Monitoring the Top-*m* Rank Aggregation of Spatial Objects in Streaming Queries

Farhana M. Choudhury Zhifeng Bao J. Shane Culpepper
RMIT University, Melbourne, Australia
Email: farhana.choudhury, zhifeng.bao, shane.culpepper@rmit.edu.au

Timos Sellis
Swinburne University of Technology
Hawthorn, Australia
Email: tsellis@swin.edu.au

***Abstract*—In this paper, we propose and study the problem of top-*m* rank aggregation of spatial objects in streaming queries, where, given a set of objects *O*, a stream of spatial queries (*k*NN or range), the goal is to report the *m* objects with the highest aggregate rank. The rank of an object with respect to an individual query is computed based on its distance from the query location, and the aggregate rank is computed from all of the individual rank orderings. In order to solve this problem, we show how to upper and lower bound the rank of an object for any unseen query. Then we propose an approximation solution to continuously monitor the top-*m* objects efficiently, for which we design an Inverted Rank File (IRF) index to guarantee the error bound of the solution. In particular, we propose the notion of *safe ranking* to determine whether the current result is still valid or not when new queries arrive, and propose the notion of *validation objects* to limit the number of objects to update in the top-*m* results. We also propose an exact solution for applications where an approximate solution is not sufficient. Last, we conduct extensive experiments to verify the efficiency and effectiveness of our solutions. This is a fundamental problem that draws inspiration from three different domains: rank aggregation, continuous queries and spatial databases, and the solution can be used to monitor the importance / popularity of spatial objects, which in turn can provide new analytical tools for spatial data.**

I. Introduction

Rank aggregation is a classic problem in the database community which has seen several important advances over the years [1, 2, 3, 4, 5]. Informally, *rank aggregation* is the problem of combining multiple rank orderings to produce a single "best" ordering. Typically, this translates into finding the top-*m* objects with the highest aggregate rank, where the algorithms used for ranking and aggregation can take several different forms. Common ranking and aggregation metrics include majority ranking (sum, average, median, and quantile), consensus-based ranking (Borda count), and pairwise disagreement based ranking (Kemeny optimal aggregation) [1, 2]. Rank aggregation has a wide variety of practical applications such as determining election winners, sports analytics, collaborative filtering, meta-search, and aggregation in database middleware.

One such application area where rank aggregation can be applied is in spatial computing [6]. In spatial databases for example, a fundamental problem is to rank objects based on their proximity from a query location, where two pervasively used spatial query types are *k*-nearest neighbor (*k*NN) query and range query (which ranks the objects within the search range by their distance from the query *q* [7]).

Spatial queries are an important tool that provides partially ranked lists over a set of objects. Each object *o* receives a different ranking (or is not ranked at all) which depends on the query location. Thus, aggregating the ranks of spatial objects can provide key insights into object importance in many different scenarios. A recent example that inspires this work is a real estate data visual analytics system called Houseseeker [1] we have built [8]. Houseseeker facilitates finding houses based on spatial preference. The ranking of each house is based on the distance from a preferred location (close to a school, railway station, or supermarket). A house that is ranked high by many users' queries has a higher aggregate rank, and is therefore more popular in the market. Clearly, popularity in this context changes continuously over time as new buyers search for houses. Continuously monitoring the housing properties with the highest aggregate rank, regardless of how rank is defined is of practical importance to both buyers and sellers: (1) the buyers can continuously be informed of the trending houses on the market, and make informed buying decisions; (2) In Houseseeker, we intend to operationalize the solutions developed in this paper by using user preferences to continuously monitor the "hot" houses on the market, and to find similar houses which are not currently for sale to provide insights and notifications to potential sellers on the best time to enter the market.

In this paper, we consider the problem of top-*m* rank aggregation of spatial objects for streaming queries, where, given a set of objects *O*, a stream of spatial queries (*k*NN or range), the problem is to report the *m* objects with the highest aggregate rank. Here, an object that satisfies the query constraint is ranked based on its distance from the query location, and the aggregation is computed using all of the individual rank orderings. To maintain recency information and minimize memory costs, a sliding window model is imposed on the query stream, and a query is valid only while it remains in the window. We consider two of the most common models for sliding windows, the *count-based* window and the *time-based* window [9].

Our work draws inspiration from three different domains – spatial databases, continuous queries, and rank aggregation. While several seminal papers have considered various combinations of these three domains, no previous work has considered approaches to combining all three. We summarize previous work, and the subtle distinctions between previous

[1]http://115.146.89.158

best solutions in these problem domains and our work in Section II.

In the domain of rank aggregation, previous solutions have addressed the problem of incrementally computing individually ranked lists using *on-demand* algorithms [3, 4]. However, these approaches do not consider streaming queries, and there is no obvious way to easily extend these approaches to the sliding window problem. A related body of work is to find the most frequent objects over a stream, which is essentially a count aggregation [10, 11]. However, our goal is to efficiently compute the aggregation without explicitly requiring the ranked results to be fully computed for every query separately, so these approaches are not directly amenable to our problem.

In the domain of continuous spatial queries, objects are streaming, but the queries do not change [9, 12, 13]. Continuous result updates of top-k queries where the query location is changing have also been extensively studied in the literature [14, 15]. These approaches make the assumption that a query location can move only to an adjacent location, and construct a *safe region* around the queries, such that the top-k results do not change as long as the query location remains in the safe region. These problems are subtly different from the streaming query problem explored in this work, where each new query location can be anywhere in space and the query does not move.

In the domain of spatial databases, other related work on finding the top objects with the maximum number of Reverse k Nearest Neighbors (RkNN) exists [16, 17, 18]. Given a set of objects O, the $RkNN(o)$ is the set of objects containing o as a kNN. Another variant of RkNN is bichromatic, where given a set of objects O and a set of users U, the $RkNN(o)$ is the set of users regarding o as a kNN of O. Although the count of RkNN is also an aggregation, these solutions do not consider the rank position of the objects for the aggregation. Rather, the approaches rely on properties of skyline and k-skyband queries to estimate the number of RkNN for an object. Finding the exact rank of an object in a skyline or a k-skyband is not straightforward. Moreover, to the best of our knowledge, there is no previous work on the continuous case of finding the object with the maximum number of RkNN for streaming queries (users).

Our contributions are summarized as below. (i) We propose and formalize the problem of top-m rank aggregation on a sliding window of spatial queries, which draws inspiration from the three classical problem domains – rank aggregation, continuous query and spatial databases. (Sec. III)
(ii) We propose an exact solution to continuously monitor the top-m ranked objects. (Sec. III)
(iii) We propose an approximation algorithm with bounded error guarantees to maximize the reuse of the computations from previous queries in the current window, and show how to incrementally update the top-m results only when necessary (Sec. V). In particular, the following three technical contributions have been made. (1) We propose the notion of *safe ranking* to determine whether the result set in a previous window is still valid or not in the current window. (2) We propose the notion of *validation objects* which are able to limit the number of objects to be updated in the result set. (3) We show how to use an *Inverted Rank File* (IRF) index to bound the error of the solution.

To summarize, aggregating spatial object rankings can provide key insights into the importance of objects in many different problem domains. Our proposed solutions are generic and applicable to many different spatial rank aggregation problems, and a variety of different query types such as range queries, k-nearest neighbor (kNN) queries, and reverse kNN (RkNN) queries can be adapted and used within our framework.

II. Related Work

Since our work draws inspiration from three different problem domains in database area – rank aggregation, spatial queries and streaming queries, we review the related work for each domain and combinations of two domains (if any).

A. Rank aggregation

Given a set of ranked lists, where objects are ranked in multiple lists, the problem is to find the top-m objects with the highest aggregate rank. This is a well studied and classic problem, mostly for its importance in determining winners based on the ranks from different voters [1, 2, 3, 4].

The approaches of Fagin et al. [1] and Dwork et al. [2] assume that the ranked lists exist before aggregation. When the complete ranked lists are not available a priori, or random access in a ranked lists is expensive, Mamoulis et al. [4] and a variant presented by Fagin et al. [3] have shown that the ranked objects for an individually ranked list can be computed incrementally one-by-one using an on-demand aggregation. However, such incremental approaches [3, 4] are not straightforward to extend to sliding windows where queries are also removed from the result set as new queries arrive.

B. Database queries

The relevant work can be categorized mainly as - (i) moving, (ii) streaming, and (iii) maximum top-k queries.

Moving queries. In spatial databases, given a moving query and a set of static objects, the problem is to report the query result continuously as the query location moves [14, 15, 19]. The most common assumption made to improve efficiency is that a query location can move only to a neighboring region [15, 19]. By maintaining a *safe region* around the query location, a result set is updated only when the query moves out of the safe region. Li et al. [14] substitute the safe region with a set of *safe guarding objects* around the query location such that as long as the current result objects are closer to query than any safe guarding objects, the current result remains valid.

The problem of continuously updating kNN results when both the query location and the object locations can move was initially explored by Mouratidis et al. [20]. Mouratidis et al. solve the problem by using a conceptual partitioning of the space around each query, where the partitions are processed iteratively to update results when the query or any of the objects moves. In contrast, streaming queries studied in this paper can originate anywhere in space and does not move, thus the safe region based approaches are not applicable.

Query processing over streaming objects. Many different streaming problems have been explored over the years, among which the problem of continuous maintenance of query results [12] is most closely related to our problem. Bohm et al.

[12] explore an expiration time based recency approach where objects are only valid in a fixed time window. Other related work explored sliding window models where objects are valid only when they are contained in the sliding window [9, 13, 21]. The two most common variants of sliding windows are - (i) count based windows which contain the $|W|$ most recent data objects; and (ii) time-based windows which contain the objects whose time-stamps are within $|W|$ most recent time units.

The general approach in all of these solutions is as follows – Queries are registered to an object stream, and as a new object arrives, the object is reported to the queries if it qualifies as a result for that query. The solutions rely on the idea of a skyline, where the set of objects that are not dominated by any other object in any dimension must be considered. In these models, the queries are static, and the skyline is computed for a query. Newly arriving objects can be pruned based on the properties of the skyline. The key difference between our problem and related streaming problems is that objects are static in our model while the queries are streaming.

Maximizing Reverse Top-*k*. Another related body of work is reverse top-*k* querying [17, 22, 23]. Given a set of objects and a set of users, the problem is to find the object that is a top-*k* object of the maximum number of users. Li et al. [17] explore solutions for spatial databases using precomputed Voronoi diagrams. Other solutions use skyline and *k*-skyband to estimate the number of users that have an object as a top-*k* result. A *k*-skyband contains the objects that are dominated by at most $k-1$ objects. Unlike top-*k* queries, the number of result objects of a range query is not fixed, therefore maintaining a skyband is not straightforward for range queries.

