Loyalty-based Retrieval of Objects That Satisfy Criteria Persistently

Zhitao Shen Muhammad Aamir Cheema Xuemin Lin

University of New South Wales, Australia
{shenz, macheema, lxue}@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-201224
August 2012

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

A traditional query returns a set of objects that satisfy user defined criteria at the time query was issued. The results are based on the values of objects at query time and may be affected by outliers. Intuitively, an object better meets the user's needs if it persistently satisfies the criteria, i.e., it satisfies the criteria for majority of the time in the past T time units. In this paper, we propose a measure named *loyalty* that reflects how persistently an object satisfies the criteria. Formally, the loyalty of an object is the total time (in past T time units) it satisfies the query criteria. In this paper, we study top-k loyalty queries over sliding windows that continuously report k objects with the highest loyalties. Each object issues an update when it starts satisfying the criteria or when it stops satisfying the criteria. We show that the lower bound cost of updating the results of a top-k loyalty query is $O(\log N)$, for each object update, where N is the number of updates issued in last T time units. We conduct a detailed complexity analysis and show that our proposed algorithm is optimal. Moreover, effective pruning techniques are proposed that not only reduce the communication cost of the system but also improve the efficiency. We experimentally verify the effectiveness of the proposed approach by comparing it with a classic sweep line algorithm.

1 Introduction

A traditional query Q returns every object that satisfies the query criteria at the time t query was issued. The traditional queries do not consider the history of the objects' values, i.e., the values of objects in the recent past. Hence, the traditional queries fail to capture how persistently an object satisfies the query criteria. Consider the example of a stock broker who issues a query at time t to retrieve the profitable stocks. He may define a set of criterions to denote the profitability. A traditional query returns every stock s that satisfies the criterions at time t. Although a returned stock s meets the criteria at time t, the history of the stock s may indicate that it usually does not satisfy the criteria and is not a good choice for investment. Hence, a query that does not take into account the history of stock items is not suitable.

To address the above mentioned problem, in this paper, we propose a new query operator called *loyalty queries*. A loyalty query considers how persistently the objects satisfy the query criteria. Consider a *traditional query* Q that defines a set of criterions. Let $Q(o, t)$ denote whether an object o satisfies the criteria of query Q at time t or not. More specifically, $Q(o, t)$ is true if and only if the object o satisfies the query criteria at time t. Let T be a user defined parameter. The loyalty of an object o is the total time duration for which $Q(o, t)$ is true within last T time units. The measure is called "loyalty" because it signifies how persistently the object o meets the criteria in the recent past. In this paper, we study continuous *top-k* loyalty queries that continuously report k objects with the highest loyalties. We also show that the proposed approach can be easily used to answer threshold loyalty queries that return every object with loyalty greater than a given threshold.

Loyalty queries have many interesting applications in different areas such as location based services, wireless sensor network, stock market, traffic monitoring, and internet applications, etc. For instance, in the example of the stocks, the stock broker may retrieve top-k loyal objects to retrieve better options for investment. Consider another example of a paid parking system that notifies the nearby cars of its availability, i.e., the cars that are in its *influence zone* [6] or the cars that are within 1 Km of the parking space [5]. At a given time t, the system may send SMS to some cars that satisfy the criteria (e.g. a car that lies within 1Km at time t). However, most of such cars may just be passing through that area and may not be interested in parking. On the other hand, a car that satisfies the criteria for majority of the time in recent past may actually be looking for the parking. Hence, the system may use top-k loyalty queries to send notifications to such cars.

Consider another example of a wireless sensor network. An environmental scientist may be interested in monitoring the most rainy sites. A traditional query selects every sensor that reports rain at a given time t. Clearly, the query may miss a site s that is usually the rainiest but it is not raining there at time t. Moreover, the results are also affected by an erroneous reading by a sensor at time t. For these reasons, a top-k loyalty query is a more feasible tool to retrieve the rainiest sites.

We next summarize our contributions in this paper.

- **Novel query operator**. To the best of our knowledge, we are the first to study continuous loyalty queries. In this paper, we formalize the definition of loyalty queries and present a framework that efficiently solves the loyalty queries.
- **Continuous updates.** We study the problem in a continuous time domain where the updated results are reported as soon as the results change as opposed to the

time-stamp model where the results are updated after every u time units. Note that the time-stamp model suffers from either high computational cost or low accuracy. More specifically, if u is small, the computation cost increases because the results are to be updated more often. On the other hand, if u is large, the accuracy is reduced because the results may have become invalid between two successive time-stamps. The continuous updates provided by our algorithm do not have these limitations.

- **Optimal computation cost**. An object issues an update if it starts satisfying the query criteria or if it stops satisfying the query criteria. Note that the top-k loyal objects may change whenever an object issues an update. Let N be the total number of object updates issued in the last T time units. Upon receiving an object update, our algorithm updates the top-k loyal objects in $O(\log N)$. We prove that this is the lower bound update cost for top-k loyalty queries, hence our algorithm is optimal.

- **Low communication cost**. In distributed environment, the updates of an object are generated locally and sent to a centric server for query processing. We obverse that some updates do not contribute to computing the final results of the loyalty queries. These updates are so called *trivial* updates. We further develop an efficient pruning technique on the trivial updates to further reduce the communication cost and the overall computation cost as well.

- **Extensive evaluation and analysis**. We theoretically analyse the complexity of our algorithm and prove that it meets the lower bound cost. We also conduct experiments to show the effectiveness and the efficiency of our proposed approach. We compare our algorithm with the Bentley-Ottmann sweep line algorithm [3]. For N object updates, the total cost of the Bentley-Ottmann algorithm is $O(N^2 \log N)$ in the worst case. In contrast, the total worst case cost of our algorithm is $O(N \log N)$. Extensive experiments conducted on both real and synthetic data sets demonstrate that our proposed approach is an order of magnitude faster than the Bentley-Ottmann algorithm.

The remainder of the paper is organized as follows. In Section 2, we give an overview of the related work and formalized definition of loyalty queries. We introduce our framework in Section 3, while in Section 4 we present our solution to the top-k loyalty queries. The techniques of the threshold queries are presented in Section 5. The experimental results are reported in Section 6. Section 7 concludes the paper.

2 Background

In this section, we first present an overview of the related work. Then, we formally define the problem studied in this paper.

2.1 Related Work

Sweep line algorithms

The Bentley-Ottmann algorithm is a sweep line algorithm for reporting all intersections between all line segments in the plane. The algorithm is initially proposed by Bentley

and Ottmann [3] and discussed in detail by Preparata and Shamos [22]. Consider a vertical sweep line, first placed at the extreme left of the plane. Then, it will move to the right by jumping between endpoints of the line segments and intersections. The algorithm maintains the vertical ordering of the line segments intersecting the vertical line. An event is created when two adjacent line segments on the vertical line will possibly intersect in the future, namely, the two line segments will possibly exchange their vertical ordering. An event queue is organized for processing the future events. The Bentley-Ottmann algorithm can be used to retrieve the top-k loyalty objects as it always keeps the total ordering of the line segments. Our proposed algorithm also uses a sweep line approach. However, we create less events. The total cost of the Bentley-Ottmann algorithm is $O((N + M) \log N)$, where N is the number of line segments and M is the number of events (intersections). In the worst case, M can be $O(N^2)$. We improve the complexity of solving the top-k loyalty queries and the total cost of our proposed algorithm is $O(N \log N)$.

Li et al [16] present a solution to find top-k objects on temporal data. They use a B-tree based indexing structure for the historical data. Top-k objects are efficiently answered based on the index. Tao et al [25] study the problem of processing spatial-temporal window aggregation queries over historical data. However, such offline algorithms [16, 25] cannot be utilized to efficiently solve our problem because the loyalty queries report the results on the fly and it is not efficient to build the index for online processing.

