2013-12-23

Scalable Multi-Parameter Outlier Detection Technology

Jiayuan Wang
Worcester Polytechnic Institute

Scalable Multi-Parameter Outlier Detection Technology

by

Jiayuan Wang

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

Jan 2014

APPROVED:

Professor Elke A. Rundensteiner, Thesis Advisor

Professor Mohamed Eltabakh, Thesis Reader

Professor Craig Wills, Department Head

Abstract

The real-time detection of anomalous phenomena on streaming data has become increasingly important for applications ranging from fraud detection, financial analysis to traffic management. In these streaming applications, often a large number of similar continuous outlier detection queries are executed concurrently. In the light of the high algorithmic complexity of detecting and maintaining outlier patterns for different parameter settings independently, we propose a shared execution methodology called SOP that handles a large batch of requests with diverse pattern configurations.

First, our systematic analysis reveals opportunities for maximum resource sharing by leveraging commonalities among outlier detection queries. For that, we introduce a sharing strategy that integrates all computation results into one compact data structure. It leverages temporal relationships among stream data points to prioritize the probing process. Second, this work is the first to consider predicate constraints in the outlier detection context. By distinguishing between target and scope constraints, customized fragment sharing and block selection strategies can be effectively applied to maximize the efficiency of system resource utilization. Our experimental studies utilizing real stream data demonstrate that our approach performs 3 orders of magnitude faster than the start-of-the-art and scales to 1000s of queries.

Acknowledgements

I would like to express my gratitude to my advisor professor Elke Rundensteiner. Thank for her continuous support for my research and thesis work. Thank for her time on revising my thesis again and again, to make it perfect. I really appreciate her patient guide, encouragement, as well as immense knowledge, which help me to continuously grow and improve during my master study.

My thanks also go to my thesis reader professor Mohamed Eltabakh, for his valuable advises on my thesis work, which helps me to improve the quality of this thesis.

I also thank the mentor in WPI DSRG: Lei Cao, for all the inspirations and feedbacks on my research work.

I thank all my DSRG member in WPI: Chuan Lei, Xizhao Chen, Di Yang, Medhabi Ray, Xika Lin, Qingyang Wang for all the feedbacks and discussion on my work;

Lastly, I want to thank my whole family: my parents Zhihe Wang and Xiaoying Chen. Thank for giving me tremendous support for no matter my life or study.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Nowadays, almost all information is generated, transmitted, and stored as digital data, giving rise to a prevalent focus on how to extract insight from those huge volumes of data. Outlier detection [8] is one popular technique to identify anomalous patterns and then interpret them as the phenomenon in the real world. We can catch a glimpse of its increasing importance in many modern applications in a variety of fields ranging from fraud detection, traffic management, damage evaluation to economical analysis.

The idea of detecting outliers originates from the notion of capturing abnormal phenomena in data put forward in [21]. An abnormal phenomenon is introduced as the core principle that outliers can be identified by using the similarity among points in a dataset. Based on this foundation, one important class of distance-based outlier definitions stands out [5,8,22,23]. The one we used in this paper is given initially in [5] where outlier is an object O with fewer than k neighbors in the database, where a neighbor is an object that is within a distance R from object O. This definition fits well for applications where the threshold for outlier is clear.

In many applications a popular data stream is monitored by many analysts [12,13,14,15,16]. For example, some financial analysts may continuously monitor the stock transaction streams from the New York Stock Exchange to evaluate their stability. They might detect a widely-fluctuating real estate stock by bounding the range of the purchasing price difference from 100 to 200USD and set the configurations for number of neighbors needed at 30 for similar transactions. Meanwhile, other analysts might prefer to use a more relaxed demarcation line to delineate unstable performance, setting the range from 80 to 220USD or the number of similar transactions to 20. Also, some may be interested in the stability across one year, while others may submit queries with a much shorter time span such as six months. Thus applications may receive a huge workload of similar concurrent queries with different pattern-specific and window-specific parameter settings over the same data stream. Accordingly some strategies should be applied to process this workload of similar outlier detection queries to serve applications in real time, by reducing the processing time and increasing the efficiency of delivery of results.