A related problem in spatial databases is to find a region in space such that if an object is placed in that region, the object will have the maximum number of reverse *k*NNs [24, 25, 26]. Solutions for this problem depend on static queries (users), and are therefore not directly applicable to our problem. Moreover, these solutions do not consider the rank position of the object in the top-*k* results in their solutions.

Gkorgkas et al. [27] study the problem of maximizing R*k*NN in temporal data, which returns the object with the highest continuity score (the maximum number of consequent intervals for which an object o is a top-*m* object based on its R*k*NN count). Although the problem is scoped temporally, both the queries and the objects in the database are static.

III. Preliminaries

A. Problem Definition

Table I presents the basic notation used in the remainder of the paper. Let O be a set of N objects where $o \in O$ is a point in d-dimensional Euclidean space, X^d. Now consider a stream of user queries SQ which is an infinite sequence $\langle q_1, q_2, \dots \rangle$ in order of their arrival time. Each query q is a point in X^d, associated with a spatial constraint, $Con(q)$, such as range or *k*NN. In this work, we focus primarily on range queries, but our solutions are easily generalized to other spatial query types.

We adopt the sliding window model where a query is only valid while it belongs to the current sliding window W. We present our work for a count-based sliding window, and then discuss the extensions required for a time-based window in

TABLE I: Basic notation

Symbol	**Description**
W	Sliding window of $\lvert W\rvert$ most recent queries.
$\boldsymbol{d}(o,q)$	Euclidean distance between object o and query q.
$Con(q)$	Spatial constraint (range or *k*NN) of q.
$\boldsymbol{r}(o,q)$	Ranked position of o based on $\boldsymbol{d}(o,q)$.
O_q^+	The set of objects in O that satisfy $Con(q)$.
$\rho(o,W)$	Popularity (aggregated rank) of o for queries in W.
c	A leaf level cell of a Quadtree.
$\boldsymbol{d}^{\downarrow}(o,c_q)$ ($\boldsymbol{d}^{\uparrow}(o,c_q)$)	The minimum (maximum) Euclidean distance between o and any query in c_q.
$\boldsymbol{r}^{\downarrow}(o,c_q)$ ($\boldsymbol{r}^{\uparrow}(o,c_q)$)	Lower (upper) bound rank of o for any query q in cell c_q.
B	Block size of the rank lists.
$\boldsymbol{d}^{\downarrow}(b,c_q)$ ($\boldsymbol{d}^{\uparrow}(b,c_q)$)	The minimum (maximum) distance between any object in a block b and any query in cell c_q.

Sec. V-F. A count-based window contains the $|W|$ most recent items, ordered by arrival time. Before defining our problem, we first present a rank aggregation measure of an object for a window of $|W|$ queries, denoted as *popularity*.

Popularity measure. Each query q partitions O into two sets, $O_q^+ = \{o \in O \mid o \text{ satisfies } Con(q)\}$ and $O_q^- = \{o \in O \mid o \text{ does not satisfy } Con(q)\}$. Each object $o^+ \in O_q^+$ is ranked based on the Euclidean distance from q, $\boldsymbol{d}(o,q)$. Other distance measures can be used to rank the objects, but are not illustrated in this work due to space limit. The rank of o w.r.t. q, $\boldsymbol{r}(o,q) = i$, is defined as the i·th position of $o \in O_q^+$ in an ordered list indexed from $i = 1$ to $|O_q^+|$ where $\boldsymbol{d}(o_i^+,q) \leq \boldsymbol{d}(o_{i+1}^+,q)$.

The popularity of an object $o \in O$ in a sliding window W of queries is an aggregation of the ranks of o with respect to the queries in W. We now formally define *Popularity* (ρ) as a rank aggregation function for a sliding window of $|W|$ queries. Other similar aggregation functions are applicable to our problem but beyond the scope of this paper.

$$\rho(o,W) = \frac{1}{|W|}\sum_{i=1}^{|W|} \begin{cases} N - \boldsymbol{r}(o,q_i) + 1 & \text{where } o \in O_{q_i}^+ \\ 0 & \text{otherwise} \end{cases}$$

A higher value of $\rho(o,W)$ indicates higher popularity. If an object does not satisfy the constraint of a query, the contribution in the aggregation for that query is zero. We now formally define our problem as follows:

Definition 1. Top-*m* popularity in a sliding window of spatial queries (T*m*ρQ) problem. *Given a set of objects O, the number of objects to monitor m, and a stream of spatial queries SQ* $(q_1, q_2, \dots)$*, maintain an aggregate result set* $\mathcal{R}$*, such that* $\mathcal{R} \subseteq O$, $|\mathcal{R}| = m$, $\forall o \in \mathcal{R}$, $o' \in O \backslash \mathcal{R}$, $\rho(o,W) \geq \rho(o',W)$*, where W contains the* $|W|$ *most recent queries.*

B. Baseline

A straightforward approach to continuously monitor the top-*m* popular objects in W is: (i) Each time a new query, qn arrives, compute the rank of all the objects in O_{qn}^+ that satisfy $Con(qn)$. (ii) Update ρ of the objects $o \in O_{qn}^+$ for qn, and the objects $o' \in O_{qo}^+$ for the query qo. Here, qo is the least recent query that is removed from W as qn arrives. (iii) Sort all of the objects that are contained in O_q^+ for at least one query q in the current window, and return the top-*m* objects with the highest ρ as $\mathcal{R}$. As there is no prior work on aggregating spatial query results in a sliding window (See Section II), we consider this straightforward solution as a baseline approach.

Unfortunately, the baseline approach is computationally expensive for several reasons: (i) For each query, the ranks of

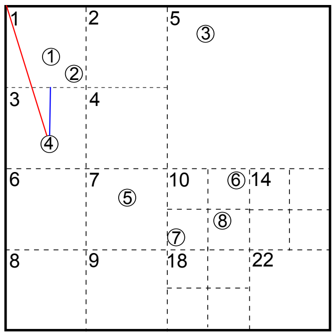


Fig. 1: Computing rank bounds

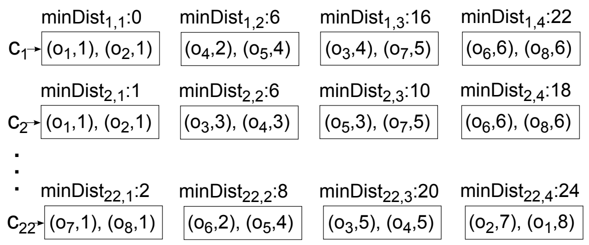


Fig. 2: An example inverted rank file

all objects that satisfy the $Con(q)$ must be computed. As the number of objects can be very large, and the queries can arrive at a high rate, this step incurs a high computational overhead. (ii) Each time the sliding window shifts, $\boldsymbol{\rho}$ for a large number of objects may need to be updated. (iii) The union of all of the objects that satisfy $Con(q)$ for each query in the current window must be sorted by the updated $\boldsymbol{\rho}$.

To overcome these limitations, we seek techniques which avoid processing objects for the query stream that cannot affect the top-m objects in $\mathcal{R}$. This minimizes the number of popularity computations that must occur. Two possible approaches to accomplish this, are: accurately estimate the rank of the objects for newly arriving queries, or reuse the computations from prior windows efficiently. We consider both of these approaches in the following sections.

IV. Rank Bounds and Indexing

In this section, we first present how to compute an upper bound and a lower bound for the rank of an object w.r.t. an unseen query, and then propose an indexing approach referred to as an *Inverted Rank File* (IRF) that can be used to estimate the rank of objects for arriving queries.

A. Computing Rank Bounds

Here, we assume that the space has been partitioned into cells (the space partitioning step is explained in Section V-A). The rank bound for an object o w.r.t. a cell c_q is computed as follows. For any query q arriving with a location in cell c_q, the rank of o satisfies the condition, $\boldsymbol{r}^{\downarrow}(o,c_q) \leq \boldsymbol{r}(o,q) \leq \boldsymbol{r}^{\uparrow}(o,c_q)$, where $\boldsymbol{r}^{\downarrow}(o,c_q)$ and $\boldsymbol{r}^{\uparrow}(o,c_q)$ are the lower and the upper bound rank of o for any query q in cell c_q, respectively.
Lower bound rank. The lower bound rank is computed from the number of objects $o' \in O \backslash o$ that are definitely closer to q in c_q than o. Note that a smaller rank indicates a smaller Euclidean distance from the query. Specifically, let ℓ_n be the number of objects $o' \in O \backslash o$ such that $\boldsymbol{d}^{\uparrow}(o',c_q) \leq \boldsymbol{d}^{\downarrow}(o,c_q)$, where $\boldsymbol{d}^{\uparrow}(o',c_q)$ ($\boldsymbol{d}^{\downarrow}(o,c_q)$) are the maximum (minimum) Euclidean distance between o' (o) and cell c_q. Therefore, even if the query location q is the closest point of c_q to o, there are still at least ℓ_n objects closer to q than o. So the rank of o must be greater than ℓ_n for any query in c_q, i.e., $\boldsymbol{r}^{\downarrow}(o,c_q) = \ell_n + 1$.
Upper bound rank. Let, u_n be the number of objects $o' \in O \backslash o$ such that, $\boldsymbol{d}^{\downarrow}(o',c_q) < \boldsymbol{d}^{\uparrow}(o,c_q)$ for any query q in c_q. Therefore, even if a query q arrives at the farthest location in c_q from o, there are at most u_n objects that can possibly be closer to q than o. So the rank of o cannot be greater than $u_n + 1$ for any query in c_q, resulting in $\boldsymbol{r}^{\uparrow}(o,c_q) = u_n + 1$.

