precision level with a small number of queries from *merely the ranks of the returned tuples*. We also study a subtle extension to cases where $k > 1$; in particular we study the challenge brought by this case by the (possibly) concave nature of top-k Voronoi cells, and develop an efficient algorithm to detect potential concaveness and guarantee the accuracy of the inferred Voronoi cell. We combine the above ideas to produce Algorithm LNR-LBS-AGG, an estimator for COUNT and SUM queries with or without selection conditions. Unlike Algorithm LR-LBS-AGG, this estimator may be biased, but the bias can be controlled to any arbitrary desired precision.

1.4 Summary of Contributions

- We consider the problem of aggregate estimation over location based services. We consider both LR-LBS (locations returned) as well as the more novel LNR-LBS (locations not returned) interfaces.
- For LR-LBS interfaces, we develop Algorithm LR-LBS-AGG for estimating COUNT and SUM aggregates with or without selection conditions. It represents a significant improvement over prior work along multiple dimensions: a novel way of precisely calculating Voronoi cells leads to completely unbiased estimations; top-k returned tuples are leveraged rather than only top-1; several innovative techniques developed for reducing error and increasing efficiency.
- For LNR-LBS interfaces, we develop Algorithm LNR-LBS-AGG for aggregate estimation. This is a novel problem with no prior work. The estimated aggregates are not bias-free, but the bias can be controlled to any desired precision.
- We conduct a comprehensive set of real-world experiments. Specifically, we conducted online tests over a number of real-world LBS, such as Google Maps, WeChat and Sina Weibo.

2. BACKGROUND

2.1 Model of LBS

A location based service (LBS) supports kNN queries over a database D of tuples with location information. These tuples can be points of interest (e.g. Google Maps) or users (e.g. WeChat, Sina Weibo). A kNN query q takes as input a location (e.g., longitude/latitude combination), and returns the top-k nearest tuples selected and ranked according to a pre-determined ranking function. Since the only input to a query is a location, we use q to also denote the query's location without introducing ambiguity. Almost all of the popular LBS follow kNN query model. For most parts of the paper, we consider the ranking function to be Euclidean distance between the query location and each tuple's location. Extensions to other ranking functions are discussed in [20].

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

LR-LBS: A Location-Returned-LBS (LR-LBS) returns the precise location for each of the top-k returned tuples, along with possibly other attributes. Google Maps, Bing Maps, Yahoo! Maps, etc., are prominent examples of LR-LBS, as all of them display the precise location of each returned POI. Note that some LBS may return a variant of the precise locations - e.g., Skout and Momo returns not the precise location of a tuple, but the precise distance between the query location and the tuple location. We consider such LBS to be in the LR-LBS category because, through previously studied techniques such as trilateration (e.g., [17]), one can infer the precise location of a tuple with just 3 queries.

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

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

Many LBS also impose certain interface restrictions: One is the aforementioned top-k restriction (i.e., only the k nearest tuples are returned). Another common one is a *query rate limit* - i.e., many LBS limit the maximum number of kNN queries one can issue per unit of time. For example, by default Google Maps allows 10,000 location queries per day while Sina Weibo allows only 150 queries per hour. This is an important constraint for our purpose because it makes the *query-issuing* process the bottleneck for aggregate estimation. To understand why, note that even with the generous limit provided by Google Maps, one can issue only 7 queries per minute - this 8.6 second per query overhead[1] is orders of magnitude higher than any offline processing overhead that may be required by the aggregate estimation algorithm. Thus, this interface limitation essentially makes *query cost* the No. 1 performance metric to optimize for aggregate estimation. An LBS might impose other, more subtle constraints - e.g., a maximum coverage limit which forbids tuples far away (say more than 5 miles away) from a query location to be returned. Extensions to address these constraints are in [20].

2.2 Voronoi Cells

Voronoi cell [15] is a key geometric concept used extensively by our algorithms developed in the paper. Consider each tuple $t \in D$ as a point on a Euclidean plane bounded by a box B (which covers all tuples in D). We have the following definition.

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

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

For the purposes of our paper, we define an extension of the Voronoi cell concept to accommodate the top-k (when $k > 1$) query return semantics. Specifically, given a tuple $t \in D$, we define the *top-k Voronoi cell* of t, denoted by $V_k(t)$, as the set of query locations on the plane that return t as one of the top-k results. There are four important observations about this concept:

First, the top-k Voronoi cells for different tuples are no longer mutually exclusive. Each location l belongs to exactly k top-k Voronoi cells corresponding to the top-k tuples returned by query over l. For example, if query over l returns $\{t_i, t_j\}$, then $l \in V_k(t_i)$ and $l \in V_k(t_j)$. Second, our concept of top-k Voronoi cells is *fundamentally different* from the k-th ordered Voronoi cells in geometry [15] - each of which is formed by points with the exact same k closest tuples. The difference is illustrated in Figure 1 - while each colored region is a k-th ordered Voronoi cell, a top-k Voronoi cell may cover multiple regions with different colors. For example, the top-2 Voronoi cell for tuple A is marked by a red border and consists of two different k-th ordered Voronoi cells (AB and AE).

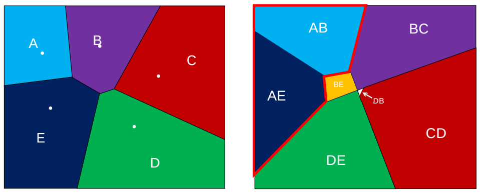

Figure 1: Concavity of top-k Voronoi Diagrams

Third, while both top-1 Voronoi cells and k-th order Voronoi cells are guaranteed to be convex [15], the same does not hold for top-k Voronoi cells when $k > 1$. For example, from Figure 1 we can see that the aforementioned top-2 Voronoi cell for tuple A is concave. Fourth, a top-k Voronoi cell tend to contain many more edges than a top-1 Voronoi cell. As we shall discuss later in the paper, the larger number of edges and the potential concaveness makes computing the top-k Voronoi cell of a tuple t more difficult.

2.3 Problem Definition

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

There are two important notes regarding the selection condition *Cond*. First, we support any selection condition that can be independently evaluated over a single tuple - i.e., it is possible to determine whether a tuple t satisfies *Cond* based on nothing but t. Second, for both LR- and LNR-LBS, we support the specification of a tuple's location as part of *Cond* - even when such a location is not returned, like in LNR-LBS. The technical report [20] describes a technique that allows one to derive the location of a tuple (even with LNR-LBS) to arbitrary precision after issuing a small number of queries. As such, we support aggregates such as the percentage of female WeChat users in Washington, DC. In most part of the technical sections, we focus on aggregates without selection conditions - the straightforward extensions to various types of selection conditions will be discussed in §5.