For $k = 1$, the top-k loyalty query is equivalent to finding the upper envelope [13] of N line segments in the plane. The upper envelope computation can be done in $O(N \log N)$. Kinetic data structures [2, 23] can also be used to find the upper envelop in a sweep-line fashion. However, it is non-trivial to extend the existing variants of kinetic data structures such as the kinetic heap or the kinetic tournament to support the top-k objects queries. Moreover, our proposed algorithm is theoretically more efficient even when $k = 1$. As it is necessary to maintain a priority queue for scheduling the events in the continuous time domain for these data structures, the total cost of the kinetic heap is $O(N \log^3 N)$ and the total cost of the kinetic tournament is $O(N \log^2 N)$, where N is the number of line segments to process. We remark that these techniques for finding the upper envelope can only handle the case when $k = 1$ and are not applicable for $k > 1$.

Queries over sliding windows

Processing aggregate queries on data stream [17, 30, 21, 29, 27, 1, 26] has been extensively studied. Li et al. [17] propose an efficient algorithm to compute aggregate queries over sliding windows. We may perform an aggregate query to count the occurrences of the query results over sliding windows in the discrete time domain. However, there are some disadvantages of using the time-stamp model in a discrete time domain. The streaming data in the discrete time domain is usually retrieved by sampling the physical world every every u time units. If u is too small, then the size of data to be processed is large, which will affect the efficiency of the algorithm. If we choose a large u, the accuracy cannot be guaranteed because some value changes are lost in the processing. Therefore, it is either imprecise or inefficient to perform aggregate queries to find the loyal objects by sampling in the discrete time domain.

Alternatively, we may process the data stream of the result changes from the traditional queries. However, this involves the current time as an attribute in the aggregation operator, which makes the aggregation results (loyalties) are changeable from time to

time. Therefore, a data stream processor has to monitor the aggregated values at every moment, which actually incurs enormous overhead. We remark that most algorithms [12, 17, 30, 21] for computing aggregation over data streams do not specifically consider this point, and thus are not able to efficiently support loyalty queries. Additionally, the objects should be further ranked by their loyalties for processing top-k loyalty queries. This is also non-trivial to be implemented in a data stream processor due to the changeable loyalty values. In this paper, our focus is on efficiently processing the continuous updates and detecting loyalty query results in the continuous time domain.

Continuous spatial and temporal queries

The database community has devoted significant research attention to continuously processing spatial and temporal queries [10, 11, 5, 18, 28, 4, 19, 24, 20]. The difference between traditional continuous queries and our queries is that the traditional continuous queries return the query results at each timestamp, while our queries return the objects which appear in query results for a majority of the recent time. Continuous spatial and temporal queries such as the continuous range queries [10, 11, 5] and the k-nearest neighbour queries [18, 28, 4] are well studied. Mokbel et al.[19] present an incremental evaluation paradigm for continuous queries in spatial and temporal databases and its variant [28] can be used to solve continuous k-nearest neighbour queries. We argue that loyalty queries can be used as filters to eliminate the noisy (low loyalty) results from continuous queries. Farrell et al. [9] present a system to process continuous range queries considering spatiotemporal tolerance. However, their scheme is different from ours. We remark that the loyalty queries may help users discover interesting motion patterns based on a large number of existing techniques.

2.2 Problem Definition

Traditional queries. A traditional query Q defines a set of criteria. Given an object o and a timestamp t, we use $Q(o,t)$ to denote whether o satisfies the query criteria of Q at t. For ease of presentation we define $Q(o,t)$ using a step function.

$$Q(o,t) = \begin{cases} 1 & \text{if } o \text{ satisfies the query critera of } Q \text{ at } t; \\ 0 & \text{if } o \text{ does not satisfy the query critera of } Q \text{ at } t. \end{cases}$$

Consider an application for monitoring the cars around the parking space and the parking system notifies the cars with high loyalties in Figure 2.1. Given two moving objects(cars) o_1 and o_2, o_1 enters the space at time 5 and leaves at time 8. o_2 enters at time 10 and leaves at time 18. Therefore, $Q(o_1,6) = 1$ and $Q(o_2,6) = 0$.

Sliding windows. Usually users are not interested in the entire past history of the data stream but rather the recent data over sliding windows. In this paper, we consider a data stream model in the continuous time domain. For a fixed length of time period T, a sliding window contains all the objects and the corresponding attributes within last T time units. We argue that the stream model in the continuous time domain is more general than the model in the discrete time domain. In the rest of the paper we only consider our problem in the continuous time domain. However, our techniques can also be applied to answer the loyalty queries in the discrete time domain.

Loyalty of an object. Given a traditional query Q and a sliding window size T, we define $loyalty(o,t)$ (the loyalty of an object o at time t) as follows.

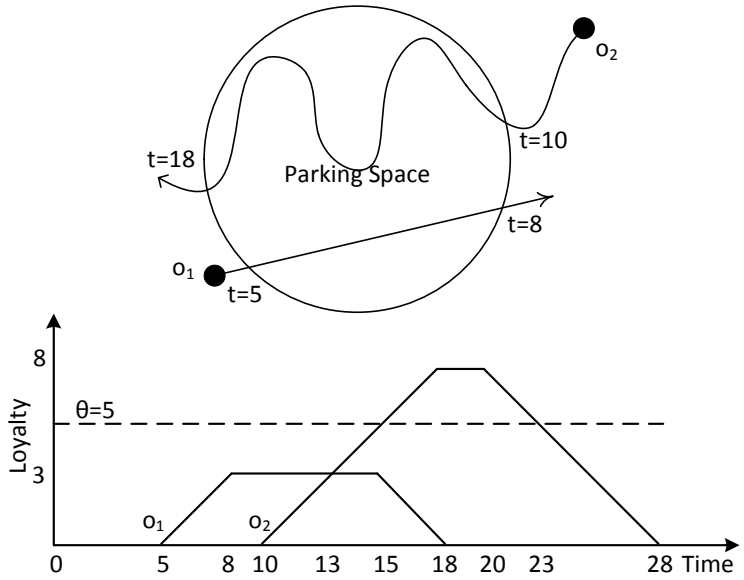


Figure 2.1: Example of Loyalty Queries

$$loyalty(o,t) = \int_{t-T}^{t} Q(o,x)dx$$

The loyalty of an object o shows how long o is a query result of Q during the last T time units. Without loss of generality, in this paper we prefer the objects with higher loyalties.

Consider a sliding window of size 10 ($T = 10$) in Figure 2.1. Then, $loyalty(o_1, 8)$ (the loyalty of o_1 at time 8) is 3 because o has been around the parking space for 3 time units. Note that the coordinates in Figure 2.1 present the loyalties of o_1 and o_2 as the time t changes. Similarly, we can see that $loyalty(o_2, 13) = 3$.

Top-k loyalty queries. Consider a set of objects O, a traditional query Q, a sliding window size T and a parameter k. The top-k loyalty query at time t returns an answer set from O that consists of k objects such that for every object o in the answer set and for any other $o' \in O$, $loyalty(o,t) \geq loyalty(o',t)$.

Consider the example in Figure 2.1. If we monitor the top-1 loyal object and the window size T is 10, o_1 is the result of the top-1 loyalty query from 5 to 13 and o_2 is the result from 13 to 28.

Threshold loyalty queries. Consider a set O of objects, a traditional query Q, a sliding window size T and a threshold θ, the threshold loyalty query at time t returns an answer set from O that consists of any object o such that $loyalty(o,t) \geq \theta$.

In Figure 2.1, given the threshold $\theta = 5$, o_2 is a result of the threshold loyalty query from time 15 to 23.

Continuous queries. In this paper, we study the continuous loyalty queries, namely, we issue the query once and it monitors the query results continuously. Since we solve the queries in the continuous time domain, it is impossible to compute the results for an infinite number of time snapshots. In this paper we shows that although the loyalty of an object is changing over time, we do not need to update the loyalty and the query

results for every time snapshot.