Example 1. *In Figure 1, let $O = \{o_1, o_2, \ldots, o_8\}$ be the set of objects and c_1 be a cell in X^d. The minimum and maximum distances between c_1 and an object o_4 are shown with a blue and a red line, respectively. Here, only the maximum distance between c_1 and o_1 is less than $\boldsymbol{d}^{\downarrow}(o_4,c_1)$. Therefore, $\boldsymbol{r}^{\downarrow}(o_4,c_1) = 1 + 1 = 2$. The minimum distance between c_1 and each of the objects o_1, o_2 and o_3 is less than $\boldsymbol{d}^{\uparrow}(o_4,c_1)$, so, $\boldsymbol{r}^{\uparrow}(o_4,c_1) = 3 + 1 = 4$.*

B. Indexing Rank

We present an indexing technique called an *Inverted Rank File* (IRF) where X^d is partitioned into different cells, and the bounds of each object's rank for queries appearing in the cell is precomputed. The rank information is indexed such that, if a query q arrives anywhere inside a cell c_q, the rank of any object for q can be estimated. A quadtree structure is employed to partition X^d into cells. We present the general structure of an IRF, then in Section V-A we present a space partitioning approach to approximately answer T$m\rho$Q with bounded error.
Inverted Rank File. An IRF consists of a collection of all leaf level cells of the quadtree, and a set of *rank lists*, one for each leaf level cell c of the quadtree. Each rank list is a sorted sequence of tuples of the form $\langle o, \boldsymbol{r}^{\downarrow}(o,c_q)\rangle$, one for each object $o \in O$, sorted in ascending order of $\boldsymbol{r}^{\downarrow}(o,c_q)$. If multiple objects have the same $\boldsymbol{r}^{\downarrow}(o,c_q)$ for a cell c_q, those tuples are sorted by $\boldsymbol{d}^{\downarrow}(o,c_q)$. Here, $\boldsymbol{r}^{\downarrow}(o,c_q)$ is the lower bound rank of o for a query q coming in cell c_q. Each rank list is stored as a sequence of blocks of a fixed length, B. Each block b of the rank list for cell c_q is associated with the minimum distance between c_q and any object in b, where $\boldsymbol{d}^{\downarrow}(b,c_q) = \min_{o \in b} \boldsymbol{d}^{\downarrow}(o,c_q)$.

Let a quadtree partition the space into 22 disjoint leaf cells in Figure 1 and the block size $B = 2$. Figure 2 shows the resulting IRF index for the objects $O = \{o_1, o_2, \ldots, o_8\}$ derived from Figure 1.

V. Approximate Solution

As queries can arrive at a very high rate, there may be instances where the cost of computing the exact solution is too expensive. In this section, we show how to exploit the rank bounds to find an approximate solution with a guaranteed bound for the T$m\rho$Q problem. Table II summarizes the notation used to present the approximate solution. At the highest level, the approximate solution consists of the following steps:

1) A space partitioning technique is presented (in Sec. V-A) to construct an IRF index that supports an incremental computation of the approximate solution of T$m\rho$Q (in Sec. V-B).
2) A *safe rank* is computed which represents a threshold, if this threshold is exceeded by a result object currently in $\mathcal{R}$,

TABLE II: Notation used in Sec. V

Symbol	Description
qo	The least recent query, which is excluded from W.
qn	The most recent query, which is added to W.
$\hat{\boldsymbol{r}}(o,q)$ $(\hat{\boldsymbol{\rho}}(o,W))$	Approximation of $\boldsymbol{r}(o,q)$ $(\rho(o,W))$
$\mathcal{R}_i$	The set of result objects for a window W_i.
o_m	The m·th object from the set $\mathcal{R}_{i-1}$.
$\hat{\boldsymbol{r}}^{\downarrow}(b,q)$ $(\hat{\boldsymbol{r}}^{\uparrow}(b,q))$	Lower (upper) bound rank of any object in block b for q.
$\hat{\boldsymbol{\rho}}(o_{m+1},W_{i-1})$	The approximate popularity of top $(m+1)$·th object from $\mathcal{R}_{i-1}$ in previous window W_{i-1}.
$\hat{\boldsymbol{\rho}}(o_m,W_{i-1} \backslash qo)$	The approximate popularity of top m·th object from $\mathcal{R}_{i-1}$, updated w.r.t. excluding q_o from Window W_{i-1}.
$\hat{\boldsymbol{\rho}}(o_m,W_i)$	The approximate popularity of top m·th object from $\mathcal{R}_i$.

that object must remain in $\mathcal{R}$ as a valid result. Specifically, the current safe rank can be computed by combining: (i) a *block-level* and (ii) an *object-level safe rank* (Sec. V-C)

3) If the ranks for all of the result objects are *safe*, $\mathcal{R}$ does not need to be updated. Otherwise, more verification must be done to determine if any object can affect $\mathcal{R}$. This can be achieved using a second technique called *validation objects*, which incrementally identifies the objects that can affect $\mathcal{R}$ (Sec. V-D). As long as the current result objects have a higher popularity than the validation objects, $\mathcal{R}$ does not need to be updated.
4) If $\mathcal{R}$ must be updated, the approximate popularity of the affected objects are computed. We show that the computations of the prior windows can be used to efficiently approximate the popularity scores of the objects that must change in the current window (Sec. V-D).

The approximation error bound is discussed in Sec. V-E.

A. Space partitioning

Ideally, rank bound estimations should be as close as possible to the actual rank of each object. If the quadtree leaf cell where a query q arrives is as small as a single point location (the same as the location of q), then both the upper and the lower bound ranks of any object will be exactly the same as the actual rank of that object for q. However, if the space is partitioned in this way, then the number of cells will be infinite. Therefore, we propose a partitioning technique which guarantees that the difference between the bound of any object's rank and its true rank is bounded by a threshold, ε. Specifically, for any $o \in O$, and any leaf level quadtree cell c, the difference between the upper and the lower bound rank must be within a *percentage of the lower bound rank*:

$$\boldsymbol{r}^{\uparrow}(o,c) - \boldsymbol{r}^{\downarrow}(o,c) \leq \varepsilon \times \boldsymbol{r}^{\downarrow}(o,c) \tag{1}$$

Otherwise, c is further partitioned until the condition holds. As an example, let $\varepsilon = 0.5$. For an object o, and a cell c_i, let $\boldsymbol{r}^{\downarrow}(o,c_i) = 10$ and $\boldsymbol{r}^{\uparrow}(o,c_i) = 20$. So, c_i needs to be further partitioned for o until the condition is met. Let for another cell c_j, $\boldsymbol{r}^{\downarrow}(o,c_j) = 100$ and $\boldsymbol{r}^{\uparrow}(o,c_j) = 120$. Now cell c_j does not need to be partitioned for o since $120 - 100 \leq 0.5 \times 100$.

The intuition behind this partitioning scheme becomes quite clear when the notion of "top" ranked objects is taken into consideration. Getting the exact position of the highest ranked object matters much more than getting the exact position of the object at the thousandth position. So, the granularity of exactness in our inequality degrades gracefully with the true rank of the object.

Partitioning process. The partitioning can be achieved iteratively, where the quadtree root is initialized with the entire space X^d. The process starts from the root and recursively partitions X^d. If the partitioning condition is not satisfied for an object o and a cell c, partitioning of c continues until Condition (1) is met. The process terminates when $\forall o$, the condition holds for all of the current leaf level quadtree cells c.

Why use a quadtree? We use a quadtree to partition the space and then organize the spatial information for each quadtree cell. The rationale is as follows: (i) A quadtree partitions the space into mutually-exclusive cells. In contrast, MBRs in an R-tree may have overlaps, so a query location can overlap with multiple partitions, making it difficult to estimate the object ranks in new queries. (ii) A quadtree is update-friendly, and the partition granularity can be dynamically changed using ε to improve the accuracy of the approximation. This allows performance to be quickly and easily tuned for different collections. (iii) In a quadtree, a cell c is partitioned only when any rank bounds for c do not satisfy Condition (1). In contrast, if a regular grid structure of equal cell size is used, enforcing partitioning using Condition (1) will result in unnecessary cells being created.

B. Framework of approximate solution

In this section, we first introduce how to compute the approximate popularity of an object for a given sliding window, then we show how to aggregate the top-m approximate results. Since this section is all about how to compute the approximate popularity of objects, we use the terms popularity and approximate popularity interchangeable, unless specified otherwise.

First, a lemma is presented to show that the rank of any object o for a query q arriving in a cell c can be estimated using only the lower bound rank, $\boldsymbol{r}^{\downarrow}(o,c)$ within an error bound.

Lemma 1. *For any object $o \in O$, and any query q arriving in cell c, $\boldsymbol{r}^{\downarrow}(o,c) \leq \boldsymbol{r}(o,q) \leq (1+\varepsilon) \times \boldsymbol{r}^{\downarrow}(o,c)$ always holds.*

Proof: The rank bounds are computed such that $\boldsymbol{r}^{\downarrow}(o,c) \leq \boldsymbol{r}(o,q) \leq \boldsymbol{r}^{\uparrow}(o,c)$ always holds. For any object $o \in O$, and for any leaf level cell c of the quadtree, the space is partitioned in a way that guarantees $\boldsymbol{r}^{\uparrow}(o,c) - \boldsymbol{r}^{\downarrow}(o,c) \leq \varepsilon \times \boldsymbol{r}^{\downarrow}(o,c)$, so, clearly $\boldsymbol{r}^{\downarrow}(o,c) \leq \boldsymbol{r}(o,q) \leq (1+\varepsilon) \times \boldsymbol{r}^{\downarrow}(o,c)$ also holds. ■

Based on Lemma 1, we approximate the rank of an object with an error bound as:

$$\hat{\boldsymbol{r}}(o,q) = (1+\frac{\varepsilon}{2}) \times \boldsymbol{r}^{\downarrow}(o,c) \tag{2}$$

Corollary 1. *For any object $o \in O$, and any query q arriving in cell c, $|\boldsymbol{r}(o,q) - \hat{\boldsymbol{r}}(o,q)| \leq \varepsilon/2 \times \boldsymbol{r}^{\downarrow}(o,c)$ always holds.*

Proof: Here, the maximum difference between $\boldsymbol{r}^{\downarrow}$(o,c) and $\hat{\boldsymbol{r}}$(o,q), and that between $\boldsymbol{r}^{\uparrow}$(o,c) and $\hat{\boldsymbol{r}}$(o,q), are both $\varepsilon/2 \times \boldsymbol{r}^{\downarrow}(o,c)$. Therefore, the proof follows from Lemma 1. ■

The approximate popularity $\hat{\boldsymbol{\rho}}(o,W)$ of an object o for the queries q in a W can be computed using rank approximation:

$$\hat{\boldsymbol{\rho}}(o,W) = \frac{1}{|W|}\sum_{i=1}^{|W|} \begin{cases} N - \hat{\boldsymbol{r}}(o,q_i) + 1 & \text{where } o \in O^{+}_{q_i} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Updating a count based sliding window W_i of queries from the previous window W_{i-1} can be formulated as replacing the least recent query qo by the most recent query qn. As a result, only the leaf level cells (in the quadtree) that contain qn and qo need to be found, namely c_{qn} and c_{qo}. The rank lists corresponding to these cells can be quickly retrieved from the