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

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

3. LR-LBS-AGG

We start this section with our key idea of precisely computing the (top-k) Voronoi cell of a given tuple, which enables the unbiased aggregate estimations over LR-LBS query interfaces. However, it may require a large number of queries leading to a large error when the query budget is limited. Hence we develop four techniques for reducing the estimation error while *maintaining* the complete unbiasedness of aggregate estimations. Finally, we combine all ideas to produce Algorithm LR-LBS-AGG at the end of this section.

3.1 Key Idea: Precisely Compute Voronoi Cells

Reduction to Computing Voronoi Cells: We start by describing a baseline design which illustrates why the problem of aggregate estimations over an LBS's kNN interface ultimately boils down to

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

[2] We assume general positioning [15] - i.e., no two tuples have the exact same location and no four points on the same circle.

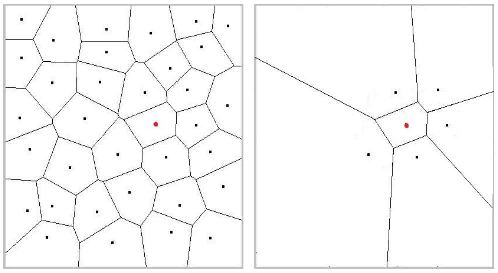

Figure 2: Illustration of Theorem 1

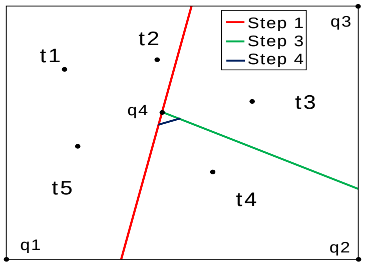

Figure 3: Illustration of LR-LBS-AGG

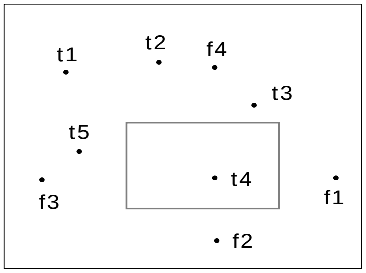

Figure 4: Faster Initialization - Success

computing the volume of the Voronoi cell corresponding to a tuple t. As an example, consider the estimation of COUNT(*) (over a given region) through an LR-LBS with a top-1 interface.

We start by choosing a location q uniformly at random from the region, and then issue a query at q. Let t be the tuple returned by q. Suppose that we can compute the Voronoi cell of t (as defined in §2), say $V(t)$. A key observation here is that the sampling probability of t, i.e., the probability for the above-described randomized process to return t, is exactly $p(t) = \frac{|V(t)|}{|V_0|}$ where $|V(t)|$ and $|V_0|$ are the volume of $V(t)$ and the entire region, respectively. Note that knowledge of $p(t)$ directly leads to a completely unbiased estimation of COUNT(*): $r = 1/p(t)$, because

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

where $\text{Exp}(\cdot)$ is the expected value of the estimation (taken over the randomness of the estimation process), and $|D|$ is the total number of tuples in the database. From (1), one can see that every SUM and COUNT aggregate we support can be estimated without bias - the only change required is on the numerator of estimation. Instead of having 1 as in the COUNT(*) case, it should be the evaluation of the aggregate over t - e.g., if we need to estimation SUM(A_1) where A_1 is an attribute, then the numerator should be $t[A_1]$, i.e., the value of A_1 for t. If the aggregate is COUNT with a selection condition, then the numerator should be either 1 if t satisfies the condition, or 0 if it does not. One can see from the above discussions that, essentially, the problem of enabling unbiased SUM and COUNT estimations is reduced to that of *precisely* computing the volume of $V(t)$, i.e., the Voronoi cell of a given tuple t.

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

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

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

Example 1: Figure 2 provides an illustration for Theorem 1. In order to compute the Voronoi cell of the tuple corresponding to the red dot, it suffices to know the location of the adjacent tuples. Since the region is bounded and each Voronoi edge is a perpendicular bisector between the tuples, the Voronoi cell of the tuple can be computed from the Voronoi edges with the adjacent tuples.

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

According to the theorem, a simple algorithm for constructing the exact Voronoi cell for t is as follows: We start with $D' = \{t\}$. Now the Voronoi cell is the entire region (say an extremely large rectangle). We issue queries corresponding to its four vertices. If any query returns a point we have not seen yet - i.e., not in D' - we append it to D', recompute the Voronoi cell, and repeat the process. Otherwise, if all queries corresponding to vertices of the Voronoi cell return points in D', we have obtained the real Voronoi cell for $t \in D$. One can see that the query complexity of this algorithm is $O(n)$, where n is the number of points in the database D, because each query issued either confirms a vertex of the final Voronoi cell (which has at most $n-1$ vertices), or returns us a new point we have never seen before (there are at most $n-1$ of these too). It is easy to see that the bound is tight - as one can always construct a Voronoi cell that has $n-1$ edges and therefore requires $\Omega(n)$ top-1 queries to discover (after all, each such query returns only 1 tuple). An example here is when t is in the center of a circle, on which the other $n-1$ points are located. Algorithm 1 shows the pseudocode of the baseline approach which we improve in Section 3.2.

Algorithm 1 LNR-LBS-AGG-Baseline

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

Example 2: Figure 3 provides a simple run-through of the algorithm for a dataset with 5 tuples $\{t_1, \ldots, t_5\}$. Suppose we wish to compute $V(t_4)$. Initially, we set $D' = \{t_4\}$ and $V(t_4) = V_0$, the entire region. We issue query q_1 that returns tuple t_5 and hence $D' = \{t_4, t_5\}$. We now obtain a new Voronoi edge that is the perpendicular bisector between t_4 and t_5. The Voronoi edge after step 1 is highlighted in red. In step 2, we issue query q_2 that returns t_4 resulting in no update. In step 3, we issue query q_3 that returns t_3.

$D' = \{t_3, t_4, t_5\}$ and we obtain a new Voronoi edge as the perpendicular bisector between t_3 and t_4 depicted in green. In step 4, we issue query q_4 that returns t_2 resulting in the final Voronoi edge depicted in blue. Further queries over the vertices for $V(t_4)$ does not result in new tuples concluding the invocation of the algorithm.