Note that the top-k loyalty queries are more challenging to solve, since we need to consider the relationships among the objects. In this paper we mainly focus on solving the top-k loyalty queries.

3 Framework

In this section we introduce our framework for solving loyalty queries for any given traditional query Q. In real world scenarios, given a set of objects of observation O, the objects may be distributed and users may want to know the global results of a loyalty query. Thus, we present a general framework that aims to handle the loyalty queries in both centralized and distributed environments.

Our framework consists of two main components: the traditional query module and the loyalty query module.

3.1 Traditional query module

Given a traditional query Q (e.g., a range query), each object issues an *update* when it starts satisfying the query criteria or when it stops satisfying the criteria. More specifically, traditional query module is responsible to report to loyalty query module whenever the value of $Q(o,t)$ is changed for any object o.

In a centralized system, the system detects the updates of the objects and processes the updates internally. In distributed environments such as client-server architectures, an object (client) sends a message to the loyalty query module (server) to report an update.

Object updates. Given a query Q and an object o, we say there is an update u of o at time t if the derivative of Q at t is infinity, i.e., $\frac{d}{dt}Q(o,t) = \infty$. In other words, $Q(o,t)$ changes at time t.

Consider the example in Figure 2.1, a moving object issues the update only when it enters or exits the monitoring space. Therefore, o_1 reports two updates at time 5 and 8, and o_2 reports two updates at time 10 and 18.

Basically we adopt existing techniques for continuously monitoring the results of the traditional queries. A straightforward way is to continuously monitor the traditional query result and report once the update occurs. However, since most state-of-art techniques for continuous monitoring queries compute and output their results incrementally, it is seamless to report updates based on these online algorithms. For instance, the techniques in the papers [5, 6] can work as a traditional query module to find the loyal objects within the query range or the influence zone for a majority of the recent time. As this part of work has already been done and our aim is to support a variety of traditional queries in our loyalty query framework, in this paper we focus on efficiently processing of the loyalty queries.

3.2 Loyalty query module

If we assume the attributes of an object is varying continuously such as a moving object, the number of updates during a time period is finite. For a specified loyalty query, it receives updates from the traditional query module in the form of an update stream $U = \{u_1, u_2, u_3, ..., u_n\}$. The updates arrive in the time order. We process the updates continuously in the loyalty query module and output the results to users.

Our query algorithm is triggered only when the update arrives or a possible result change of the loyalty query happens. Therefore, we can output updated results of the loyalty queries when the result changes. In other words, we report which object is newly added in the answer set or which object is removed from the set. Then, the cost of each output is $O(\Delta)$ where Δ is the number of result changes. Figure 3.1 shows the general framework for processing loyalty queries in a distributed system.

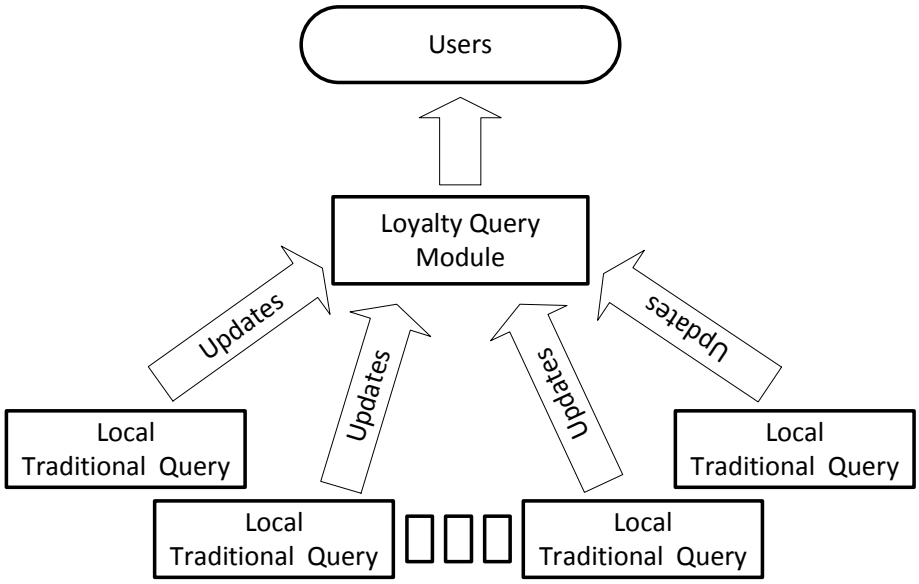


Figure 3.1: Framework of Loyalty Queries

4 Top-k loyalty queries

Before we describe the algorithm of processing threshold loyalty queries, we present the details of answering top-k loyalty queries. This is because it is more challenging to solve the top-k loyalty queries and similar techniques can be applied to the threshold queries.

Initially, we present the base algorithm of solving top-k loyalty queries. We then extensively analyze the time and space complexity of the proposed approach. Finally, we present an efficient pruning technique to further accelerate the base algorithm.

4.1 Algorithms

Consider a top-1 loyalty query. If we draw the loyalty changes in a loyalty-time plane (see Figure 2.1), intuitively this problem is similar to finding the upper envelop in this plane. Similarly the top-k query is to retrieve kth upper envelops. This problem can be solved by the line sweep algorithm in computational geometry. Given N line segments in the plane, the Bentley-Ottmann sweep algorithm [3, 22] maintains the exact vertical ordering of the intersections of the line segments, when the vertical line sweeps the plane from left to right. The total cost is $O((N + M) \log N)$ where M is the number of intersections of the line segments. In the worst case, the number of intersections

M can be $O(N^2)$. A simple example is that the half of lines are horizontal and the other half are increasing. In this case the number of intersection is $N^2/4$. Therefore, the overall complexity can be $O(N^2 \log(N))$. In our problem, we use N to denote the number of updates issued in the last T time units. Then, the amortized cost of the Bentley-Ottmann algorithm is $O(N \log N)$ for each update. In this paper we present an algorithm to answer the top-k loyalty queries in $O(\log N)$ time for each update. The space requirement of our algorithm is $O(N)$.

Before we describe the algorithm, we show some obversion for handling updates in the sliding windows to enable the efficient computation.

States of objects. The state of an object o denotes whether the loyalty of o is increasing, stationary or decreasing. The state of o can be derived by computing the derivative of $loyalty(o, t)$.

$$\begin{aligned} state(o,t) &= \frac{d}{dt} loyalty(o,t) \\ &= \frac{d}{dt} \int_{t-T}^{t} Q(o,x)dx \\ &= Q(o,t) - Q(o,t-T) \end{aligned}$$

As shown above, $state(o, t)$ depends on the traditional query result at the current time $Q(o, t)$ and the result T time before the current time $Q(o, t-T)$. Moreover, there are only three types of states: increasing, stationary and decreasing.

- *Increasing.* The loyalty of o is increasing if $state(o, t) = 1$.
- *Stationary.* The loyalty of o is stationary if $state(o, t) = 0$.
- *Decreasing.* The loyalty of o is decreasing if $state(o, t) = -1$.

In Figure 2.1, the loyalty of object o_1 is increasing from 5 to 8. Then, the loyalty of o_1 is stationary from 8 to 15 and finally becomes decreasing from 15 to 18.

Echo updates. As soon as an *original* update arrives from the traditional query module, we know the traditional query result at the current time $Q(o, t)$ changes. Moreover, these updates will expire from the sliding window after T time, which will affect $Q(o, t - T)$. Therefore, we clone a series of *original* updates and make them take effect after T time. These updates are annotated as *echo* updates. For example, in Figure 2.1 we retrieve an original update at time 5 that o_1 becomes a result of the range query. Given $T = 10$, the echo update is created at time 15. The updates stand for both original and echo updates in the following of the paper, unless mentioned otherwise.

Determining states. When we receive an original update u from a traditional query, we update the current query result $Q(o, t)$. Then we create an echo update u'. The timestamp of u' is $t+T$ and we also attach the new query result. Therefore, $state(o, t)$ can be computed by maintaining $Q(o, t)$ and $Q(o, t - T)$ correctly. In our algorithm we only handle the update if the state of an object changes.