Algorithm 1: T$m\rho$Q

```
1.1  Input: Window $W_i$, number of result objects $m$, the result objects $\mathcal{R}_{i-1}$ of the previous window $W_{i-1}$, and the $m+1$·th best popularity $\hat{\rho}(o_{m+1}, W_{i-1})$ of the previous window $W_{i-1}$.
1.2  Output: Result objects $\mathcal{R}_i$ of the current window $W_i$.
1.3  Initialize a max-priority queue $PQ$
1.4  $\mathcal{R}_i \leftarrow \emptyset$; $qn \leftarrow W_i \backslash W_{i-1}$; $qo \leftarrow W_{i-1} \backslash W_i$
1.5  for $o \in \mathcal{R}_{i-1}$ do
1.6      $\hat{\rho}(o, W_{i-1} \backslash qo) \leftarrow \hat{\rho}(o, W_{i-1}) - \zeta(\hat{r}(o, qo))/|W_i|$
1.7  $\hat{\rho}(o_m, W_{i-1} \backslash qo) \leftarrow$ the approximate popularity of top $m$·th object from $\mathcal{R}_{i-1}$ after updating for $qo$.
1.8  $BSR \leftarrow$ BLOCK_SAFE_RANK($\hat{\rho}(o_{m+1}, W_{i-1})$,$\hat{\rho}(o_m, W_{i-1} \backslash qo)$,$PQ$)
1.9  for $o \in \mathcal{R}_{i-1}$ do
1.10     $\hat{\rho}(o, W_i) \leftarrow \hat{\rho}(o, W_{i-1} \backslash qo) + \zeta(\hat{r}(o, qn))/|W|$
1.11     if $\hat{r}(o, qn) \leq BSR$ AND $o \in O^+_{qn}$ then
1.12         $\mathcal{R}_i \leftarrow o$
1.13 if $|\mathcal{R}_i| < m$ then
1.14     $\hat{\rho}(o_m, W_i) \leftarrow$ current $m$·th best popularity of $\mathcal{R}_{i-1}$ in $W_i$.
1.15     $OSR \leftarrow$ OBJECT_SAFE_RANK($\hat{\rho}(o_{m+1}, W_{i-1})$, $\hat{\rho}(o_m, W_i)$,$PQ$)
1.16     for $o \in \mathcal{R}_{i-1} \backslash \mathcal{R}_i$ do
1.17         if $\hat{r}(o, qn) \leq OSR$ AND $o \in O^+_{qn}$ then
1.18             $\mathcal{R}_i \leftarrow o$
1.19 if $|\mathcal{R}_i| < m$ then
1.20     $VO \leftarrow$ VALIDATION_OBJECTS($\hat{\rho}(o_m, W_i)$, $PQ$)
1.21     if $VO \neq \emptyset$ then
1.22         $\mathcal{R}_i \leftarrow$ UPDATE_RESULTS($VO$,$\mathcal{R}_{i-1} \backslash \mathcal{R}_i$)
1.23 RETURN $\mathcal{R}_i$
```

IRF index. For each window W_i, assume that $m+1$ objects with the highest $\hat{\rho}$ are computed, where the top-m objects are returned as the result $\mathcal{R}_i$ of T$m\rho$Q for W_i, and the popularity of the $(m+1)$·th object is used in the next window to identify the safe rank and the validation objects efficiently.

The steps for updating the approximate solution of T$m\rho$Q for a window W_i are shown in Algorithm 1. Note that the necessary notation is defined in Table II. Here, $\zeta(o,q)$ is the contribution of q to the popularity of o, and is computed as:

$$\zeta(\hat{r}(o,q)) = \begin{cases} N - \hat{r}(o,q) + 1 & \text{where } o \in O^+_q \\ 0 & \text{otherwise} \end{cases}$$

First, the approximate popularity of the result objects $o \in \mathcal{R}_{i-1}$ for the excluded query qo, $\hat{r}(o,qo)$ is updated. Let the updated m·th highest popularity from $\mathcal{R}_{i-1}$ be $\hat{\rho}(o_m, W_{i-1} \backslash qo)$ (Lines 1.5 - 1.7 in Algorithm 1). The rest of the algorithm consists of three main components - (i) computing the safe rank in two steps (block-level and object-level safe rank), (ii) finding the set of validation objects, and (iii) updating $\mathcal{R}_i$.

Locating an object in IRF. Since some of the steps in Algorithm 1 require finding the entry of a particular object in a rank list of IRF, we first present an efficient technique for locating objects, and then describe the remaining steps of the algorithm. We start with a lemma that find a relation between the minimum Euclidean distance of the objects from a cell c and the lower bound ranks of the objects for any query in c.

Lemma 2. *For any two objects $o_i, o_j \in O$ and a cell c, if $\boldsymbol{r}^{\downarrow}(o_i,c) \leq \boldsymbol{r}^{\downarrow}(o_j,c)$, then $\boldsymbol{d}^{\downarrow}(o_i,c) \leq \boldsymbol{d}^{\downarrow}(o_j,c)$ always holds.*

Proof: See extended ArXiV version [28] for proof. ■

Lemma 2 show that sorting the objects by $\boldsymbol{r}^{\downarrow}(o,c)$ is equivalent to sorting the objects by their minimum Euclidean distance to c, $\boldsymbol{d}^{\downarrow}(o,c)$. If multiple objects have the same $\boldsymbol{r}^{\downarrow}(o,c)$, they are already stored as sorted by $\boldsymbol{d}^{\downarrow}(o,c)$ as described in Sec. IV-B. Therefore, we can locate the position of o in the rank list of cell c as: (1) Compute the minimum Euclidean distance $\boldsymbol{d}^{\downarrow}(o,c)$ of c from o. (2) As described in Sec. IV-B, each block b of the rank list for cell c is associated with the minimum distance between c_q and any object in b, $\boldsymbol{d}^{\downarrow}(b,c)$, so a binary search on $\boldsymbol{d}^{\downarrow}(b,c)$ can be performed to find the position of the block b where o is stored. (3) Perform a linear scan in that block to find the entry for o.

The entire process has a $\mathcal{O}(log_2(N/B)+B)$ time complexity, where B is the number of objects in a block.

C. Safe rank

Recall that in Algorithm 1 the purpose of finding a safe rank is to minimize the number of updates in $\mathcal{R}_{i-1}$ (result objects in the previous window W_{i-1}) to get the result set $\mathcal{R}_i$ (in the current window W_i) whenever the sliding window shifts. In particular, the idea is to compute the safe rank OSR for the objects $o \in \mathcal{R}_{i-1}$ such that, if $\hat{r}(o,qn) < OSR$, then no other object from $o' \in O \backslash \mathcal{R}_{i-1}$ can have a higher $\hat{\rho}$ than o, thereby o is a valid result in $\mathcal{R}_i$ as well. Note that a smaller value of rank implies a higher contribution to the popularity measure. The safe rank is defined w.r.t. the current window W_i by default.

Before presenting the computation of an object's safe rank, the concept of **popularity gain** of an object o is introduced, which results from replacing the least recent query qo by the most recent query qn, and is denoted by Δ_o:

$$\Delta_o = \hat{\rho}(o,W_i) - \hat{\rho}(o,W_{i-1}) = \frac{\zeta(\hat{r}(o,qn)) - \zeta(\hat{r}(o,qo))}{|W_i|} \quad (4)$$

Here, if o does not satisfy the query constraint $Con(q)$, the contribution of q to the popularity of o, $\zeta(\hat{r}(o,qn)) = 0$. Let $\Delta^{\uparrow}_o$ denote the maximum popularity gain among all objects (in the current window W_i). Then the popularity of any object $o' \in O \backslash \mathcal{R}_{i-1}$ can be at most $\hat{\rho}(o_{m+1}, W_{i-1}) + \Delta^{\uparrow}_o$, where $\hat{\rho}(o_{m+1}, W_{i-1})$ is the $(m+1)$·th highest approximate popularity in the previous window W_{i-1}. In other words, if the updated popularity of an object $o \in \mathcal{R}_{i-1}$ is higher (better) than $\hat{\rho}(o_{m+1}, W_{i-1}) + \Delta^{\uparrow}_o$, then such an o is guaranteed to remain in $\mathcal{R}_i$, which inspires the design of object-level safe rank OSR as:

$$\hat{\rho}(o_m, W_{i-1} \backslash qo) + \frac{N - OSR + 1}{|W_i|} \geq \hat{\rho}(o_{m+1}, W_{i-1}) + \Delta^{\uparrow}_o \quad (5)$$

Since a lower rank indicates a higher contribution to the popularity, the gain will be maximized when the difference between $\zeta(\hat{r}(o,qn))$ and $\zeta(\hat{r}(o,qo))$ is maximized. Therefore, the goal of minimizing the updates of R_{i-1} can be reduced to the challenge of how to compute a tight estimation of $\Delta^{\uparrow}_o$.

1) Block-level popularity gain: Since the objects are arranged blockwise in an IRF index, and each object o is sorted by the lower bound rank $\boldsymbol{r}^{\downarrow}(o,c_q)$ in ascending order, a *block-level gain* can also be used as the first step in finding a tighter estimation of the maximum gain. A block-level maximum gain $\Delta^{\uparrow}_b$ is computed such that $\Delta^{\uparrow}_b \geq \Delta^{\uparrow}_o$, which can be used to find the block-level safe rank, BSR. If the rank of any result object is not better than BSR for qn, then an object-level maximum gain, $\Delta^{\uparrow}_o$ is computed. The object-level safe rank OSR can be computed using this value, where $OSR \geq BSR$, as a lower value of rank implies a higher gain. If the rank of any result object is still not safe, then the validation objects (proposed in Sec. V-D) must be checked to decide if $\mathcal{R}_{i-1}$ needs to be updated. Here, a part of the safe rank calculations can be reused to find the validation objects, which will be explained in Section V-D.

Block-level gain computation. Given a block b from the rank list of qn, the block-level gain Δ_b is an overestimation of the gain of the objects $o \in b$, such that $\Delta_b \geq \Delta_o$. As the gain is maximized when the difference between $\zeta(\hat{\boldsymbol{r}}(o,qn))$ and $\zeta(\hat{\boldsymbol{r}}(o,qo))$ is maximized, a technique to compute Δ_b can be actualized by finding: (i) a lower bound estimation of the rank of any object $o \in b$ for qn, namely $\hat{\boldsymbol{r}}^{\downarrow}(b,qn)$, where, $\hat{\boldsymbol{r}}^{\downarrow}(b,qn) \leq \hat{\boldsymbol{r}}(o,qn)$; and (ii) an upper bound estimation of the rank that any object $o \in b$ can have for qo, denoted as $\hat{\boldsymbol{r}}^{\uparrow}(b,qo)$, such that $\hat{\boldsymbol{r}}(o,qo) \leq \hat{\boldsymbol{r}}^{\uparrow}(b,qo)$.

Since the objects are sorted in ascending order of lower bound ranks in the IRF index, the lower bound rank of the first entry of b is implicitly $\hat{\boldsymbol{r}}^{\downarrow}(b,qn)$. Here, $\forall o \in b, \hat{\boldsymbol{r}}^{\downarrow}(b,qn) \leq \boldsymbol{r}^{\downarrow}(o,qn)$ holds by definition.