Extension to $k > 1$: Interestingly, no change is required to the above algorithm when we consider the top-k Voronoi cell rather than the traditional, i.e., top-1 Voronoi cell. To understand why, note that Theorem 1 directly extends to top-k Voronoi cells - as a top-k Voronoi computed from D' still must completely cover that for D; and any vertex of the top-k Voronoi from D' which is outside that from D must return at least one tuple outside D'. We further describe how to leverage $k > 1$ in Sections 3.2.3 and 4.2.

3.2 Error Reduction

Before describing the various error reduction techniques we develop for aggregate estimations over LR-LBS, we would like to first note that, while we use the term "error reduction" as the title of this subsection, some of the techniques described below indeed focus on making the computation of a Voronoi cell more efficient. The reason why we call all of them "error reduction" is because of the inherent relationship between efficiency and estimation error - if the Voronoi-cell computation becomes more efficient, then we can do so for more samples, leading to a larger sample size and ultimately, a lower estimation error (which is inversely proportional to the square root of sample size [16]).

3.2.1 *Faster Initialization*

A key observation from the design in §3.1 is its bottleneck: the initialization process. At the beginning, we know nothing about the database other than (1) the location of tuple t, and (2) a large bounding box corresponding to the area of interest for the aggregate query. Naturally, $D' = \{t\}$, leading to the initial Voronoi cell being the bounding box, and our first four queries being the corners of these bounding boxes. Of course, the tentative Voronoi cell will quickly close in to the real one with speed close to a binary search - i.e., the average-case query cost is at log scale of the bounding box size. Nonetheless, the initialization process can still be very costly, especially when the bounding box is large.

To address this problem, we develop a *faster initialization* technique which features a simple idea: Instead of starting with $D' = \{t\}$, we insert four fake tuples into D', say $D' = \{t, t_1^{\mathrm{F}}, \ldots, t_4^{\mathrm{F}}\}$, where $t_1^{\mathrm{F}}, \ldots, t_4^{\mathrm{F}}$ that form a bounding box around t. The size of the bounding box should be conservatively large - even though a wrong size will not jeopardize the accuracy of our computation.

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

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

Algorithm 2 Fast-Init

```
Input: $t$;     Output: $V(t)$
$D' = \{t, t_1^F, t_2^F, t_3^F, t_4^F\}$; Compute $V(t)$ from $D'$
If all queries over vertices of $V(t)$ return $t$, then $V_t = V_0$
repeat till $D'$ does not change between iterations
    for each vertex $v$ of $V(t)$: $D' = D' \cup$ query($v$)
    Update $V(t)$ from $D'$
return $V(t)$
```

Example 3: Figures 4 and 5 show two different scenarios where the strategy is successful and not successful respectively based on whether the bounding box due to fake tuples is conservatively large. Given a small dataset with tuples $\{t_1, \ldots, t_5\}$, we initialize them with a bounding box induced by fake tuples $\{f_1, \ldots, f_4\}$. In Figure 4, the initial bounding box is tight enough and results in the computation of the precise $V(t_4)$ with much lower query cost (i.e. only tuples $\{t_3, t_5\}$ are visited as against tuples $\{t_2, t_3, t_5\}$ from Example 2. If the bounding box is not tight (as in Figure 5), then queries over all the vertices of the bounding box return t_4 and we revert to original bounding box V_0.

In practice, external knowledge such as US Census data [1] can be used to compute the bounding box. Since the distribution of POIs are often correlated with population density, one can utilize a smaller bounding box (say, 1 square mile) for urban areas and a larger bounding box (say, 50 square miles) for rural areas.

3.2.2 *Leverage history on Voronoi-cell computation*

Another natural optimization is to leverage the information that is gleaned from computing the Voronoi cells of past tuples to compute a tighter initial Voronoi cell. Recall that our algorithm to compute Voronoi cell of a tuple t (i.e $V(t)$), using Theorem 1 starts with an initial Voronoi cell that is an extremely large bounding box that covers the entire plane that then converges to $V(t)$. In the process of computing this Voronoi cell, our algorithm retrieved additional new tuples (by issuing queries for each vertex of the bounding box). Notice that for a LBS with static tuples (such as POIs in Google Maps), the results of location query ordered by distance remains static. Hence it is not necessary to restart every iteration of the algorithm with the same large bounding box. Specifically, when computing the Voronoi cell for the next tuple, we could leverage history by starting with a "tighter" initial bounding box whose vertices are the set of tuples that we have seen so far. In other words, we reuse the tuples that we have seen so far and make them as input to further rounds. Notice that this approach remains the same for both $k = 1$ and $k > 1$. Since the location of each tuple in top-k are returned in LR-LBS, each of these tuples could be leveraged. As we see more tuples, the initial Voronoi cell becomes more granular resulting in substantial savings in query cost. Algorithm 3 provides the pseudocode for the strategy. While the pseudocode uses the simple perpendicular bisector half plane approach [15], it could also use more sophisticated approaches such as Fortune's algorithm [15] to compute the bounding box around tuple t using the tuples from historic queries.

Example 4: As part of computing $V(t_4)$ (see Example 1), we obtained the locations of $t_2, \ldots, t_5$. When computing $V(t_2)$, instead of starting with V_0 (entire region), we could start with a tighter bounding box (shown in red in Figure 6) that can be computed *offline* - i.e. without issuing any queries from previous tuples.