Data structures. In order to efficiently maintain the top-k loyal objects over sliding windows and the sequence of future updates and events, our algorithm maintains the following data structures:

- **Update queue** U (a FIFO data structure) is utilized to maintain a sequence of echo updates. Each update is associated with the timestamp t_4 when it will be issued, the object o and the updated traditional query result $Q(o, t)$. The echo updates are created in the sequence of the original updates. Therefore, we can simply use a FIFO to organize the echo updates.

- **Border object** BO is denoted as the $(k+1)$th loyal object at time t. We set BO empty if the number of objects is less than $k+1$. We define the *border line* indicating the $(k+1)$th line segment which divides the top-k lines and the remaining lines in the loyalty-time plane. In our algorithm only the line intersections related to the border line are processed. Consider a more complicated example of the top-2 loyalty query in Figure 4.1. The object o_3 is the 3rd loyal object from t_1 to t_3. Hence, $BO = o_3$ from t_1 to t_3. We mark the border line with a bold polyline in Figure 4.1.

- **Event queue** E (a priority queue) is utilized to maintain a sequence of potential future events. *Events* denote the potential future result changes of the loyalty queries. The result changes occur only when the border object swap its order of the loyalty with another object. In the loyalty-time plane, the event is created when one line will potentially intersect the border line in the future. If an event is created at time t, each event is associated with the *signatures* of the border object BO and another o at time t. A signature is the identification of the last update of an object. The signature of o will be changed if any update or event related to o is processed. An event is invalid and will not be processed if the signature of BO or o of the event is not up-to-date. The event is inserted into the event queue with the timestamp t' where t' is the potential intersecting time. Consider the example in Figure 4.1. We can predict that the line segment of o_1 will potentially intersect the border line at t_3. Therefore, an event is created to handle the intersection.

- **Top-k sets** $A = A_+ \cup A_= \cup A_-$ maintain the objects geometrically *above* the border object, namely the top-k loyal objects. A is divided into three subsets according to the states of the objects. A_+, $A_=$ and A_- are the subsets of the top-k objects with increasing, stationary and decreasing states respectively. Each subsets is organized in a binary search tree and the elements in the subset are sorted in the decreasing order of their loyalties. Consider the example of Figure 4.1. A contains two objects o_1 and o_2 at t_1. $A_+ = \{o_2\}$ and $A_= = \{o_1\}$.

- **Bottom sets** $B = B_+ \cup B_= \cup B_-$ maintain the remaining objects *below* the border object. B is also divided into B_+, $B_=$ and B_- according to the states. Note that unlike other subsets, B_- can be organized just in a list without sorting their loyalties. B_+ and $B_=$ are represented explicitly in the binary search trees with the decreasing order of loyalties. Consider the example of Figure 4.1. B contains one object o_4 at t_4 and $B_+ = \{o_4\}$.

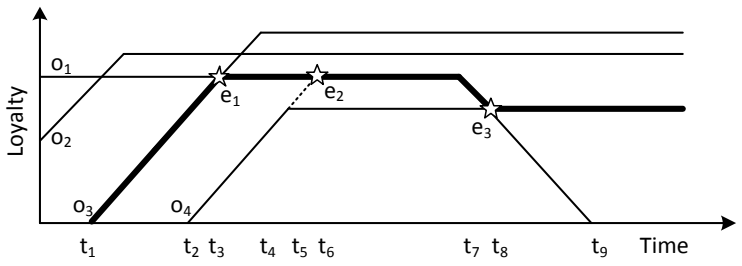


Figure 4.1: Example of Top-2 Loyalty Queries

Solution overview. Before we present the details of our algorithm for processing top-k loyalty queries, we show the main idea of our algorithm. The algorithm uses a sweep line approach to process updates and create events for handling the possible result changes. The algorithm is triggered when 1) an original update arrives from traditional query module, or 2) an echo update arrives from the update queue, or 3) an event arrives from the event queue. We make sure that our algorithm correctly maintains the border object and the objects in top-k set. An event is created if a possible result change of the loyalty query will occur in the future.

Algorithm 1 ProcessUpdate(u)

1: Determine $state(o, t)$ and update $loyalty(o, t)$.
2: **if** $o \in A$ **then** /* o `is in the top-`k `set` */
3: Remove o from the subset A_i
4: Add o into the corresponding subset A_j
5: **else if** $o \in B$ **then** /* o `is in the bottom set` */
6: Remove o from the subset B_i
7: Add o into the corresponding subset B_j
8: **else if** $o \notin BO$ **then** /* o `is a new comer` */
9: **if** $|A| < k$ **then** /* `# of objects less than` k */
10: Add o into A_+
11: **else if** $BO = \emptyset$ **then** /* `# of objects is` k */
12: $BO = o$
13: **else**
14: Add o into B_+
15: CheckSetVariation(BO, A, B) /* `Call Algorithm 2` */
16: Update the signature of o.

Processing updates. When a new update arrives from the update queue, we first recompute the state and loyalty of the corresponding object. Then, the object is moved to the correct subset. As the position of the object in a subset may be changed, we check the subsets and the border object and create the possible events.

Algorithm 1 shows our algorithm for processing a newly arriving update. First we determine the state of the object o based on the query results on both slides of the sliding window (see line 1). If the update is original, we create an echo update by cloning the original update and insert it into the update queue P. Since the state of the object o changes, we move o into the corresponding subset based on its state and current position(lines 2–8). As the orders of elements in the subsets may change, we call Algorithm 2 to check the variation related to the border line and create new events to handle the future intersection(line 15). Finally we update the signature of o (line 16) as the state of o changes.

Handling set variations. Algorithm 2 shows the procedure of handling the variation of the subsets and creating events for the possible result changes. We observe that any intersection related to the border line is associated with the line segment immediately above or below the border line in each subset. Another important observation is that the two line segments with the same state(in the same subset) will not intersect each other. Therefore, we only check the last elements(objects with the minimal loyalty) in $A_=$ and A_-, and the first elements(objects with the maximal loyalty) in B_+ and $B_=$, which are the only potential line segments (objects) to first intersect the border line without considering the new updates in the future (see lines 1–8). In Figure 4.1,

Algorithm 2 CheckSetVariation(BO, A, B)

1: **if** $A_{=}.last$ is varied and $state(BO, t)$ is increasing **then**
2: AddEvent($A_{=}.last$, BO)
3: **else if** $A_{-}.last$ is varied and $state(BO, t)$ is not decreasing **then**
4: AddEvent($A_{-}.last$, BO)
5: **else if** $B_{+}.first$ is varied and $state(BO, t)$ is not increasing **then**
6: AddEvent($B_{+}.first$, BO)
7: **else if** $B_{=}.first$ is varied and $state(BO, t)$ is decreasing **then**
8: AddEvent($B_{=}.first$, BO)
9: **if** BO is varied **then**
10: **if** $state(BO, t)$ is increasing **then**
11: AddEvent($A_{=}.last$, BO)
12: AddEvent($A_{-}.last$, BO)
13: **else if** $state(BO, t)$ is stationary **then**
14: AddEvent($A_{-}.last$, BO)
15: AddEvent($B_{+}.first$, BO)
16: **else if** $state(BO, t)$ is decreasing **then**
17: AddEvent($B_{+}.first$, BO)
18: AddEvent($B_{=}.first$, BO)

$A_{=}$ contains two objects o_2 and o_3 at t_6, and the last element in $A_{=}$ is o_2. Then, we consider the state change of the border object BO. Based on the state of the border line, two events are created to handle the possible intersections(lines 9–18).

Algorithm 3 AddEvent(o, BO)

1: Compute the intersecting time t' of o and BO.
2: Create an event e associated with o, BO and their signatures.
3: Insert e into event queue E with timestamp t'.