Next, for the same block b of the rank list of qn, finding the maximum rank $\hat{\boldsymbol{r}}^{\uparrow}(b,qo)$ that any object $o \in b$ can have for qo is needed. To achieve this, a block b' is found such that all of the objects $o \in b$ are guaranteed to be in the rank list of qo before b'. As the objects are sorted by $\boldsymbol{r}^{\downarrow}(o,c_{qo})$ in the rank list of c_{qo}, $\boldsymbol{r}^{\downarrow}(o',c_{qo})$ is guaranteed to be greater than that of any object in b', where o' is the first entry of b'. Therefore, $\boldsymbol{r}^{\downarrow}(o',qo)$ is taken as the upper bound estimation, $\hat{\boldsymbol{r}}^{\uparrow}(b,qo)$.

For a tight estimation of $\hat{\boldsymbol{r}}^{\uparrow}(b,qo)$, the block b' with the smallest $\boldsymbol{r}^{\downarrow}(o',qo)$ must be found. As the objects and blocks of a rank list are sorted by the minimum Euclidean distance from the corresponding cell (Section IV-B), and $\forall o \in b$, $\boldsymbol{d}^{\downarrow}(o,c_{qo}) \leq \boldsymbol{d}^{\uparrow}(b,c_{qo})$, a binary search over the blocks of the rank list of qo is performed to find the first position of the block b' where $\boldsymbol{d}^{\uparrow}(b,c_{qo}) \leq \boldsymbol{d}^{\downarrow}(b',c_{qo})$. Here, $\boldsymbol{d}^{\uparrow}(b,c_{qo})$ is computed as the maximum Euclidean distance between the minimum bounding rectangle of the objects $o \in b$ and cell c_{qo}.

Example 2. *In Figure 2, let c_1 and c_2 be the cell where qn and qo arrive respectively. Let the constraint of both queries are satisfied by all of the objects; let $b = \langle (o_4,2),(o_5,4) \rangle$ be the block of the rank list of c_1 currently under consideration. Here, $\hat{\boldsymbol{r}}^{\downarrow}(b,qn) = 2$, which is the lower bound of the first entry of b. Let $\boldsymbol{d}^{\uparrow}(b,c_2) = 14$, computed from the MBR of block b and cell c_2. Now, a binary search is performed with the value 14 over the $\boldsymbol{d}^{\downarrow}$ of the blocks in c_2. Here, we get $b' = \langle (o_6,6),(o_8,6) \rangle$, as $\boldsymbol{d}^{\downarrow}(b',c_2) = 18$, which is the smallest value of $\boldsymbol{d}^{\downarrow}$ greater than 14, shown with an arrow. So, $\hat{\boldsymbol{r}}^{\uparrow}(b,qo) = 6$ is the lower bound rank of the first entry of b'.*

2) Block-level safe rank: By making use of the values $\hat{\boldsymbol{r}}^{\downarrow}(b,qn)$ and $\hat{\boldsymbol{r}}^{\uparrow}(b,qo)$ of block b, a block-level estimation of the maximum gain for W_i can found, and a block-level safe rank *BSR* can be computed, as shown in Algorithm 2. Algorithm 2 shows the steps to compute the block-level safe rank by finding the maximum gain of a block using the rank lists of qn and qo. A max-priority queue *PQ* is used to keep track of blocks that must be visited, where the key is Δ_b. Here, Δ_b is an overestimation of the gain of the objects in b. For any object $o \in b$, $\Delta_o \leq \Delta_b$, is computed in Line 2.10 as -

$$\Delta_b \leftarrow \frac{1}{|W_i|} \zeta(\hat{\boldsymbol{r}}^{\downarrow}(b,qn)) - \zeta(\hat{\boldsymbol{r}}^{\uparrow}(b,qo))$$

Recall that in the IRF index, each object o in the rank list is sorted in ascending order of the lower bound rank w.r.t.

Algorithm 2: BLOCK_SAFE_RANK

```
2.1  Input: $\hat{\rho}(o_{m+1},W_{i-1})$ - $(m+1)$·th highest popularity of $W_{i-1}$,
     $\hat{\rho}(o_m,W_{i-1}\backslash qo)$ - $m$·th highest popularity from $\mathcal{R}_{i-1}$ after updating for
     $qo$, and $PQ$ - a max-priority queue.
2.2  Output: Block based safe rank - BSR
2.3  $b \leftarrow$ first block in the rank list of $c_{qn}$.
2.4  $\Delta_b^{\uparrow} \leftarrow 0$
2.5  do
2.6      $\hat{r}^{\downarrow}(b,qn) \leftarrow r^{\downarrow}(o,c_{qn})$ of the first entry $o$ from $b$.
2.7      $d^{\uparrow}(b,c_{qo}) \leftarrow$ Maximum Euclidean distance between $b$, $c_{qo}$.
2.8      $b' \leftarrow$ First block position of $c_{qo}$, where $d^{\uparrow}(b,c_{qo}) \leq d^{\downarrow}(b',c_{qo})$.
2.9      $\hat{r}^{\uparrow}(b,qo) \leftarrow r^{\downarrow}(o',c_{qo})$ of the first entry $o'$ of $b'$.
2.10     $\Delta_b \leftarrow \frac{\zeta(\hat{r}^{\downarrow}(b,qn)) - \zeta(\hat{r}^{\uparrow}(b,qo))}{|W|}$
2.11     ENQUEUE $(PQ,b,\Delta_b)$
2.12     $\Delta_b^{\uparrow} \leftarrow \Delta_{top(PQ)}$
2.13     $b \leftarrow$ NEXT $(c_{qn})$
2.14 while $b$ cannot have a better gain than $\Delta_b^{\uparrow}$;
2.15 $BSR \leftarrow$ Compute from $\hat{\rho}(o_{m+1},W_{i-1})$,$\hat{\rho}(o_m,W_{i-1}\backslash qo)$, $\Delta_b^{\uparrow}$ as Eqn. 6.
2.16 RETURN BSR
```

Algorithm 3: OBJECT_SAFE_RANK

```
3.1  Input: $\hat{\rho}(o_{m+1},W_{i-1})$ - $(m+1)$·th highest popularity of $W_{i-1}$,
     $\hat{\rho}(o_m,W_{i-1}\backslash qo)$ - updated $m$·th highest popularity from $\mathcal{R}_{i-1}$ after
     removing $qo$, $PQ$ - a max-priority queue from BLOCK_SAFE_RANK.
3.2  Output: Object-level safe rank, OSR
3.3  while PQ not empty do
3.4      $E \leftarrow$ DEQUEUE $(PQ)$
3.5      if E is object then
3.6          $\Delta_o^{\uparrow} \leftarrow \Delta_E$; BREAK
3.7      else
3.8          for o in E do
3.9              $\Delta_o \leftarrow \frac{\zeta(\hat{r}(o,qn)) - \zeta(\hat{r}(o,qo))}{|W|}$
3.10             ENQUEUE $(PQ,o,\Delta_o)$
3.11 $OSR \leftarrow$ Compute from $\hat{\rho}(o_m,W_i)$,$\hat{\rho}(o_m,W_{i-1}\backslash qo)$,$\Delta_o^{\uparrow}$ (by Eqn. 5).
3.12 RETURN OSR
```

the cell c_{qn}, and the traversal starts from the beginning of the rank list of c_{qn} so that the objects with a higher gain are most likely to be explored first. The traversal continues until the subsequent blocks of the rank lists of qn cannot have a better gain than the current maximum gain $\Delta_b^{\uparrow}$ found so far. Here, the **terminating condition** of Line 2.14 is:

$$\zeta(\hat{\boldsymbol{r}}^{\downarrow}(b,qn))/|W_i| < \Delta_b^{\uparrow}$$

Lastly, in Line 2.15 the maximum gain value $\Delta_b^{\uparrow}$ is used to compute the block-level safe rank as follows:

$$\hat{\boldsymbol{\rho}}(o_m,W_{i-1}\backslash qo) + \frac{N - BSR + 1}{|W_i|} \geq \hat{\boldsymbol{\rho}}(o_{m+1},W_{i-1}) + \Delta_b^{\uparrow} \quad (6)$$

3) Object-level safe rank: If the rank of any object $o \in \mathcal{R}_{i-1}$ for qn is not smaller (better) than the block-level safe rank *BSR*, then the object-level safe rank is computed, where $OSR \geq BSR$ is used to further determine whether the result needs to updated or not (Lines 1.13 - 1.18 in Algorithm 1).

Algorithm 3 shows a best-first approach to compute the maximum object level gain $\Delta_o^{\uparrow}$, using the same priority queue *PQ* maintained in the block-level computation. In each iteration, the top element E of *PQ* is dequeued. If E is a block, the approximate rank of each $o \in E$ for qn and qo is computed using the lower bound rank in the corresponding rank lists. The objects are then enqueued in *PQ* according to the gain computed using Eqn. 4. If E is an object, then the gain is returned as $\Delta_o^{\uparrow}$ (Lines 3.5 - 3.6). The object-level safe rank, *OSR*, is then computed in the same manner as Eqn. 6 with $\Delta_o^{\uparrow}$.

D. Validation objects

If the rank of any object $o \in \mathcal{R}_{i-1}$ is not safe, a set of validation objects VO is found such that, as long as $\forall vo \in VO$, $\hat{\rho}(o, W_i) \geq \hat{\rho}(vo, W_i)$, o is a valid result object of $\mathcal{R}_i$. We present an efficient approach to incrementally identify VO. Furthermore, we show that if the result needs to be updated, the new result objects also must come from VO.

First, after a new query qn arrives, the approximate rank for each object $o \in \mathcal{R}_{i-1}$ is computed, and the appropriate popularity scores are updated. Let the updated m-th highest approximate popularity from $\mathcal{R}_{i-1}$ be $\hat{\rho}(o_m, W_i)$ (Line 1.14 of Algorithm 1). The priority queue PQ maintained for safe rank computation is used to find the set VO of validation objects, where $\hat{\rho}(o_m, W_i)$ is used as a threshold to terminate the search.

A best-first search is performed using PQ to find the objects that have gain high enough to be a result. Specifically, if the dequeued element E from PQ is a block, the $\hat{r}$ of each object o in E is computed for qn and qo in the same manner as described for the object-level safe rank computation. As the popularity of an object $o \in O \backslash \mathcal{R}_{i-1}$ can be at most $\hat{\rho}(o_{m+1}, W_{i-1}) + \Delta_o$, an object o is included in the validation set if o satisfies the following condition:

$$\hat{\rho}(o_{m+1}, W_{i-1}) + \Delta_o \geq \hat{\rho}(o_m, W_i) \tag{7}$$

As PQ is a max-priority queue which is maintained for the gain of the objects and the blocks, the process can be safely terminated when the gain of a dequeued element E does not satisfy the condition in Equation 7. If no validation object is found, that means there is no object that can have a higher popularity than the current results, so the result set $\mathcal{R}_{i-1}$ (of previous window W_{i-1}) is the result of current window W_i. Otherwise, the popularity of each object in $\mathcal{R}_{i-1} \backslash \mathcal{R}_i$ needs to be checked against the popularity of the validation objects $vo \in VO$ to update the result.