3.2.3 *Variance reduction with larger k*

When the system has $k > 1$, we can of course still choose to use the top-1 Voronoi cell as if only the top result is returned. Or we

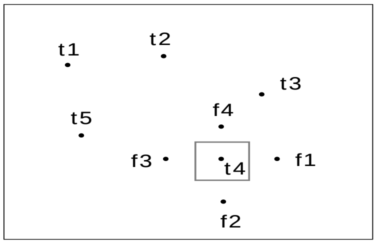

Figure 5: Faster Initialization - Failure

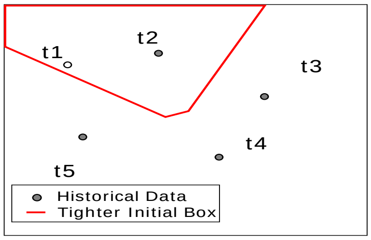

Figure 6: Leveraging History

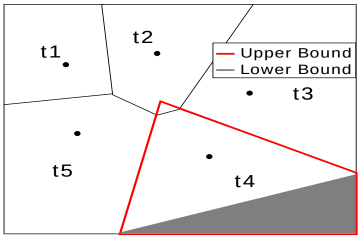

Figure 7: Upper/Lower Bounds

Algorithm 3 Leverage-History

```
Input: $t$ and $H$ (set of tuples obtained from historic queries)
Output: Initial bounding box $V'(t)$
$V'(t) = V_0$
for each tuple $h \in H$
    Update $V'(t)$ with perpendicular bisector between $h$ and $t$
return $V'(t)$ with the tightest bounding box around $t$
```

can choose from any of the top-h Voronoi cells as long as $h \leq k$. While intuitively it might appear that using all k returned tuples is definitely better than using just the top-1, the theoretical analysis suggests otherwise - indeed, whether top-1 or top-h Voronoi cell is better depends on the exact aggregate being estimated - specifically, whether the distribution of the attribute being aggregated is better "aligned" with the size distribution of top-1 or top-h Voronoi cells. To see why, simply consider an extreme-case scenario where the aggregate being estimated is AVG(Salary), and the salary of each user (tuple) is exactly proportional to the size of its top-1 Voronoi cell. In this case, nothing beats using the top-1 Voronoi cells as doing so produces zero variance and thus zero estimation error.

Having said that, however, many aggregates can indeed be better estimated using top-h Voronoi cells, because the sizes of these top-h cells are more uniform than those of the top-1 cells, which can vary extremely widely (see Figure 10 in the experiments section for an example), while many real-world aggregates are also more uniformly distributed than the top-1 cell volume (again, see experiments for justification). But simply increasing h also introduces an undesired consequence: recall from §2 that the larger h is, the more "complex" the top-h Voronoi cell becomes - in other words, the more queries we have to spend in order to pin down the exact volume of the Voronoi cell.

Thus, the key is to make a proper tradeoff between the benefit received (i.e., smaller variance per sample) and the cost incurred (i.e., larger query cost per sample). Our main idea is a combination of two methods: leveraging history in §3.2.2 and upper/lower bound approximation in §3.2.4. Specifically, for each of the k returned tuples, we perform the following process:

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

$$\sum_{t_i : h(t_i) \leq i \leq k} \frac{Q(t_i)}{|V_h(t_i)|} \tag{2}$$

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

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

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

While the above explanation is heuristic in nature it is important to note that, regardless of how we set $h(t_i)$, the estimation we produce for SUM and COUNT aggregates in (2) is always unbiased.

Algorithm 4 Variance-Reduction

```
Input: $H$; Output: Aggregate estimate from all top-$k$ tuples
$q$ = location chosen uniformly at random
for each tuple $t_i$ returned from query($q$)
    $h(t_i)$ = $\max\{h | h \in [2, k], \lambda_h(t_i) \leq \lambda_0\}$
    $h(t_i) = 1$ if no $h$ satisfied the condition $\lambda_h(t_i) \leq \lambda_0$
    Generate estimate for $t_i$ using Equation 2
```

3.2.4 Upper/lower bounds on Voronoi-cell

Note that in the entire process of Voronoi-cell computation (barring the very first step of the faster initialization idea discussed in §3.2.1), we maintain a tentative polygon that covers the entire real Voronoi cell - i.e., an upper bound on its volume. What often arises in practice, especially when computing top-k Voronoi cells (which tend to have many edges), is that even though the bounding polygon is very close to the real Voronoi cell in volume, it has far fewer edges - meaning we still need to issue many more queries to pin down the exact Voronoi cell.

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

$|V'(t)|$ and $|V(t)|$ are the volumes of the bounding polygon and the real Voronoi cell of t, respectively.

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

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

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

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

Example 5: The upper bound $V'(t_4)$ of $V(t_4)$ after Step 3 in the Example 2 (i.e. run-through of Algorithm LR-LBS-AGG-Baseline) is shown in Figure 7 as a quadrilateral with red edges. The three lower vertices of $V'(t_4)$ are guaranteed to be in $V(t_4)$ using the criteria described above and hence the polygon induced by them provides a lower bound estimate for $V(t_4)$.

3.3 Algorithm LR-LBS-AGG

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

Algorithm 5 LR-LBS-AGG

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

4. LNR-LBS-AGG

4.1 Voronoi Cell Computation: Key Idea

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

We start by defining a primitive operation of "binary search" as follows. Consider the objective of finding the Voronoi cell of a tuple t in the database. Given any location c_1 and c_2 (not necessarily occupied by any tuple), where c_1 returns t, consider the half-line from c_1 passing through c_2. Since a Voronoi cell is convex and c_1 resides within the Voronoi cell, this half-line has one and only one intersection with the Voronoi cell - which is associated with one or two edges of the Voronoi cell. We define the primitive operation of *binary search* for given c_1, c_2 to be the binary search process of finding one Voronoi edge associated with the intersection. Please refer to Appendix A for the detailed design of this process.

Naturally, such a binary search process is associated with an error bound on the precision of the derived edge. For example, we can set an upper bound ϵ on the maximum distance between any point on the real Voronoi edge (i.e., a line segment) and its closest point on the derived edge, which we refer to as the *maximum edge error*, and use ϵ as the objective of the binary search operation. One can see that the number of queries required for this binary search is proportional to $\log(1/\epsilon)$. Due to space limitations, detailed analysis of the query complexity can be found in [20].

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

We start with one query at point q which returns t. Then, we construct 4 points that bound q (say $q_1 : \langle x(q) - 1, y(q)\rangle$, $q_2 : \langle x(q) + 1, y(q)\rangle$, $q_3 : \langle x(q), y(q) - 1\rangle$, $q_4 : \langle x(q), y(q) + 1\rangle$, where $x(\cdot)$ and $y(\cdot)$ are the two dimensions, e.g., longitude and latitude, of a location, respectively) and call upon the binary search operation to find the corresponding Voronoi edges intercepting the half lines from q to $q_1, \ldots, q_4$, respectively. One can see that, no matter what the discovered edges might be, they must form a closed polygon[3] which we can use to initiate the testing process described in §3.1. If all vertices pass the test, then we have already obtained the Voronoi cell of t. Otherwise, for each vertex (say v) that fails the test, we perform the binary search operation on the location of v to discover another Voronoi edge. We repeatedly do so until all vertices pass the test - at which time we have obtained the real Voronoi cell - subject to whatever error bound specified for the binary search process (as described above).