Most optimization principles on sharing system resources are concentrated on pattern-specific and window-specific parameters [7]. To date, predicates have not yet been considered in the outlier detection context. Yet predicates are crucial in expressing the semantics of outliers. For instance, the stocks that some analysts are interested in are confined to Boston only, while others might pay attention to stocks to watch the economy in Massachusetts. Moreover, sometimes whether a data point is an outlier does not entirely rely on the condition of the stream it belongs to. Rather it is common that several data streams need to be considered in context. Again, considering the stock example, the stability of real estate stocks in Boston may be relevant to the stability of building material stocks in Massachusetts, or even the larger scope of material stocks under the USA. Therefore, assuming real estate stocks in Boston are the original streams then building material stocks are streams we use to compare against. Predicates play a role to control and filter

these streams. Clearly, a system that supports hundreds of outlier detection queries with predicates is extremely resource intensive.

1.2 State-of-art Limitations

Handling a huge workload of different outlier detection queries on a single system continuously under high input rates is a challenging problem. Accordingly, although the state-of-the-art method [1] that efficiently deals with a single distance-based outlier detection query has been proved to be optimal theoretically, executing each of the queries independently via using this method still has prohibitively high demands on both computational and memory resources. For example, it takes LEOC [1] 10s to update the processing query results with 1M data points in the window, then it would take us 1000s to process 100 queries by executing LEOC 100 times. This does not meet the real time responsiveness requirement and thus would be prohibitively costly. Thus the method on a single query is not feasible and applicable for practical applications, especially when the number of queries to be executed is large. Therefore, we now propose to leverage the insights and technique of LEOC while designing a resource-shared query processing approach to process a huge of workload of queries.

With regards to lots of extensively researched solutions on the shared workload of outlier detection queries processing [2,4], they reduce the massive system resource utilization caused by the full scan of the whole window for each points being processed. One of the cutting-edge approaches proposed in [2] scales to handle a large quantity of queries where only pattern-specific parameters, namely number of neighbors K and neighbor search range R, are supplied. More specifically, they do not take the window-specific parameters into consideration. Also, they do not support predicates. In addition, the main idea of its outlier detection is based on the single query strategy [2], which is based on

the expensive range query during the search process.

Predicates are known to play an important role in screening the data we are interested in by signifying selection conditions in a query. However, none of the existing outlier techniques [1,2,6] integrates predicates as parameters into the outlier detection process. One insightful sharing strategy [3] of predicates in streaming data environments comes up with an idea of dividing the data set into fragments with different signatures attached to recognize lists of queries that they belong to. We leverage their method to maximize the resource sharing of arbitrary selection predicates. However, prior work concentrates on aggregation only, it is not possible to directly be applied to outlier detection query processing. Especially in our case where the evaluation of the outlier qualification of data points in one stream sometimes depends on another data stream. This implies that outlier detection can involve several data streams instead of a particular one in tradition. The one to be detected and the one to be probed. Based on this idea, predicates can be put on both.

1.3 Challenges & Proposed Solution

This paper is to design, implement and evaluate an optimized technique for processing multiple outlier detection queries in streaming environments. The efficient algorithm we propose, called SOP, handles a huge workload of queries varying three parameter sets where each two variations are contained. They include the number of neighbors K, search range R, window size W, sliding size S, predicate TARGET and predicate SCOPE respectively. In our framework, we introduce several innovative strategies and search operations for optimizing the shared processing of multiple queries to ease the burden of available CPU and memory resources.

First, we design the status indicator technique that takes advantage of the relationship between the parameter setting values in different but similar outlier detection queries. For

queries with less restricted specifications on parameter settings, their outputs are completely contained in the results of queries with more restricted parameter settings. Based on this observation, namely pattern containment, using the status indicator allows us to handle multiple queries without maintaining separate details about a workload of queries. Then we propose a technique so that we simply keep least evidence sufficient for the most restricted queries instead of routinely conducting expensive range queries to gain all qualified neighbors.

Second, we also present a technique to handle predicate parameters in our distance-based outlier detection process. In light of the demands from the real world request, outliers to be detected sometimes are not only being confirmed as abnormal from data streaming they belong to. Instead, their identification of the outlier status depends on some other related data streams. Therefore data streams can be categorized into target and scope based on their purpose in outlier detection queries. Target is the stream where data reside to be evaluated if they are outliers or not and scope is the stream that all target data probe into to find neighbors. By dividing target data into fragments and scope data into blocks based on different predicates specifications on target and scope, we do not have to use brute force to apply our outlier detection technology on data stream over and over. We aim to utilize the uniqueness of fragments and blocks to actualize sharing purpose when executing outlier detection technology. This is based on the characteristic that all data contained in one fragment or block will not be included in other fragments or blocks at the beginning when they are established. Meanwhile we exploit the bitmap as a signature to signify different fragments and blocks we are detecting and probing as well as the lists of associated queries, keeping track of the outputs for each query. Thus duplicate computations can be avoided.