1) Updating results: As described in Section V-D, the set of validation objects VO is computed such that no object $o \backslash VO$ can have a higher popularity than any of the objects in $\mathcal{R}_{i-1}$. Therefore, only objects in VO are considered when updating the result set. To update the results using the objects $vo \in VO$, the popularity of vo for the current window must be computed. Therefore, an efficient technique to compute the popularity of the validation objects is now presented.

Computing $\hat{\rho}$ of the validation objects. As the popularity gain of each $vo \in VO$ has already been computed as described in Section V-D, it is sufficient to find the $\hat{\rho}(vo, W_{i-1} \backslash qo)$ and use it to compute $\hat{\rho}(vo, W_i)$. Since the popularity of every object for every window is not computed, a straightforward way to compute $\hat{\rho}(vo, W_{i-1} \backslash qo)$ is to find the rank of vo for each $q \in W_{i-1} \backslash qo$ using the corresponding rank lists. However, this approach is computationally expensive, especially when the window size is large. Moreover, if vo was a validation object or a result object in a prior window W_{i-y}, then the same computations are repeated unnecessarily for the queries shared by the windows (the queries contained in $W_i \cap W_{i-y}$).

Therefore, if $\hat{\rho}$ of a result or a validation object is computed for a window W_{i-y}, the aim is to reuse this computation for later windows in an efficient way. This can be accomplished by storing the popularity of a subset of "necessary" objects from prior windows for later reuse. We show that the choice of these limited number of windows is optimal, and storing the popularity for any additional windows cannot reduce the computational cost any further.

Choosing the limited number of prior windows. The popularity computations can be reused if the number of shared queries among the windows is greater than the number of queries that differ. Otherwise, the popularity must be computed for the window W_i from scratch rather than reusing the popularity computations from W_{i-y}. Specifically, let Y be the number of shared queries among windows W_i, W_{i-y} ($Y = |W_i \cap W_{i-y}|$), $Q_o = W_{i-y} \backslash W_i$, and $Q_n = W_i \backslash W_{i-y}$. So in a count based window, $|Q_n| = |W_i| - Y$ and $|Q_o| = |W_i| - Y$, as each time a new query is inserted, the least recent query is removed from the window. If the number of computations required for the shared queries is greater than the number of computations for $|Q_n| + |Q_o|$, i.e., $Y \geq 2(|W_i| - Y)$, then computations can be reused. So the number of shared queries, Y, should be greater than or equal to $2|W_i|/3$ for efficient reuse.

Reusing popularity computations. If the condition $Y \geq 2|W_i|/3$ holds, the popularity of an object o computed for W_{i-y} can be used to compute $\hat{\rho}(o, W_i)$ as follows:

$$\hat{\rho}(o, W_i) = \hat{\rho}(o, W_{i-y}) + \frac{\sum_{qn \in Q_n} \zeta(\hat{r}(o, qn)) - \sum_{qo \in Q_o} \zeta(\hat{r}(o, qo))}{|W_i|} \tag{8}$$

Popularity lookup table. A popularity lookup table is maintained with the popularity of the result and validation objects for the most recent $2|W_i|/3$ windows. If a validation object vo of the current window W_i is found in the lookup table, the popularity is computed using Equation 8. Otherwise, the popularity of vo is computed from the rank lists of the queries in W_i. The popularity vo for W_i is then added to the popularity lookup table for later windows.

Obtaining Results. The objects $vo \in VO$ are considered one by one to update the results. After computing the popularity of an object $vo \in VO$, if $\hat{\rho}(vo, W_i) > \hat{\rho}(o_m, W_i)$, then vo is added to $\mathcal{R}_i$. The set $\mathcal{R}_i$ is adjusted such that it contains m objects with the highest $\hat{\rho}$, and the value of $\hat{\rho}(o_m, W_i)$ is adjusted accordingly. During this process, if the overestimated popularity of an object vo computed with Eqn 7 is less than the updated $\hat{\rho}(o_m, W_i)$, that object can be safely discarded from consideration without computing its popularity.

E. Approximation error bound

Here, we formalize the approximation error bound of our proposed approach. For any object $o \in O$, and any window W of queries, the ratio between $\hat{\rho}(o, W)$ and $\rho(o, W)$ is bounded.

Lemma 3. *For any object $o \in O$, and a window W of queries, the approximation ratio is bounded by $1 - \varepsilon/2N$.*
(i) $\hat{\rho}(o, W)/\rho(o, W) \leq 1 - \varepsilon/2N$ when $\rho(o, W) \geq \hat{\rho}(o, W)$; and (ii) $\rho(o, W)/\hat{\rho}(o, W) \leq 1 - \varepsilon/2N$ when $\hat{\rho}(o, W) \geq \rho(o, W)$ always holds.

Proof: See extended ArXiV version [28] for proof. ∎

F. *Extending the solution for time based window*

Given a fixed time interval, a time-based sliding window W contains all of the queries that have arrived within the most recent interval. In contrast to a count based window, the number of new queries included, and the number of queries excluded from a time based window can vary at each interval. Let the set of new queries included be Q'_n and the set of queries excluded be Q'_o for a time based window. Then, the following modifications of the solution presented in Sec. V can be made to support this window type:

1) Similar to a count based window, if the aggregated rank estimation of a result object o is high enough after a shift of the sliding window such that no other object can have a higher popularity, o must remain as a result object. Therefore, the popularity gain and the safe rank needs to be computed for the sets Q'_n and Q'_o for the queries each time the sliding window shifts. For example, the object level gain is computed as: $\Delta_o = \frac{1}{|W_i|}\zeta(\sum_{qn\in Q'_n}\hat{r}(o,qn)) - \zeta(\sum_{qo\in Q'_o}\hat{r}(o,qo))$
2) If the current result objects are not safe, we need to find the validation objects. Similar to a count based window, a best-first search can be performed to find the objects that can have a higher popularity than the current results, where the gain values are computed for the sets Q'_n and Q'_o.

Note that, although the solution can be extended for a time based window, the same approximation bound is not applicable as the window size varies at each interval. We leave this and other window-based variations of our solution to future work.

VI. Experimental Evaluation

Here, we present the experimental evaluation for our solutions to monitor the top-m popular objects in a sliding window of streaming queries. As there is no prior work that directly answers this problem (Sec. II), we compare our approximate solution (proposed in Sec. V), denoted by AP, with the baseline exact approach (proposed in Sec. III-B), denoted by BS.

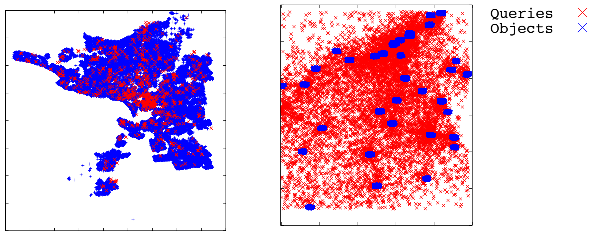
A. *Experiment Settings*

All algorithms were implemented in C++. Experiments were ran on a 24 core Intel Xeon $E5-2630$ 2.3 GHz using 256 GB RAM, and 1TB 6G SAS 7.2K rpm SFF (2.5-inch) SC Midline disk drives. All index structures are memory resident.

Datasets and query generation. All experiments were conducted using two real datasets, (i) Melb dataset at a city scale and (ii) Foursq[2] dataset at a country scale.

The Melb dataset contains 52,913 real estate properties sold in Melbourne in 2013-2015, collected from the real estate advertising site[3]. The locations of the queries were created by using locations of 987 facilities (train stations, schools, hospitals, supermarkets, and shopping centers) in this city. We generated two sets of queries from these locations, each of size $20K$. Repeating queries were created using two different approaches: (i) uniform (U); and (ii) skewed (S) distribution, respectively. The radius of the queries are varied as an experimental parameter, and is discussed further in Sec. VI-B.

[2]https://sites.google.com/site/yangdingqi/home/foursquare-dataset
[3]http://www.realestate.com.au



(a) Melb (b) Foursq
Fig. 3: Dataset and query locations

The Foursq dataset contains 304,133 points of interest (POI) from Foursquare[4] in 34 cities spread across USA. From the check-ins of each user, we generated a query whose location was the centroid of all check-ins of that user, and the query radius was set to the minimum distance that covers these check-ins. If a user had only one check-in, we set the query location as the check-in location, and the radius of the query is randomly assigned from another user. As a result, a total of 22,442 queries were generated for the Foursq dataset.

Since the Melb dataset represents real city-level data, and queries are real facilities in that city, it is the best candidate for an effectiveness study. So we performed efficiency and effectiveness studies using Melb. Since the queries generated for Foursq are spread over a much wider area, it is more suitable for the efficiency and scalability study. Nevertheless, we also used Foursq to validate our effectiveness study.

After initializing the sliding window, we evaluated the performance of both BS and AP for $10K$ query arrivals in the stream, using $10K$ shifts of the sliding window. We repeated the process 50 times and reported the mean performance. For the Melb dataset, the arrival order of queries was randomly generated, and for the Foursq dataset, the order was obtained from the most recent check-in time of the corresponding user. Figure 3 shows the location distribution of the objects and the queries for both datasets, where the blue (red) points represent object (query) locations. Note that, for the Foursq dataset, the POIs are clustered in large cities (blue clusters). As a user may check-in at different cities, the queries (which are the centroid of the check-in locations) are distributed in different locations across USA. The index construction time for different values of ε are shown in Table III.