Creating events. Algorithm 3 shows how we create a new event. We first compute the intersecting time t' of the two lines segments(line 1). Then, the event e is created and inserted into the event queue E with t'(lines 2 and 3). Note that it is important for us to store the signature information of o and BO with event e. The change of the signature of o indicates that the state or position of the object has been updated before the event occurs. Therefore, the event is invalid and will not be processed.

Algorithm 4 ProcessEvent(e)

1: **if** the signatures of o and BO are not varied **then**
2: Add BO into the subset of o and remove o from the subset.
3: $BO = o$.
4: CheckSetVariation(BO, A, B)
5: Update the signatures of o and BO.

Processing events. The details for processing an event is shown in Algorithm 4. When an event e arrives from the event queue E, we first check the validity of the line intersection by verifying the signatures (line 1). If it is valid, we swap the positions of BO and o (lines 2 and 3), and call Algorithm 2 again since the subsets and BO are changed(see line 4). We update the signatures of the objects as well (see line 5), since

the positions of the objects are changed.

Handling objects with zero or maximum loyalty. Note that in the above algorithms we do not especially handle the objects with zero loyalties or maximum loyalties(the loyalty is T). Here, we show that these objects can be processed more efficiently. For an object o with $loyalty(o,t) = 0$ and $state(o,t) = 0$, we simply remove o from the subsets. For the objects with $loyalty(o,t) = T$ and $state(o,t) = 0$, we maintain a list F to store the objects instead of placing them in $A_{=}$. We can save the cost because maintaining the list is constant in time.

Example 1: Consider the top-2 loyalty query shown in Figure 4.1. Initially, there are two objects o_1 and o_2 with non-zero loyalties. An update of o_3 arrives at t_1. o_3 becomes the border line object(line 11 in Algorithm 1). We mark the border line with a bold line in Figure 4.1. Then, we check the set variation(Algorithm 2). Since BO has been changed, we create an event e_1 with the last element in $A_{=}$ (o_1) for possible order swapping at t_3(line 11 in Algorithm 2). We mark the created event with a star. Note that we only create and process the events (intersections) related to BO. An update of o_4 arrives at t_2. We check subsets variation and no event is created. Event e_1 is processed at t_3. o_3 is moved into A_{+} and o_1 becomes the border object(line 2 and 3 in Algorithm 4). We check the set variation (line 4 in Algorithm 4) and create an event e_2 with o_4 at t_6 (line 15 in Algorithm 2). After that o_4 issues another update at t_5 and the signature of o_4 is changed. Therefore, e_2 is invalid and is not processed at t_6. At t_7, o_4 issues an update and the state of BO is changed. After checking the variation, e_3 is created similarly.

4.2 Analysis

Proof of Correctness

In the proposed algorithm, we make the border object BO present the $(k+1)$th loyal object correctly. All the potential events (intersections) related to BO are created and processed. Therefore, we always make the following inequalities hold.

$$\begin{cases} \min_{o \in A} \{loyalty(o,t)\} \geq loyalty(BO,t) \\ loyalty(BO,t) \geq \max_{o \in B} \{loyalty(o,t)\} \\ |A| \leq k \end{cases}$$

In our algorithm the objects in top-k set cannot be changed unless BO is changed. As a consequence, our algorithm correctly determines the top-k loyal objects.

Performance Analysis

We first analyze the time complexity of our algorithm. As we use binary search trees to maintain the subsets, the cost of inserting or removing an object in a subset of A is $O(\log k)$ and the corresponding cost in a subset of B is $O(\log L)$ where L is the number of objects which have updates in the last T time unit. The cost of insertion in the update queue is $O(1)$ because the update queue is a FIFO. Let M be the number of events processed in the last T time units and M' be the number of events created in the last T time units. Note that some created events may become invalid and will not be processed in the future. As the event queue is organized by a priority queue, the cost of insertion in the event queue is $O(\log M')$, where M' is also the size of the event queue. Let N be the number of updates issued in the last T time units. For each processed update and event, the algorithm creates constant number of events. Therefore, $M' = O(N + M)$.

Then, the total cost in the last T time units is $O((N+M)(\log M' + \log k + \log L) = O((N+M)(\log (N+M) + \log k + \log L)$. Note that $k \leq L$ and L is usually much smaller than the total number of objects n. Therefore, the total cost in the last T time units is $O((N+M)(\log (N+M) + \log L)$.

In Theorem 1 we prove that the number of processed events is at most twice of the number of updates, i.e., $M \leq 2N$. For each processed update (see Algorithm 1 and Algorithm 2), we create at most two events. Note that actually at most one event will be processed among the created two events. This is because after one event is processed, the signature of the object is changed and the other event becomes invalid. However, when we process an event (see Algorithm 4), another two events will be created. Hence, the theorem is non-trivial. We show that the theorem can be proved by the geometry property of the border line.

Theorem 1. *Given N updates, our algorithm processes at most $2N$ events. $M \leq 2N$.*

Proof. Consider the loyalty-time plane and assume that each line segment presents an update in the plane (see Figure 4.1). The border line is actually one of the connected line segments that go through the plane from left to right. For an increasing line or decreasing line, it appears in the border line at most once, while a horizontal line may appear in the border line multiple times. However, the horizontal lines are only connected with the increasing and decreasing lines in the plane. Assume that the border line has at least two line segments. Therefore, one horizontal line on the border line must connect with one increasing line or decreasing line. Let P be the number of line segments on the border line and Q be the number of increasing and decreasing lines. In the worst case, every horizontal line segment is associated with one increasing or decreasing line. Therefore, $P \leq 2Q$. Each connected vertex on the border line presents a processed event. Consequently, we prove that $M \leq 2N$. □

Theorem 1 indicates that the number of processed events is at most twice of the number of updates. We can derive that $M = O(N)$. Moreover, $L \leq N$ because the number of objects which have updates will not larger than the number of updates. Therefore, the total cost of our algorithm in the last T time units is $O(N(\log N))$. The cost for each update is $O(\log N)$.

Proof of Optimality

Theorem 2. *In the worst case, the lower bound cost of updating the results of a top-k loyalty query is $O(\log N)$ for each update where N is the number of updates issued in the last T time units.*

Proof. We show that it is necessary to maintain a priority queue to process the future events. An event actually means a possible result change of the loyalty query. Consider that we have n objects with the stationary state and different loyalties, and we are monitoring a top-1 loyalty query. Let o_i be the ith loyal object. The border object is o_2. Then, the object with lowest loyalty o_n has an update and the loyalty of the object becomes increasing. This creates an event because o_n is possible to become a border object in the future. After that o_{n-1} issues an update and becomes increasing and so forth. Assume loyalty of o_2 is much higher than the objects below. Therefore, we have N updates and may create N events where $N = n - 2$. Firstly, we argue that we must store all the these possible events to correctly report a future result change, otherwise we may miss a possible result change. This is because any object is possible to become

a border line object if all the objects above it issues an update and become stationary state. Secondly, we must keep the event in order so that we can efficiently know the first event in future. In other words, we employ the priority queue to maintain all the possible events. The minimum cost of maintaining an event in such data structure is $O(\log N)$. Therefore, in the worst case it takes $O(\log N)$ time to process an update. □

In the worst case, our algorithm meets the lower bound cost of the problem, thus is optimal in the worst case.

Space Analysis

Next, we investigate the space requirement of our algorithm. The space of the update queue is $O(N)$ where N is the number of updates issued in the last T time units. The size of the event queue is $O(M')$. According to the above analysis, $M' = O(N)$. The size of each subset is $O(L)$. If we do not consider the objects with maximum loyalties, then $L \leq N$. Therefore, for each top-k loyalty query, our algorithm uses $O(N)$ space.

4.3 Pruning

Although the algorithm is already optimal for solving the top-k loyalty queries in terms of time complexity, in this subsection we show that we can further prune some of the updates from the computation of the final results. The pruning rule can reduce both the overall computation cost and the communication cost in terms of the number of messages exchanged over distributed data streams. We first present an observation that can reduce the number of considered updates, and show how the pruning rule works over centralized data streams.