Lastly, our experimental studies on both synthetic and real data demonstrate that SOP successfully reduces the CPU and memory utilization significantly by almost three or-

ders of magnitudes, confirming the effectiveness and superiority over the state-of-the-art alternatives.

1.4 Contributions

In our SOP approach, we successfully tackle all problem outlined above. Contributions of our work on solving this real time outlier detections of multiple queries are summarized as follows:

1) We introduce the concept of a status indicator to efficiently share same patterns for different outlier detection queries. This frees the repeatedly executions on the same data stream which is common for the state-of-the-art methodology [1].

2) We present the least search technique that plays a role in controlling the timing of neighbor search termination. The appropriate termination can minimize the number of comparisons before sufficient evidence for a data point is collected without neglecting to compare some other data points when delivering outliers for all queries in the workload.

3) We integrate these techniques into one framework to enable general parameter settings on outlier detection in streaming environments.

4) We propose an innovative way to incorporate predicate parameters into outlier detection specification settings by fragment sharing strategy and block selection operation on Target and Scope separately to share overlapped portions in predicates. No other approach of sharing of outlier detection is known to support this rich set of parameters.

5) We validate the improved performance of our approach with experiments against other edge-cutting methods on both synthetic and real data.

The rest of the paper is organized as follows. Chpater 2 briefly introduces the preliminary knowledge about distance-based outlier detection and the problem formalized in 3. The technique of SOP on sharing strategies given multi-query with arbitrary parameter

settings is given in Chapter 4 and Chapter 5. Experimental results are analyzed in Chapter 6. Chapter 7 covers related work, while Chapter 8 concludes the whole paper.

Chapter 2

Distanced-Based Outlier Detection Basics

2.1 Basic Concepts

Outliers generally is described as objects that behave differently from the ”typical case”. In recently years, several outlier definitions have been developed to separate outliers from the normal majority. One of the most widely used definitions is based on distance [2,3]: if there are less than k objects within a distance of range r for an object A, then A is considered as an outlier. We use the following definition of distanced-based outlier to define outliers. The function dist (p_i, p_j) is used to denote the distance between data points p_i and p_j. Given the distance threshold R, function nn (p_i, R) represents the number of neighbors a data object p_i has within range R.

Definition 1: *Given R and parameter k (k¿0), if nn (p_i, R) ¡ k, then p_i is regarded as an outlier.*

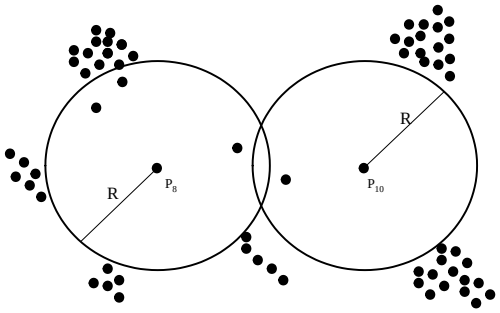


Figure 2.1: An example data set with two distance-based outliers

2.2 Outliers in Sliding Windows

One distinguishing trait of streaming data compared to static data is its infinity. Another feature is the high velocity when streaming data arrive the system. This high speed and volume arouse the difficulties in maintaining and processing all these on-the-fly data at real time. In order to tackle streaming data, a sliding window semantics that is widely used in literature [6,7] is taken out so that we can chopped infinite streaming data into continuous finite snapshots and then apply our outlier detection algorithm in each snapshot. With this window mechanism, we are able to overcome the difficulties caused by the huge volume and high arrival speed of data stream.

Meanwhile, it is well known that most analysts are more interested in the fresh data. This is because fresh data always contains more useful information hidden behind. Accordingly, among outlier detection analysis, it is the most recent data that are the main focus instead of the ancient one. Window mechanism is propitious to the processing of newly-arrived data and the expiration of old ones.