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

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

$$|E(\tilde{\theta} - \theta)| \le \sum_{t \in D} \frac{\epsilon^2 - 2 \cdot d(t) \cdot \epsilon}{(d(t) - \epsilon)^2}, \tag{3}$$

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

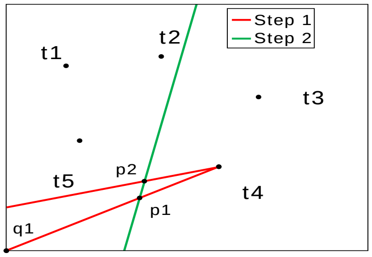

Figure 8: Illustration of Algorithm LNR-LBS-AGG

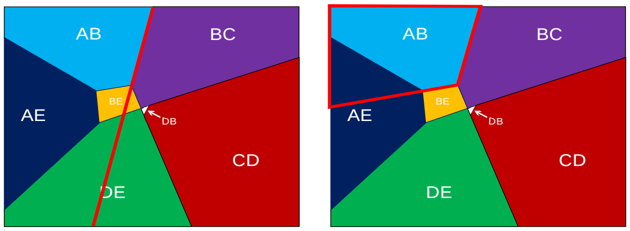

Figure 9: Handling Concavity of top-k Voronoi Diagrams

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

Due to space limitations, please refer to [20] for proof. Estimation bias for other aggregates can be derived accordingly (given the distribution of the attribute being aggregated). One can make two observations from the theorem: First, the smaller maximum edge error ϵ is or the large inter-tuple distance $d(t)$ is, the smaller the bias will be. Second, we can make the bias arbitrarily small by shrinking ϵ - which leads to a log-scale increase of the query cost. Algorithm 6 shows the pseudocode for LNR-LBS-AGG.

Algorithm 6 LNR-LBS-AGG

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

Example 6: We consider the same dataset as Example 1, except that in LNR-LBS the locations of tuples are not returned. Figure 8 shows a run-through of the algorithm by which one of the Voronoi edges of $V(t_4)$ is identified. Initially, the bounding box contains the entire region, i.e. V_0. ℓ_1 and ℓ_2 are two lines starting from t_4 constructed as per Algorithm 7. p_1 and p_2 are mid points of small line segments on ℓ_1 and ℓ_2 such that points on either side of them return different tuples when queried. The new estimated Voronoi edge is computed as the line segment connecting p_1 and p_2. Please refer to Appendix-A for further details.

4.2 Extension to $k > 1$

A complication brought by the rank-only return semantics is the extension to cases with $k > 1$. Specifically, recall from §2 that the (extended) top-k Voronoi cell might be *concave* when $k > 1$. In the case LR-LBS case, this does not cause any problem because, at any moment, our derived top-k Voronoi cell is computed from the exact tuple locations of all observed tuples and (therefore) completely covers the real top-k Voronoi cell. For LNR-LBS case, however, this is no longer the case: Since we unveil the top-k Voronoi cell edge after edge, if we happen to come across one of the "concave edges" early, then we may settle on a sub-region of the real top-k Voronoi cell. Figure 9 demonstrates an example for such a scenario.

Fortunately, there is an efficient fix to this situation. To understand the fix, a key observation is that any "inward" (i.e., concave) vertex of a top-k Voronoi cell, say that of t, must be at a position with equal distance to three tuples, one of them being t (Note: this might not hold for "outward" vertices). This property is from the following lemma whose proof can be found in [20].

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

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

Now our task is reduced to finding such a missing inward vertex. This is equivalent with finding the perpendicular bisector of t and every other tuple in D' - as once these perpendicular bisectors are identified, the rest is simply getting their intersections which can be done offline. For each tuple in D', we either have already identified the perpendicular bisector through one of the previous calls to the binary search process - or we can initiate a new one as follows.

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

5. DISCUSSIONS

5.1 Leveraging External Knowledge

In previous discussions, we focused on how to process the results returned by a randomly chosen query (e.g., how to compute the top-k Voronoi cell of a returned tuple). The way the initial query is chosen, however, remains a simple design of choosing a location uniformly at random from the bounding region. Without any knowledge of the distribution of tuple locations, uniform distribution is the natural choice. However, in real-world applications, we often have certain *a priori* knowledge of the tuple distributions, which we can leverage to optimize the sampling distribution of queries.

For example, if our goal is to estimate an aggregate, say COUNT, of Point-Of-Interests (POIs, e.g., restaurants) in the US, a reasonable assumption is that the density of POIs in a region tends to be positively correlated with the region's population density. Thus, we have two choices: either to sample a location uniformly at random - which leads to POIs in rural areas to be returned with a much higher probability (because their Voronoi cells tend to be larger); or to sample a location with probability proportional to its population density - which hopefully leads to a more-or-less uniform selection probabilities over all POIs. Clearly, the second strategy is likely better for COUNT estimation, as a more uniform selection probability distribution directly leads to a smaller estimation variance (and therefore error). For example, in the extreme-case scenario where all POIs are selected with equal probability, our COUNT estimation will be precise with zero error. Thus, an optimization technique we adopt in this case is to design the initial sampling distribution of queries according to the population density information retrieved from external sources, e.g., US Census data [1].

There are two important notes regarding this optimization: First, regardless of the accuracy of external knowledge, the COUNT and SUM estimations remain unbiased as Equation 1 in §3.1 guarantees unbiasedness no matter what the sampling distribution $p(t)$ is. Second, the optimal sampling distribution depends on both the tuple distribution and the aggregate query itself. For example, if we want to estimate the SUM of review counts for all POIs, then the optimal sampling distribution is to sample each tuple with probability proportional to its review count (as this design produces zero estimation variance and error). Given the difficulty of predicting the aggregate (e.g., review COUNT in this case) ahead of time, leveraging external knowledge is better considered as a heuristic (a very effective one, as we shall demonstrate in experimental results).

Other Extensions: Detailed discussion about the following extensions can be found in [20].