TABLE III: Index construction time (min)

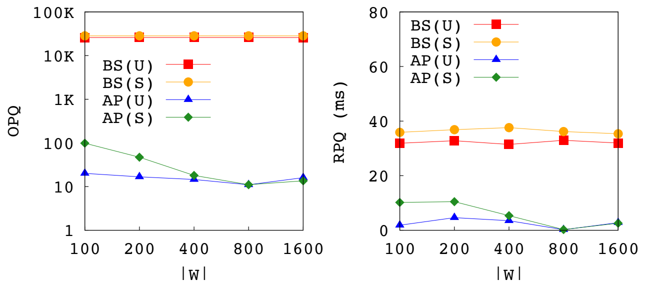
ε	**Melb**	**Foursq**
1	184	1201
2	176	1096
3	130	971
4	67	511
5	65	487

TABLE IV: Parameters

Parameter	**Range**
W	100, 200, **400**, 800, 1600
m	1, 5, **10**, 20, 50, 100
Query radius (%)	1, 2, **4**, 8, 16
ε	1, 2, **3**, 4, 5
B	32, 64, **128**, 256, 512

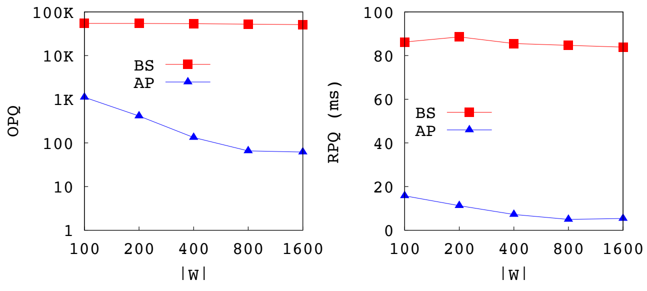
Evaluation & Parameterization. We studied the efficiency, scalability and effectiveness for both the baseline approach (BS), and the approximate approach (AP) by varying several

[4]https://foursquare.com



(a) Objects computed (b) Runtime

Fig. 4: Effect of varying $|W|$ on Melb dataset



(a) Objects computed (b) Runtime

Fig. 5: Effect of varying $|W|$ on Foursq dataset

parameters. The parameters and their ranges are listed in Table IV, where the values in bold represent the default values. For all experiments, a single parameter varied while keeping the rest as the default settings. For efficiency and scalability, we studied the impact of each parameter on: the number of objects whose popularity are computed per query (OPQ), to update the answer of T$m\rho$Q; and the runtime per query (RPQ).

In order to measure the *effectiveness* of our approximate approach, the impact of each parameter on the following two metrics were studied:

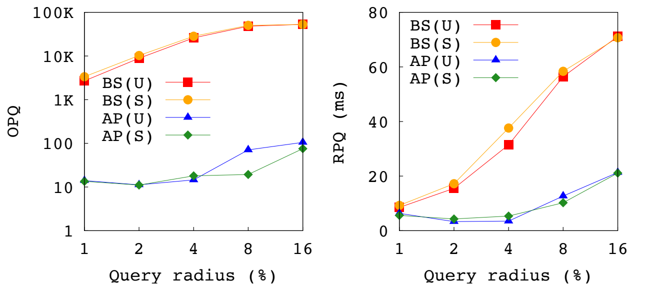
1) **Approximation ratio**: For a window W, for each $o_i \in \mathcal{R}$, $o'_i \in \hat{\mathcal{R}}$, where i is the corresponding position of the object in top-m results, we compute the approximation ratio as: $ratio = max(\hat{\rho}(o'_i, W)/\rho(o_i, W), \rho(o_i, W)/\hat{\rho}(o'_i, W))$. We report the average approximation ratio of the sliding window by varying different parameters. As the popularity of an object is an aggregation over the estimated ranks, the approximation ratio may not be "1" (the best approximation ratio) even if the approximate result list $\hat{\mathcal{R}}$ is exactly the same as that result list returned by the baseline. Therefore, we present the following metric to demonstrate the similarity of the approximate result object lists with the baseline.
2) **Percentage of result overlap**: For a window W, let $|\mathcal{R}| = |O|$, where $\mathcal{R}$ is the sorted list of all of the objects according to their exact popularity. We report the similarity between the result list returned by the approximate approach, $\hat{\mathcal{R}}$ with $\mathcal{R}$ at different depths. Specifically, for each result object $o'_i \in \hat{\mathcal{R}}$, where $|\hat{\mathcal{R}}| = m$, we record the percentage of objects in $\hat{\mathcal{R}}$, overlapping with the top-k objects of $\mathcal{R}$, where k is varied from 10 to 200. For instance, when $m = 50$, we compute how many objects in the top-50 approximate result also appear in the top-50, top-75, ..., top-150 exact results. We report the percentage of the shared objects for different choices of k, averaged by 10,000 shifts of the sliding window.

B. Efficiency & Scalability Evaluation

Varying $|W|$. Figure 4 and Figure 5 show the impact of varying the number of queries in the sliding window, $|W|$, for Melb and Foursq, respectively. For Melb, the experiments were conducted using uniform and skewed query sets, while the Foursq query set is derived directly from user check-ins.

For both datasets, the number of popularity computations required by the approximate approach AP is about 3 orders of magnitude less than the baseline BS. The reason is two-fold: (i) In AP, we compute the popularity of only the objects necessary to update the result. If the previous result objects are found as valid, we do not need to compute the popularity of any additional object. In contrast, the baseline BS must update the popularity for all objects that satisfy the query constraint. (ii) Since the popularity function is an average aggregation (see Sec. III), the popularity of an object usually does not change drastically as $|W|$ increases. So, the result objects in a window are more likely to stay valid in subsequent windows for larger values of $|W|$, thereby requiring even fewer objects being checked. As shown in Figures 4b and 5b, fewer popularity computation leads to lower running time.

In Melb, the performance of AP in both uniform and skewed query sets improves as $|W|$ increases, but drops slightly from $|W| = 800$ to 1600. This is because, if the results are *not valid* for a window, we need to check the validation objects, which is a subset of the objects that satisfy the constraint of at least one query in the current window. So although the results update less often for a larger $|W|$, an update in the results may require checking more objects for a larger $|W|$.

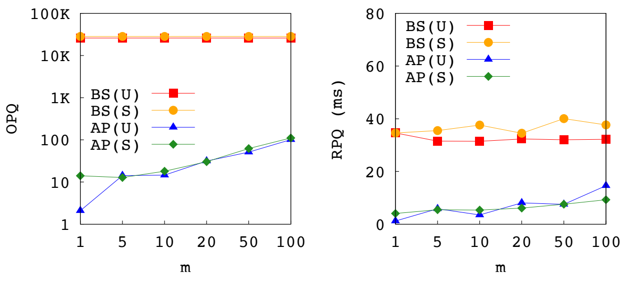


(a) Objects computed (b) Runtime

Fig. 6: Effect of varying query radius on Melb dataset

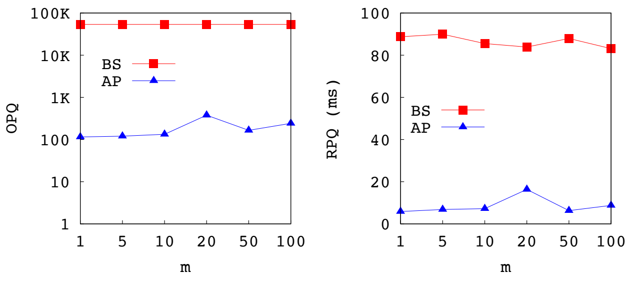
Varying query range. Figure 6 shows the performance when varying the radius of each query as a percentage of the dataspace. We vary the query radius only for Melb, as we use the radius that covers the check-in locations of a user as the query radius for Foursq. Here, the number of objects that fall in the query range grows as query radius increases. Therefore, the performance of the baseline declines rapidly when the radius increases. In contrast, the approximate approach computes the popularity of only the objects that can be a result, which is a subset of the objects that fall within the query range. Thus, the approximate approach outperforms the baseline, and the benefit is more significant as the query radius increases.

Varying m. The experimental results when varying the number of result objects, m, are shown in Figure 7 and Figure 8 for Melb and Foursq, respectively. Here, the performance of the baseline does not vary much, as it computes the popularity for all of the objects that fall in the query range regardless



(a) Objects computed (b) Runtime

Fig. 7: Effect of varying m on Melb dataset



(a) Objects computed (b) Runtime

Fig. 8: Effect of varying m on Foursq dataset

of the value of m. The approximate approach outperforms the baseline, because it considers only the objects that can potentially be in the top-m results. As more objects qualify to be a result, the performance of the approximate approach decreases with the increase of m.

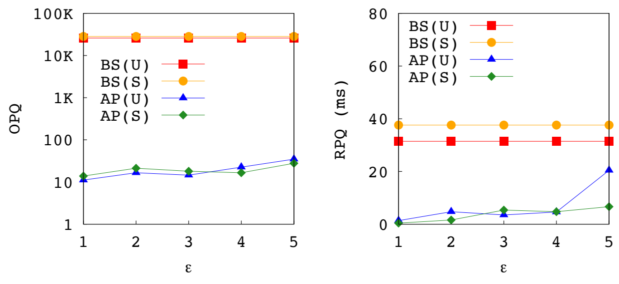
Varying ε. Figure 9 shows the performance when varying the approximation parameter ε for Melb dataset. The approximate approach AP consistently outperforms the baseline for all choices of ε. As the rank of an object is more accurately approximated for a smaller value of ε, it leads to checking fewer number of objects and a lower runtime. As a result, the performance of AP gradually decreases with the increase of ε.

Varying B. By varying the block size B of the rank lists, we find that the number of objects to check does not vary with B. If the result of a window needs to be updated, the same set of validation objects are retrieved regardless of the rank list block size. So we only show the runtime for varying B in Figure 10a. For each B, the total runtime is shown as a breakdown of the computation time for (i) block-level safe rank, (ii) object-level safe rank, and (iii) validation object computation for both uniform and skewed query sets. From Figure 10a we conclude that: (1) as the total number of blocks decreases for higher B, the time required to compute the block-level safe rank also decreases; (2) the validation object lookups dominate the computational costs of the approximate solution.

C. *Effectiveness Evaluation*

Varying $|W|$. Table V shows the average approximation ratio for both datasets. Although the ratio gradually improves as $|W|$ increases, the change does not follow any obvious pattern. The explanation for this random behaviour is that popularity is an average aggregation of $|W|$ ranks, so if both the exact and approximate popularity do not change at the same rate with $|W|$, their ratios do not change in a fixed way.

Varying m. As shown in Figure 3, the query locations originally follow a skewed distribution, and most query locations



(a) Objects computed (b) Runtime

Fig. 9: Effect of varying ε on Melb dataset

are clustered in a small area (which is the central business district of that city), while the rest are scattered regionally for Melb dataset. In the uniform query set, the queries are repeated uniformly, thus the upsized query set also follows the same (skewed) distribution of the original query set. For this reason, we evaluated our effectiveness as a percentage of result overlap (between the top-m approximate results and the top-k exact results) when using the uniformly upsized query set to capture a more realistic scenario.