Theorem 3. *Let o_k be the object with the minimal loyalty in A and o be any object in O. o will not be a result of top-k loyalty query in the next $(loyalty(o_k, t) - loyalty(o, t)/2$ time, where t is the current timestamp.*

Proof. Consider that o_k becomes decreasing and o becomes increasing at t. Let $d = (loyalty(o_k, t) - loyalty(o, t))/2$. o will be always below o_k in the time period $[t, t + d)$. Thus, $loyalty(o_k, t + \Delta t) > loyalty(o, t + \Delta t)$ where $0 \leq \Delta t < d$. Consequently, we prove Theorem 3. □

Based on the theorem, we may ignore some computation of $o \in O$ in time period $[t, t + d)$. We call d is the *safe time* of object o at t. To achieve this we maintain a list of echo updates for each object. In our algorithm we avoid the redundant computation for the trivial updates in the safe time. Next, we define the trivial updates.

Trivial updates. Let $U_o = u_1, u_2, ..., u_n$ be a series of echo updates of object o in the update queue U at time t. The trivial updates are a subset $U_t \subseteq U_o$ such that for each u_i in U_o, $u_i.time < t + d$.

Since the object will never be a top-k loyal object during the safe time, the trivial updates in this period will not affect the top-k results. Thus, it is not necessary to wait and process the trivial updates one by one. Instead, for all the trivial updates in U_o we only update the data structure once.

Algorithm 5 shows how we process the trivial updates. We process the updates from the list U_o. If the first update is non-trivial (line 3), we just call Algorithm 1 and process the update normally (line 7). If the update from the list is trivial, the algorithm

Algorithm 5 ProcessUpdateWithPruning(U_o)

1: Let u_i be the ith update in U_o.
2: $i = 1$
3: **while** u_i exists and u_i is trivial **do**
4: Recompute the loyalty of o based on u_i.
5: $i = i + 1$
6: **if** $i = 1$ **then**
7: ProcessUpdate(u_1). /* `Call Algorithm 1 without pruning` */
8: **else**
9: ProcessUpdate(u_{i-1}). /* `Call Algorithm 1 with pruning` */

continue to find the next update from U_o and update the loyalty of the objects according to the update u_i until the next update is non-trivial or there is no update left in the list.(see lines 3–5). Then, we process the echo update u_{i-1} with the modified loyalty of o (line 9).

The algorithm is triggered only when an echo update is processed. The cost of finding the trivial updates and updating the loyalty takes $O(|U_t|)$ time and processing of the update using Algorithm 1 takes $O(\log L)$. Therefore, $(|U_t| - 1)$ trivial updates scheduled to be processed in the future are processed in $O(1)$ time for each. Thus, our pruning technique reduces the total cost of the computation.

Optimizing communication cost. In the context of many applications within distributed networked systems such as sensor networks, the communication overhead is also an important issue. Since the communication is the principal energy drain for a sensor node, reduction on the number of communication times can maximize the running time of a sensor node. A lot of research has gone into design of algorithms that are optimal with respect to the number of messages exchanged[15, 7, 14, 8]. Here we consider the network messages are based on a two-way communication protocol which is commonly utilized in distributed data stream processing[7, 14], and we show that the communication cost can be reduced by pushing the pruning rule into the local nodes.

For each local node, we dynamically maintain the loyalties of a subset of objects O_i based on the updates and current objects' states. When a local node sending an update to the loyalty query module, the loyalty query module immediately returns the current loyalty of the kth object $loyalty(o_k, t)$. Thus, whenever the node detects a new update, we can determine the update is trivial or not based on Theorem 3. If the update is trivial, we do not send a message to report the update and just update the loyalty of the object locally. Note that the pruning rule can still be applied to prune the trivial updates for the echo updates on the server side. We evaluate the pruning rule in the experiments and show that the it can reduce about 45% messages exchanged in the network with a large sliding window.

5 Threshold Loyalty Queries

Different from the top-k queries, the threshold loyalty queries report the object whose loyalty is above a threshold θ. This problem is simpler because we do not need to consider the ordering the objects and each object can be considered individually. In the algorithm of threshold queries, we consider the border object BO as a dummy object with constant loyalty which is the threshold. We also maintain two sets: the top-k set A and the bottom set B, but not divide them by the different states. We also

retain the event queue and update queue in the algorithm for threshold queries. An event-based algorithm is proposed in the similar way to the algorithm of top-k queries. Here, we show the differences. 1)A and B do not need to be sorted. 2) When we process an update, the event is created if the object will potentially cross the border line. Therefore, for each update we create at most one event. 3) We do not check the set variation because each object is considered individually. 4) When we process an event of an object o, o is either moved from B to A or moved from A to B. In other words, o either becomes a query result or is removed from the result set. We do not present the details of the algorithm due to the space limitations.

Analysis. For the threshold queries, we do not make the set A and B sorted. Therefore, each insertion and deletion in A and B takes constant time. Let N be the number of updates issued in the last T time units, M be the size of event queue. The cost of maintaining the event queue is $O(\log(M))$. For each update we create at most one event and for each object we maintain at most one event, namely $M \leq N$. Therefore, the cost of the algorithm is $O(\log N)$ for each update. As we handle the events for multiple objects, the part $O(\log N)$ is necessary for our algorithm to maintain the priority queue. Similarly, our algorithm uses $O(N)$ space.

Pruning. The similar pruning technique proposed for top-k queries can be used for answering the threshold loyalty queries. The definition of the trivial updates is slightly different. Since we know the border line is horizontal, an increasing object in B will not cross the border line in the next $\theta - loyalty(o,t))$ time. Let safe time $d = \theta - loyalty(o,t))$. Therefore, any update in the time period $[t, t+d]$ is considered as a trivial update. Also, the technique for reducing the communication cost is still applicable for the threshold queries. We omit the details here.

6 Experiments

All algorithms are implemented in C++ and complied by GNU GCC. The experiments are performed on a PC with Intel Core i5 3.10GHz CPU and 8G memory under Debian Linux. We conducted extensive experiments on both real and synthetic data sets. Due to the space limitations, we present only the most representative results.

In the experiments, we focus on evaluating the performance of the proposed algorithm for answering top-k loyalty queries. Therefore, we do not count the cost of computing traditional query results and assume that all the inputs are in the form of object updates.

Real data. We use the global surface summary data (GSOD)[1] produced by the National Climatic Data Center (NCDC). We collect the climatic data from GSOD between 1930 to 1980. The record in the data set includes timestamp, station id, a variety of sensor data, and indictors for occurrence of fog, rain, snow, hail, thunder and tornado. We preprocess the data set to output the updates of the occurrences of rain. Therefore, we can find the rainiest stations over sliding windows by using a top-k loyalty quires. The data set consists of 7.6 million records collected from 12237 stations.

Synthetic data. In our experiment we simulate continuous time domain in discrete timestamps. Synthetic data is generated by a two state Markov chain model, which has many applications as statistical models of real-world processes. For each object o_i,

[1] ftp://ftp.ncdc.noaa.gov/pub/data/gsod/

$$Pr(Q(o_i, t+1) = 1 | Q(o_i, t) = 0) = p_i$$
$$Pr(Q(o_i, t+1) = 0 | Q(o_i, t) = 0) = 1 - p_i$$
$$Pr(Q(o_i, t+1) = 0 | Q(o_i, t) = 1) = p_i'$$
$$Pr(Q(o_i, t+1) = 1 | Q(o_i, t) = 1) = 1 - p_i'$$

p_i and p_i' are uniformly chosen from $[0, m]$ for each object. The data set consists of 10 million random updates with n objects.