- **Tuple Position Computation:** We show how one can infer the position of a tuple in LNR-LBS at a level of arbitrary precision. This problem, while of independent interest, is also critical for enabling the estimations of aggregates that feature selection conditions on tuples' locations (e.g., the COUNT of social network users within 10 meters of major highways).
- **Aggregates with Selection Conditions:** In our paper, we considered aggregates without selection condition. Extension to selection conditions is relatively straightforward. We discuss both selection conditions that can be "passed through" to LBS and those that are not supported by the LBS.
- **Special LBS Constraints:** Often, real-world LBS enforce a number of constraints such as *maximum radius* (that truncates tuples beyond a fixed threshold) and more complex ranking functions that take additional factors into account.
- **Extension to Higher Dimensions:** All popular LBS are limited to 2D. However, extending our algorithms to higher dimensions (if only for theoretical interests) is straightforward.

6. EXPERIMENTAL RESULTS

6.1 Experimental Setup

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

Offline Real-World Dataset: To verify the correctness of our results, we started by testing our algorithms locally over OpenStreetMap [4], a real-world spatial database consisting of POIs (including restaurants, schools, colleges, banks, etc.) from public-domain data sources and user-created data.

We focused on the USA portion of OpenStreetMap. To enrich the SUM/COUNT/AVG aggregates for testing, we grew the attributes of POIs (specifically, restaurants and schools) by "JOINing" OpenStreetMap with two external data sources, Google Maps [3] and US Census [1]. Specifically, we added for each (applicable) restaurant POI its review ratings from Google Maps; and each school POI its enrollment number from US Census. The US Census data is also used as the (optional) external knowledge source - i.e., to provide the population density data for the optimization technique discussed in §5.

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

Online LBS Demonstrations: In order to showcase the efficacy of our algorithms in real-world applications, we also conducted experiments *online* over 3 very popular real-world LBS: Google Maps [3], WeChat [6], and Sina Weibo [5]. Each of these services has at least hundreds of millions of users. Unlike the offline experiments, we do not have direct access to the ground-truth aggregates due to the lack of partnership with these LBS. Nonetheless, we did attempt to verify the accuracy of our aggregate estimations with information provided by external sources (e.g., news reports) - more details later in the section.

In online experiments for LR-LBS, we used Google Maps, specifically its Google Places API [3], which takes as input a query location (latitude/longitude pair) and (optionally) filtering conditions such as keywords (e.g., "Starbucks") or POI type (e.g., "restaurant"), and returns at most $k = 60$ POIs nearby, ordered by distance from low to high, with location and other relevant information (e.g., review ratings) returned for each POI.

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

An implementation-related issue regarding WeChat is that, unlike its mobile apps, its API does not support nearest-neighbor search. Thus, we conducted our experiments by running its Android app (with support for nearest-neighbor search) on the official Android emulator, and used the debugging feature of location spoofing to issue queries from different locations. We then used the MonkeyRunner tool[4] for Android emulator to interact with the app - i.e., sending queries and receiving results. Specifically, to extract query answers from the Android emulator, we first took a screenshot of the query-answer screen, and then parsed the results through an OCR engine, tesseract-ocr[5].

Algorithms Evaluated: We mainly evaluated three algorithms in our experiments: LR-LBS-AGG and LNR-LBS-AGG from §3 and §4, respectively, along with the only existing solution for LR-LBS

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

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

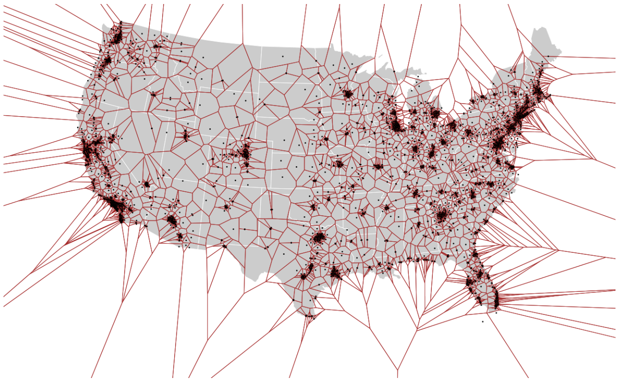

Figure 10: Voronoi Decomposition of Starbucks in US

(note there is no existing solution for LNR-LBS), which we refer to as LR-LBS-NNO [10]. LR-LBS-NNO has a number of tuneable parameters - we picked the parameter settings and optimizations from [10] that provided the best performance.

Performance Measures: As discussed in §2, we measure efficiency through query cost, i.e., the number of queries issued to the LBS. Our estimation accuracy is measured experimentally by relative error. Each data point is obtained as the average of 25 runs.

6.2 Experiments over Real-World Datasets

Unbiasedness of Estimators: Our first experiment seeks to show the unbiasedness of our estimators for LR-LBS-AGG and LNR-LBS-AGG even after incorporating the various error reduction strategies. LR-LBS-NNO is known to be unbiased from [10] after an expensive bias correction step. Figure 11 shows a trace of the three algorithms when estimating COUNT of all restaurants in US by plotting the current estimate periodically after fixed number of queries have been issued to LBS. We can see that LR-LBS-NNO has a high variance and takes significantly longer to converge while our estimators quickly converge to the ground truth much before LR-LBS-NNO. This indicates that the error reduction techniques successfully reduce the variance of our estimators.

Query Cost versus Relative Error: We start by testing the key tradeoff - i.e., query cost vs. relative error - for all three algorithm over various aggregates. Specifically, Figures 13, 14, 15 and 16 show the results for four queries, COUNT of schools in US, COUNT of restaurants in US, SUM of school enrollments in US, and AVG of restaurant ratings in Austin, Texas, respectively. One can see that not only our LR-LBS-AGG algorithm significantly outperform the previous LR-LBS-NNO [10] in all cases, even our algorithm for the LNR-LBS case achieves much better performance than the previous algorithm (despite the lack of tuple locations in query results).

Query Cost versus LBS Size: Figure 17 shows the impact of LBS database size (in terms of number of POIs or users) on query cost to estimate the COUNT of schools in US for a fixed relative error of 0.1 . We varied the database size by picking a subset of the database (such as 25%, 50%, etc) uniformly at random and estimating the aggregate over it. As expected for a sampling-based approach, the increase in database size do not have any major impact and only results in a slight increase in overall query cost (due to the more complex topology of Voronoi cells).

Query Cost versus k: Figure 18 shows how the value of k (the number of tuples returned by k-NN interface) affects the query cost. Again, we measure the query cost required to achieve a relative error of 0.1 on the aggregate COUNT of schools in US. We compared an variant that leverages our variance reduction strategy that adaptively decides which subset of tuples (i.e. h of top-k) to use with fixed variants that uses all the top-k tuples. As expected, our adaptive strategy has a lower query cost and consistently achieves a saving of 10% of query cost.