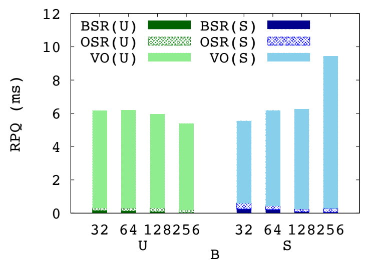
The overlap percentage for Melb dataset is shown in Figure 10b, where k ranges from 10 to 200 and we set three choices of m (10, 50, 100). As k increases, the overlap percentage also increases. For $m = 50$ and 100, the overlap percentage quickly reaches 90% when $k = 50$. Note that, if multiple objects have the same popularity value, we treat their rank position in the result as equivalent.

Figure 11a shows the overlap percentage for the Foursq dataset. Although the overlap approaches 100% for higher m, the overlap is not as high as in the Melb dataset for lower m. The reason is as follows. As shown in Figure 3 the objects in Foursq are clustered into cities, where the query locations are distributed all over the dataspace. Therefore, the popularity values of most of the objects in a city are very close to each other. Figure 11b shows a screenshot of the top-10 popularities computed in the baseline at three example instances. As we can see, the final rank of two objects can be very far away for a slight difference in their popularity values; for example, in the first instance, the difference between every adjacent objects' popularity is only 0.25 on average, while the absolute values are at the scale of 50K.

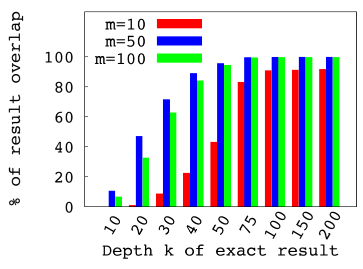
Table VI shows the approximation ratio when varying m in both datasets. We find that the approximation ratio keeps improving as m increases, since most of the objects in the top-m ranked list have very similar scores in both their exact and approximate popularity when $m < 100$.

Varying query range. The approximation ratio of the results w.r.t. varying query ranges is shown in Table VII. The approximation ratio does not indicate any obvious patterns since the approximation computation does not depend on the query radius, or the number of objects falling in that range.

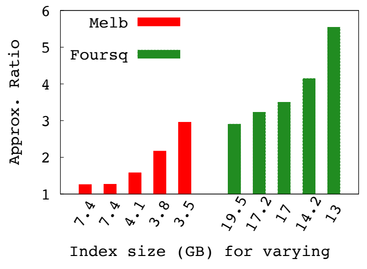
Varying ε, space vs. effectiveness tradeoff. Figure 10c shows the tradeoff between the space requirement and the effectiveness as approximation ratio for varying ε. Here, the x-axis represents the index size in GB for both datasets, where ε is varied from 1 to 5. Since the approximate popularity of an object becomes closer to the exact popularity as ε decreases, the approximation ratio also improves for smaller ε.



(a) Effect of varying B



(b) % of result overlap for varying m



(c) Index size vs. approx. ratio for varying ε

Fig. 10: Experiments on Melb dataset

Dataset \ $\|W\|$		100	200	400	800	1600
Melb	U	2.12	1.60	1.57	1.67	1.55
	S	3.19	1.55	2.14	1.30	1.34
Foursq		2.76	6.87	3.49	3.15	2.33

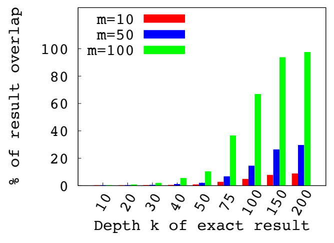
TABLE V: Approximation ratio for varying $|W|$

Dataset \ m		1	5	10	20	50	100
Melb	U	3.00	4.79	1.57	1.56	1.49	1.49
	S	5.61	2.03	2.14	1.60	1.17	1.16
Foursq		1.57	2.62	2.68	3.31	3.37	2.47

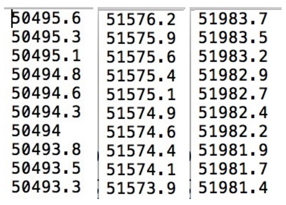
TABLE VI: Approximation ratio for varying m

Dataset \ Radius		1	2	4	8	16
Melb	U	2.55	1.55	1.57	1.61	1.63
	S	3.32	2.88	2.14	2.39	3.55

TABLE VII: Approximation ratio for varying query radius



(a) % of result overlap



```
50495.6   51576.2   51983.7
50495.3   51575.9   51983.5
50495.1   51575.6   51983.2
50494.8   51575.4   51982.9
50494.6   51575.1   51982.7
50494.3   51574.9   51982.4
50494     51574.6   51982.2
50493.8   51574.4   51981.9
50493.5   51574.1   51981.7
50493.3   51573.9   51981.4
```

(b) Popularity of top-10 objects

Fig. 11: Effectiveness study for varying m in Foursq dataset

VII. Conclusion

In this paper, we proposed the problem of top-m rank aggregation of spatial objects for streaming queries. We have shown how to bound the rank of an object for any unseen query, and proposed an approximate solution with a guaranteed error bound. We presented *safe rank* to determine whether the current result is still valid when new queries arrive, and *validation objects* to limit the number of objects to update in the results. Experiments on two real datasets have shown that the approximate approach is about 3 orders of magnitude more efficient than the exact solution, while having over 90% overlap with the results of exact solution when $m > 50$.

Acknowledgment. This work was supported by Australian Research Council's *Discovery Projects* (DP140101587), partially by ARC DP170102726, DP170102231, and National Natural Science Foundation of China (NSFC) 91646204. Shane Culpepper is the recipient of an ARC DECRA Research Fellowship (DE140100275). Farhana Choudhury is a scholarship recipient from National ICT Australia.

References

[1] R. Fagin, R. Kumar, and D. Sivakumar, "Efficient similarity search and classification via rank aggregation," in *SIGMOD*, 2003, pp. 301–312.

[2] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar, "Rank aggregation methods for the web," in *WWW*, 2001, pp. 613–622.

[3] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," in *PODS*, 2001, pp. 102–113.

[4] N. Mamoulis, K. H. Cheng, M. L. Yiu, and D. W. Cheung, "Efficient aggregation of ranked inputs," in *ICDE*, 2006, pp. 72–84.

[5] N. Ailon, M. Charikar, and A. Newman, "Aggregating inconsistent information: ranking and clustering." *JACM*, vol. 55, no. 5, p. 23, 2008.

[6] S. Shekhar, S. K. Feiner, and W. G. Aref, "Spatial computing." *Comm. of the ACM*, vol. 59, no. 1, pp. 72–81, 2016.

[7] Y. Tao, V. Hristidis, D. Papadias, and Y. Papakonstantinou, "Branch-and-bound processing of ranked queries," *Inf. Syst.*, vol. 32, no. 3, pp. 424–445, 2007.

[8] M. Li, Z. Bao, T. Sellis, and S. Yan, "Visualization-aided exploration of the real estate data," in *ADC*, 2016, pp. 435–439.

[9] K. Mouratidis, S. Bakiras, and D. Papadias, "Continuous monitoring of top-k queries over sliding windows," in *SIGMOD*, 2006, pp. 635–646.

[10] G. Cormode and M. Hadjieleftheriou, "Finding frequent items in data streams," *PVLDB*, vol. 1, no. 2, pp. 1530–1541, 2008.

[11] O. Papapetrou, M. Garofalakis, and A. Deligiannakis, "Sketch-based querying of distributed sliding-window data streams," *PVLDB*, vol. 5, no. 10, pp. 992–1003, 2012.

[12] C. Bohm, B. C. Ooi, C. Plant, and Y. Yan, "Efficiently processing continuous k-nn queries on data streams," in *ICDE*, 2007, pp. 156–165.

[13] F. Korn, S. Muthukrishnan, and D. Srivastava, "Reverse nearest neighbor aggregates over data streams," in *PVLDB*, 2002, pp. 814–825.

[14] C. Li, Y. Gu, J. Qi, G. Yu, R. Zhang, and W. Yi, "Processing moving knn queries using influential neighbor sets," *PVLDB*, vol. 8, no. 2, pp. 113–124, 2014.

[15] M. Cheema, W. Zhang, X. Lin, Y. Zhang, and X. Li, "Continuous reverse k nearest neighbors queries in euclidean space and in spatial networks," *VLDB Journal*, vol. 21, no. 1, pp. 69–95, 2012.

[16] T. Xia, D. Zhang, E. Kanoulas, and Y. Du, "On computing top-t most influential spatial sites," in *PVLDB*, 2005, pp. 946–957.

[17] C.-L. Li, E. T. Wang, G.-J. Huang, and A. L. P. Chen, "Top-n query processing in spatial databases considering bi-chromatic reverse k-nearest neighbors," *Inf. Syst.*, vol. 42, pp. 123–138, 2014.

[18] L. Zhan, Y. Zhang, W. Zhang, and X. Lin, "Finding top k most influential spatial facilities over uncertain objects," in *CIKM*, 2012, pp. 922–931.

[19] M. A. Cheema, L. Brankovic, X. Lin, W. Zhang, and W. Wang, "Multi-guarded safe zone: An effective technique to monitor moving circular range queries," in *ICDE*, 2010, pp. 189–200.

[20] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou, "Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring," in *SIGMOD*, 2005, pp. 634–645.

[21] K. Pripužić, I. P. Žarko, and K. Aberer, "Top-k/w publish/subscribe: A publish/subscribe model for continuous top-k processing over data streams," *Inf. Syst.*, vol. 39, pp. 256 – 276, 2014.

[22] J.-L. Koh, C.-Y. Lin, and A. P. Chen, "Finding k most favorite products based on reverse top-t queries," *PVLDB*, vol. 23, no. 4, pp. 541–564, 2014.

[23] A. Vlachou, C. Doulkeridis, K. Nørvåg, and Y. Kotidis, "Identifying the most influential data objects with reverse top-k queries," *PVLDB*, vol. 3, no. 1-2, pp. 364–372, 2010.

[24] R. C.-W. Wong, M. T. Özsu, P. S. Yu, A. W.-C. Fu, and L. Liu, "Efficient method for maximizing bichromatic reverse nearest neighbor," *PVLDB*, vol. 2, no. 1, pp. 1126–1137, 2009.

[25] Z. Zhou, W. Wu, X. Li, M. L. Lee, and W. Hsu, "MaxFirst for MaxBRkNN," in *ICDE*, 2011, pp. 828–839.

[26] D. Yan, R. C.-W. Wong, and W. Ng, "Efficient methods for finding influential locations with adaptive grids," in *CIKM*, 2011, pp. 1475–1484.

[27] O. Gkorgkas, A. Vlachou, C. Doulkeridis, and K. Nørvåg, "Discovering influential data objects over time," in *SSTD*, 2013, pp. 110–127.

[28] F. M. Choudhury, Z. Bao, J. S. Culpepper, and T. Sellis, "Monitoring the top-m aggregation in a sliding window of spatial queries," 2016. [Online]. Available: https://arxiv.org/abs/1610.03579