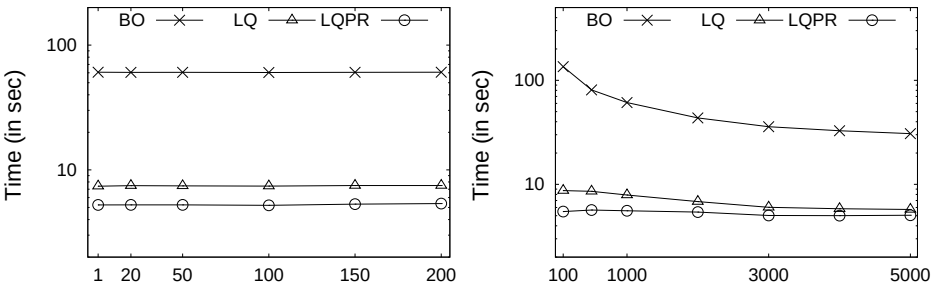
Table 6.1: Parameters used in the experiments. The default values are shown in bold

Parameter	**Range**
Sliding window size T ($\times$ 1000)	**10**, 25, 50, 75, 100
# of objects n ($\times$ 1000)	**1**, 5, 10, 15, 20
# of results k	1, 10, 20, 50, **100**, 150, 200
Probability parameter m	0.0001, **0.001**, 0.01, 0.1, 1

The table 6.1 shows the different parameters used in our experiments and the bold values are the default values used in the experiments unless mentioned otherwise.

To the best of our knowledge, we are the first to study the problem of top-k loyalty queries. We use the Bentley-Ottmann algorithm as our competitor called *BO* below. Our base loyalty query processing algorithm is called *LQ*. The loyalty query processing algorithm optimized by using the pruning rule is called *LQPR*. Note that all the figures are in the logarithmic scale except the figures for evaluating our pruning technique.

In Figure 6.1, we compare our algorithm with the Bentley-Ottmann algorithm using the real climatic data set. We process the whole data set and evaluate the running time of the algorithms. Our algorithm is extremely efficient (processing 7 million updates in seconds) and demonstrates one order magnitude improvement over the Bentley-Ottman algorithm. The algorithm with pruning rule outperforms the base algorithm in all the settings. In Figure 6.1(a) and Figure 6.1(b), we study the effect of k and T on the algorithms. The default window size is 1000. As expected, the cost of these algorithm is not significantly effected by the variation of k and T. In Figure 6.2(b) we vary the sliding window size T from 100 to 5000. An interesting observation is that the performance of the algorithms is even better when the window size T is large. This is because the range of loyalties is large when the sliding window size is large. This makes the objects less possible to swap their orders. We observe this significantly on BO since it need to process every order change among the objects.

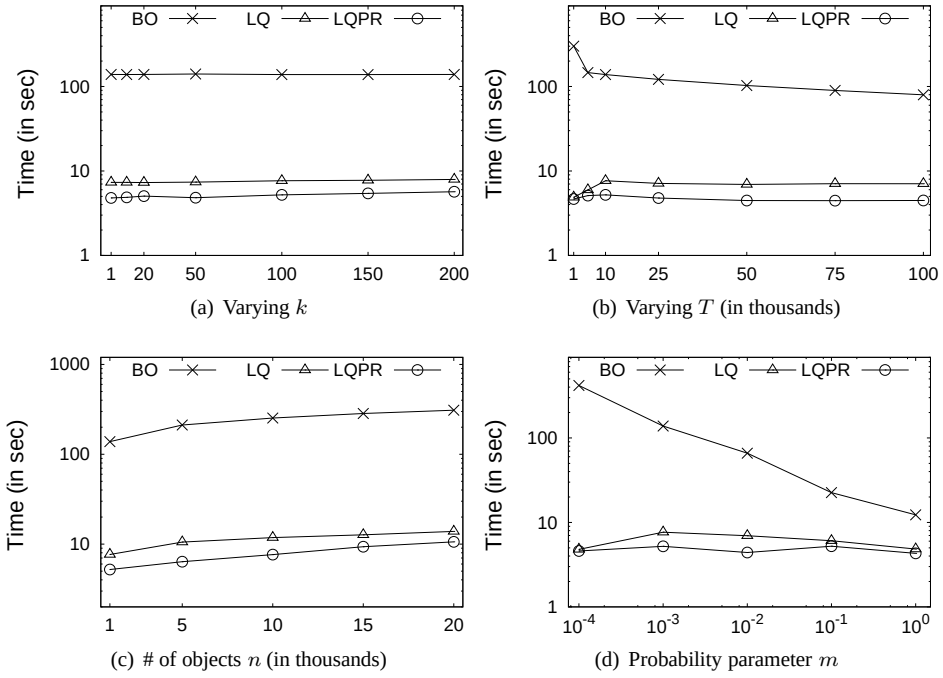


(a) Varying k (b) Varying T(in thousands)

Figure 6.1: Performance evaluation on the climatic data

In Figure 6.2, we perform experiments on syntectic data sets to conduct a more detailed evaluation. We study the effect of varying k and T in Figure 6.2(a) and Figure 6.2(b). The similar tendency can be observed on the synthetic data set. Figure 6.2(a) shows that the pruning rule does not work well when the sliding window size T is small. The reason is that the number of updates generated with certain probability in a small sliding window is small. Therefore, not many updates can be pruned according to the pruning rule.

In Figure 6.2(c) and Figure 6.2(d), we vary the number of objects n and the probability m used in generated synthetic data and study the effect on the algorithms. Figure 6.2(c) shows that the processing time of our algorithms increases with increase in n. This is because the number of objects which have updates in the sliding window L increases with larger n. Figure 6.2(d) shows that the performance of our algorithms remains unaffected with increase in the frequency of updates, although we vary m in a very large scale. LQPR does not show a good pruning power when $m = 0.0001$ because the number of updates in the sliding window is too small so that few updates can be pruned.



(a) Varying k (b) Varying T (in thousands)

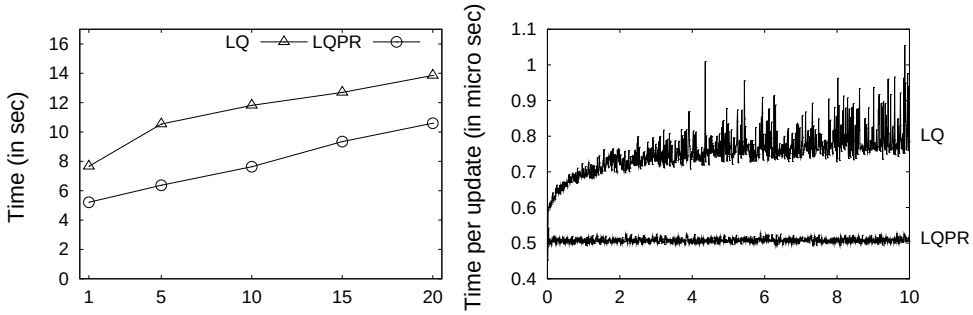
(c) # of objects n (in thousands) (d) Probability parameter m

Figure 6.2: Performance evaluation on the synthetic data

Next, we evaluate the efficiency and effectiveness of the pruning rule on the synthetic data set. We find that BO is about one order of magnitude slower than our algorithms. Thus, we exclude BO in the following evaluation and show the processing time in linear scale. We evaluate the total running time of both our algorithms for a centralized computation in Figure 6.3. Then, we assume that the updates of each object are reported on an independent local client. We simulate a distributed data stream environment and conduct the experiments on evaluating the communication cost in terms of the total number of messages exchanged in the network in Figure 6.4.