Efficacy of Error Reduction Strategies: We started by verifying the effectiveness of weighted sampling using external knowledge - Figure 12 compares the performance of the two sampling strategies - uniform and weighted - while estimating the COUNT of schools in US. One can see that the weighted sampling variants result in significant savings in query cost.

In our final set of experiments, we evaluated the efficacy of the various error reduction strategies we described in the paper. We compared 5 different variants of our algorithm for LR-LBS ranging from no error reduction strategies (LR-LBS-AGG-0) to sequentially adding them one by one in the order discussed in the paper culminating in LR-LBS-AGG that incorporates all of them. Figure 19 shows the results of this experiment. As expected the first two strategies of faster initialization and leveraging history caused a significant reduction in query cost. We observed that the results for LNR-LBS were very similar.

6.3 Online Demonstrations

Google Places: Our first online demonstration of LR-LBS-AGG was on Google Places API and estimating two aggregates with different selection conditions. The first involves selection conditions that can be passed over to LBS (COUNT of Starbucks in US) while the second involves aggregates with selection condition that cannot be passed over (see §5 for discussion) such as COUNT of restaurants in Austin, Texas that are open on Sundays.

Table 1 shows the results of the experiments. We also verified the accuracy of our estimates for first aggregate (COUNT of Starbucks) through the public release of Starbucks Corp [2]. One can see from the table that, with just 5000 queries, LR-LBS-AGG achieves very accurate estimations ($< 5\%$ relative error) for the count.

To provide an intuitive illustration of the execution of our algorithm, we also continued the estimation of COUNT(“Starbucks”) until enumerating all Starbucks in the US. Figure 10 demonstrates the Voronoi diagram constructed by our algorithm at the end. One can see the vastly different sizes of Voronoi cells - spanning hundreds of thousands km^2 in rural areas and smaller than 1km^2 in urban cities, justifying the effectiveness of weighted sampling.

WeChat and Sina Weibo: We estimated two aggregates, (1) total number of users and (2) gender ratio, over two LNR-LBS, WeChat and Sina Weibo, respectively. Table 1 shows the results of the experiments. One can observe from the table that our estimations quickly converge to a narrow range (+/- 5%) after issuing a small number of queries (10000). While we do not have access to the ground truth this time, we do note an interesting observation from our results: the percentage of male users is much higher on WeChat than on Sina Weibo - an observation verified by various surveys in China [7]. We would like to note that the COUNT aggregate measures the number of users who have enabled the location feature of WeChat and Weibo respectively and is different from the number of registered or active accounts.

Localization Accuracy: As a final set of experiments, we also evaluated the effectiveness of our Tuple position computation approaches in tracking real world users. Specifically, we sought to identify the precise location of *static* objects located across the region. We conducted this experiment over Google Places in US and WeChat in China. We treated Google Places as LNR-LBS by ignoring the location provided its API. We sought to identify the location of 200 randomly chosen POIs after issuing at most 100 queries

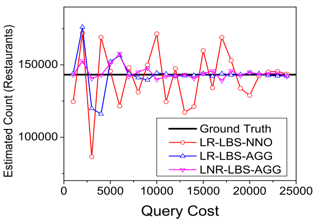

Figure 11: Unbiasedness of Estimators

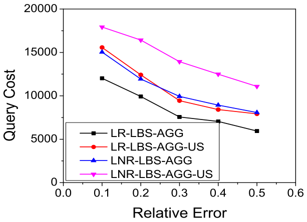

Figure 12: Impact of Sampling Strategy

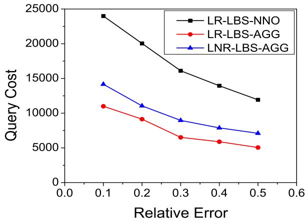

Figure 13: COUNT(schools)

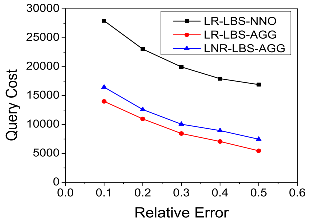

Figure 14: COUNT(restaurants)

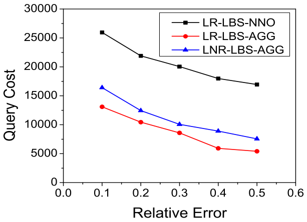

Figure 15: SUM(enrollment) in Schools

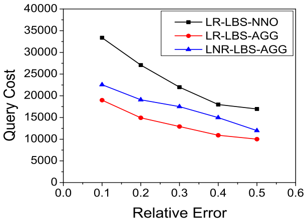

Figure 16: AVG(ratings) in Austin, TX Restaurants

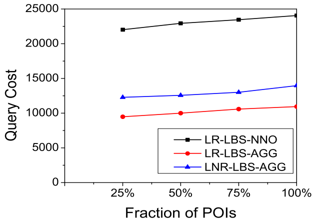

Figure 17: Varying Database Size

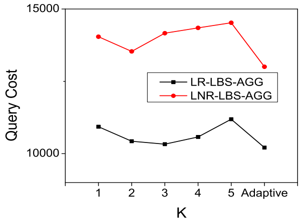

Figure 18: Varying k

Table 1: Summary of Online Experiments

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

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

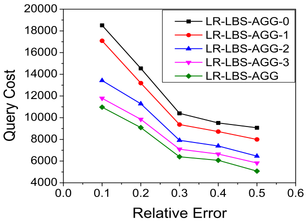

Figure 19: Query Savings of Error Reduction Strategies

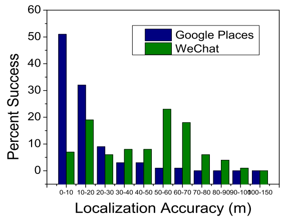

Figure 20: Localization Accuracy

7. RELATED WORK

Analytics and Inference over LBS: Location based Services (LBS) such as map services (Google Maps) and location based social networks (such as FourSquare, WeChat, Sina Weibo) are becoming popular in recent years. The prior work on analytics over LBS focussed exclusively on the LR-LBS scenario. The closest prior work is [10] that seeks to estimate COUNT and SUM aggregates over LR-LBS using a nearest neighbor oracle. It then corrects the bias by using the area of the Voronoi cell using an approach that is very expensive. Aggregate estimation over LBS such as FourSquare that does not provide nearest neighbor oracle interface could be done using [18, 23]. [18] proposed a random region sampling method with an unknown estimation bias that could be eliminated using techniques from [23]. However, none of them work for LNR-LBS. There has been work on inferring the location and other private information of users of LBS. [17] proposed trilateration based methods to infer the location of users even when the LBS only provided relative distances. There has been other extensive work [21, 22, 24] on inferring location information and re-identification of users although none of them are applicable for the LBS models studied in this paper.