Figure 6.3(a) evaluates the total processing varying the number of objects n. We

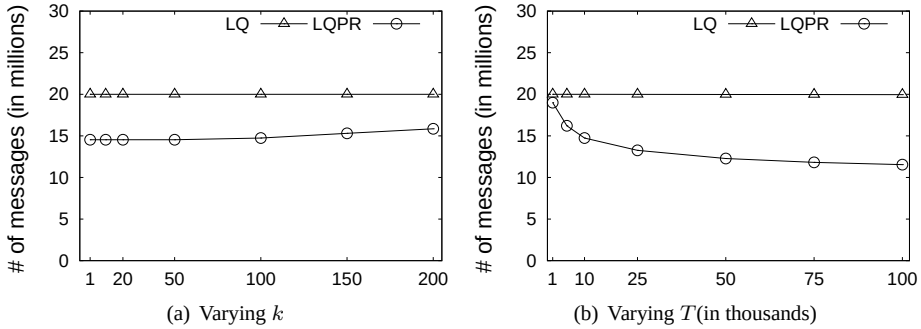
find that the processing time of both algorithms increases with the increase of the number of objects. This is because the number of updates N over the sliding windows increases when n increases for the synthetic data sets. Due to the effectiveness of our pruning technique, LQPR outperforms LQ in all the cases. Figure 6.3(b) studies the average processing per update. Since the processing time of one update is too short to capture precisely, we record the average time for each batch of 10000 updates to estimate the delay per update. It shows that both of our algorithms are very efficient. LQPR can process more than 1.8 million updates per second even in the worst case on the synthetic data set. Moreover, the processing time per update of LQPR varies in a very small range, therefore has better stability than LQ. The algorithms performs slightly better at the beginning of the data sets, because we start our algorithm from scratch.



(a) # of objects (in thousands) (b) # of updates (in millions)

Figure 6.3: Efficiency evaluation for the pruning rule

In Figure 6.4(a) and Figure 6.4(b), we vary k and the window size T and evaluate the number of messages exchanged. Figure 6.4(a) presents that we can reduce the communication cost by about 25% under the default setting. We observe that the pruning power is sightly better for a small k value, because of the higher loyalty of kth object for the small k. Figure 6.4(b) illustrates that the pruning rule works well for a larger window size. The number of messages decreases as T increases. This is due to the loyalties have large scales on large sliding windows and thus leads to a longer safe time to silence a local client. We obverse that about 45% updates be can be ignored with a sliding window of size 100 thousand.



(a) Varying k (b) Varying T(in thousands)

Figure 6.4: Evaluating communication cost

7 Conclusion

We introduce the loyalty queries for a variety of applications. We present efficient algorithms to answer the top-k and threshold loyalty queries. We prove the lower bound cost of the problem and present a detailed complexity analysis to show that our algorithm is optimal. We verify this by an experimental evaluation and demonstrate the efficiency of our approach.

Bibliography

[1] Brian Babcock, Mayur Datar, Rajeev Motwani, and Rajeev Motwani. Load shedding for aggregation queries over data streams. In *ICDE*, pages 350–361, 2004.

[2] Julien Basch, Leonidas J. Guibas, and John Hershberger. Data structures for mobile data. *J. Algorithms*, 31(1):1–28, 1999.

[3] Jon Louis Bentley and Thomas Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Computers*, 28(9):643–647, 1979.

[4] Christian Böhm, Beng Chin Ooi, Claudia Plant, and Ying Yan. Efficiently processing continuous k-nn queries on data streams. In *ICDE*, pages 156–165, 2007.

[5] Muhammad Aamir Cheema, Ljiljana Brankovic, Xuemin Lin, Wenjie Zhang, and Wei Wang. Multi-guarded safe zone: An effective technique to monitor moving circular range queries. In *ICDE*, pages 189–200, 2010.

[6] Muhammad Aamir Cheema, Xuemin Lin, Wenjie Zhang, and Ying Zhang. Influence zone: Efficiently processing reverse k nearest neighbors queries. In *ICDE*, pages 577–588, 2011.

[7] Graham Cormode, S. Muthukrishnan, Ke Yi, Qin Zhang, and Qin Zhang. Continuous sampling from distributed streams. page 10, 2012.

[8] Antonios Deligiannakis, Yannis Kotidis, and Nick Roussopoulos. Processing approximate aggregate queries in wireless sensor networks. *Inf. Syst.*, 31(8):770–792, 2006.

[9] Tobias Farrell, Kurt Rothermel, and Reynold Cheng. Processing continuous range queries with spatiotemporal tolerance. *IEEE Trans. Mob. Comput.*, 10(3):320–334, 2011.

[10] Bugra Gedik and Ling Liu. Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *EDBT*, pages 67–87, 2004.

[11] Bugra Gedik, Kun-Lung Wu, Philip S. Yu, and Ling Liu. Processing moving queries over moving objects using motion-adaptive indexes. *IEEE Trans. Knowl. Data Eng.*, 18(5):651–668, 2006.

[12] Lukasz Golab, Kumar Gaurav Bijay, and M. Tamer Özsu. Multi-query optimization of sliding window aggregates by schedule synchronization. In *CIKM*, pages 844–845, 2006.

[13] John Hershberger. Finding the upper envelope of n line segments in o(n log n) time. *Inf. Process. Lett.*, 33(4):169–174, 1989.

[14] Ram Keralapura, Graham Cormode, Jeyashankher Ramamirtham, and Jeyashankher Ramamirtham. Communication-efficient distributed monitoring of thresholded counts. In *SIGMOD Conference*, pages 289–300, 2006.

[15] Maleq Khan, Gopal Pandurangan, V. S. Anil Kumar, and V. S. Anil Kumar. Distributed algorithms for constructing approximate minimum spanning trees in wireless sensor networks. pages 124–139, 2009.

[16] Feifei Li, Ke Yi, and Wangchao Le. Top-*k* queries on temporal data. *VLDB J.*, 19(5):715–733, 2010.

[17] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD Conference*, pages 311–322, 2005.

[18] Yifan Li, Jiong Yang, and Jiawei Han. Continuous k-nearest neighbor search for moving objects. In *SSDBM*, pages 123–126, 2004.

[19] Mohamed F. Mokbel, Xiaopeng Xiong, and Walid G. Aref. Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD Conference*, pages 623–634, 2004.

[20] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD Conference*, pages 635–646, 2006.

[21] Kanthi Nagaraj, K. V. M. Naidu, Rajeev Rastogi, and Scott Satkin. Efficient aggregate computation over data streams. In *ICDE*, pages 1382–1384, 2008.

[22] F. P. Preparata and M. I. Shamos. *Computational Geometry: an Introduction*. Springer, Berlin, 1985.

[23] Daniel Russel, Menelaos I. Karavelas, Leonidas J. Guibas, and Leonidas J. Guibas. A package for exact kinetic data structures and sweepline algorithms. pages 111–127, 2007.

[24] Zhitao Shen, Muhammad Aamir Cheema, Xuemin Lin, Wenjie Zhang, and Haixun Wang. Efficiently monitoring top-k pairs over sliding windows. In *ICDE*, 2012.

[25] Yufei Tao, Dimitris Papadias, and Dimitris Papadias. Historical spatio-temporal aggregation. pages 61–102, 2005.

[26] Nesime Tatbul, Stanley B. Zdonik, and Stanley B. Zdonik. Window-aware load shedding for aggregation queries over data streams. In *VLDB*, pages 799–810, 2006.

[27] Song Wang, Elke A. Rundensteiner, Samrat Ganguly, Sudeept Bhatnagar, and Sudeept Bhatnagar. State-slice: New paradigm of multi-query optimization of window-based stream queries. In *VLDB*, pages 619–630, 2006.

[28] Xiaopeng Xiong, Mohamed F. Mokbel, and Walid G. Aref. Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *ICDE*, pages 643–654, 2005.

[29] Xiaoyan Yang, Hock-Beng Lim, M. Tamer zsu, Kian-Lee Tan, and Kian-Lee Tan. In-network execution of monitoring queries in sensor networks. In *SIGMOD Conference*, pages 521–532, 2007.

[30] Rui Zhang, Nick Koudas, Beng Chin Ooi, and Divesh Srivastava. Multiple aggregations over data streams. In *SIGMOD Conference*, pages 299–310, 2005.