Aggregate Estimations over Hidden Web Repositories: There has been a number of prior work in performing aggregate estimation over static hidden databases. [12] provided an unbiased estimator for COUNT and SUM aggregates for *static* databases with form based interfaces. [11, 13, 14, 19] describe efficient techniques to obtain random samples from hidden web databases that can then be utilized to perform aggregate estimation. For hidden databases with keyword interfaces, prior work have studied estimating the size of search engines [8, 25, 26] or a corpus [9].

8. FINAL REMARKS

In this paper, we explore the problem of aggregate estimation over location based services that are increasingly popular. We introduced a taxonomy of LBS with k-NN query interface based on whether location of the tuple is returned (LR-LBS) or not (LNR-LBS). For the former, we proposed an efficient algorithm and various error reduction strategies that outperforms prior work. We initiate study into the latter by proposing effective algorithms for aggregation and inferring the position of tuple to arbitrary precision which might be of independent interest. We verified the effectiveness of our algorithms by using a comprehensive set of experiments on a large real-world geographic dataset and online demonstrations on high-profile real-world websites.

9. ACKNOWLEDGEMENTS

The authors were partially supported by National Science Foundation under grants 0915834, 1018865, 0852674, 1117297, 1343976, Army Research Office under grant W911NF-15-1-0020 and a grant from Microsoft Research.

10. REFERENCES

[1] http://www.census.gov/2010census/data/.

[2] .http://investor.starbucks.com/phoenix.zhtml?c=99518&p=irol-financialhighlights.

[3] Google Places API. https://developers.google.com/places/documentation/.

[4] OpenStreetMap. http://www.openstreetmap.org/.

[5] Sina Weibo. http://weibo.com/.

[6] WeChat. http://www.wechat.com/en/.

[7] WeChat/Weibo Statistics. http://www.guancha.cn/Media/2015_01_29_307911.shtml.

[8] Z. Bar-Yossef and M. Gurevich. Random sampling from a search engine's corpus. *Journal of the ACM*, 55(5), 2008.

[9] A. Broder and et.al. Estimating corpus size via queries. In *CIKM*, 2006.

[10] N. Dalvi, R. Kumar, A. Machanavajjhala, and V. Rastogi. Sampling hidden objects using nearest-neighbor oracles. In *SIGKDD*, 2011.

[11] A. Dasgupta, G. Das, and H. Mannila. A random walk approach to sampling hidden databases. In *SIGMOD*, 2007.

[12] A. Dasgupta and et.al. Unbiased estimation of size and other aggregates over hidden web databases. In *SIGMOD*, 2010.

[13] A. Dasgupta, N. Zhang, and G. Das. Leveraging count information in sampling hidden databases. In *ICDE*, 2009.

[14] A. Dasgupta, N. Zhang, and G. Das. Turbo-charging hidden database samplers with overflowing queries and skew reduction. In *EDBT*, 2010.

[15] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf. *Computational geometry*. Springer, 2000.

[16] D. A. Freedman. *Statistical models: theory and practice*. cambridge university press, 2009.

[17] M. Li and et.al. All your location are belong to us: Breaking mobile social networks for automated user location tracking. In *MobiHoc*, 2014.

[18] Y. Li, M. Steiner, L. Wang, Z.-L. Zhang, and J. Bao. Dissecting foursquare venue popularity via random region sampling. In *CoNEXT workshop*, 2012.

[19] W. Liu and et.al. Aggregate estimation over dynamic hidden web databases. In *VLDB*, 2014.

[20] W. Liu, M. F. Rahman, S. Thirumuruganathan, N. Zhang, and G. Das. Aggregate estimations over location based services. *arXiv preprint arXiv:1505.02441*, 2015.

[21] C. Y. Ma, D. K. Yau, and N. S. Rao. Privacy vulnerability of published anonymous mobility traces. *ToN*, 2013.

[22] M. Srivatsa and M. Hicks. Deanonymizing mobility traces: Using social network as a side-channel. In *CCS*, 2012.

[23] P. Wang and et.al. An efficient sampling method for characterizing points of interests on maps. In *ICDE*, 2014.

[24] H. Zang and et.al. Anonymization of location data does not work: A large-scale measurement study. In *MobiCom*, 2011.

[25] M. Zhang, N. Zhang, and G. Das. Mining a search engine's corpus: efficient yet unbiased sampling and aggregate estimation. In *SIGMOD*, 2011.

[26] M. Zhang, N. Zhang, and G. Das. Mining a search engine's corpus without a query pool. In *CIKM*, 2013.

APPENDIX

A. BINARY SEARCH PROCESS

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

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

Now suppose that ℓ_1 produces a satisfactory segment of at most δ long. Let this segment be (c_5, c_6). We simply return our (estimated) Voronoi edge as the line that passes through: (1) the midpoint of (c_3, c_4), and (2) the midpoint of (c_5, c_6). One can see that the overall query cost of the binary search process is at most $3\log(b/\delta)$. Analysis for the error bound can be found in [20]. Algorithm 7 provides the pseudocode for Binary Search process.

Algorithm 7 Binary-Search

```
Input: Tuple $t$, Locations $c_1, c_2$ where query($c_1$) returns $t$
Output: An edge of $V(t)$
$c_b$ = Intersection of half-line $c_1, c_2$ with bounding box
Find $c_3, c_4$ s.t. $dist(c_3, c_4) < \delta$ and $query(c_3) \neq query(c_4)$
$r = dist(c_1, c_4)$
Construct lines $\ell_1, \ell_2$ from $c_1$ with angles $\pm \arcsin(\delta'/r)$
$(c_5, c_6)$ = line segment on $\ell_1$ or $\ell_2$ with $dist(c_5, c_6) < \delta$ and
    query($c_5$) $\neq$ query($c_6$)
if none exists, return perpendicular bisector of $(c_3, c_4)$
else return line segment passing through midpoints of $(c_3, c_4)$
    and $(c_5, c_6)$
```