However, when applying window mechanism in the distance-based outlier detection, arrival and expiration of data points inevitably affect the total number of neighbors of each data point in the latest window. This is because neighbors change over sliding win-

dow. Hence, capability of dealing with this instability obviously becomes one of the most system-resource-consuming considerations of the query processing tasks.

Here we give an example of how naive distance-based outlier detection in sliding window works. Assume that there is a distance-based outlier detection query with R specified as 5 and K specified as 3. Data point p_i have p_1, p_3, p_7 and p_9 as neighbors in the current window W_1. After 5 seconds, window slides. p_1 and p_3 expire and no longer can be counted as the neighbors of p_i. Meanwhile, there are no more neighbors found in the new-arrived data. At this point p_i only collects two neighbors. Accordingly, we claim that p_i becomes an outlier after window slides.

In streaming database systems, we assume all arriving data points have their own unique timestamps, denoted as p_i.ts. If p_i.ts is greater than p_j.ts, it means that p_i arrives earlier than p_j. For all neighbors of p_i whose timestamp is less than p_i.ts, we signify those neighbors as preceding neighbors of p_i, denoted as set P(p_i). Likewise, all neighbors of p_i whose timestamp is larger than p_i.ts, we signify those neighbors as succeeding neighbor of p_i, denoted as set S(p_i). For all data points $\in$ S(p_i), they can always be considered as neighbors of p_i no matte how window slides. Only data points $\in$ S(p_i) cause the change of total number of neighbors. In the above case, p_1 and p_3 are preceding neighbors of p_i while p_7 and p_9 are succeeding neighbors of p_i.

This observation gives us an insightful view to classify data point into three different categories based on the size of S and P. For a data point p_i, if the size of S(p_i) ¿= k, then p_i is a safe inlier. We denote I_s as the set of safe inlier. This means p_i is guaranteed to never become an outlier at any time. If the size of S(p_i) ¡ k, yet the size of P(p_i) + S(p_i) ¿= k, then p_i is an unsafe inlier. I_u is used to denote the set of unsafe inlier. This indicates when some neighbors in P(p_i) expire, pi has a chance to become an outlier if p_i is not able to find more neighbors in newly-arrived data point. Otherwise, if the number of data points in S(p_i) plus number of data points in P(p_i) are less than k. Then by applying Definition

1 here, p_i is regarded as an outlier. The set of outliers is symbolized as D.

Chapter 3

Problem Formalization

3.1 A General Problem

Generally, when an analyst is not certain about the best parameter settings for his analysis, he might submit multiple queries of the same type but with different parameter settings at the same time. Moreover, the data streaming he has been concentrating on is entirely possible being monitored by other analysts simultaneously. Therefore, efficiently sharing among computation results from both intermediate and output ones multiple of outlier detection queries that have arbitrary pattern-specific and window-specific parameters is our goal. To actualize it, the main problem we need to settle is how to share and maintain the progressive pattern in the real-time responsiveness applications.

Here is a general description of multiple queries with arbitrary pattern-specific and window-specific parameters. Given a workload of WL with n distance-based outlier detection queries Q_1(S,k_1,
r_1,w_1,s_1), Q_2(S,k_2,r_2,w_2,s_2), ...,Q_n(S,k_n,r_n,w_n,s_n) querying the same input data stream S, while all the other query parameters such as k, r, w and s differ.

As a matter of fact, all outlier detection algorithms are binding the neighbors of out-

liers they are trying to find with the data stream where outliers reside only. However, under many conditions, it is entirely possible for outliers and their neighbors coming from totally distinct multiple data streams. Therefore we categorize data stream into two types based on the demands of analysis purposes. Data stream in which all data points are the ones we would like to evaluate whether they are outliers or not is Target stream. Data stream that we probe into to see whether data points in it are neighbors of the data points in Target stream or not is Scope stream. Based on the definition of distance based outlier, Target and Scope streams can be irrelevant because what to analyze and what to probe can be totally different.

In addition, so far none of the known outlier detection technologies have been developed to support predicates sharing. However, providing another predicate type parameters, which is not the common case in outlier detection, serves to be extremely useful in practical applications. Accordingly, besides two pattern specific and window specific parameters, we aim to integrate two predicates parameters into our outlier detection algorithm dealing with multiple queries. Based on the type of data stream they are applying to, they are Target predicate and Scope predicate. It is Target predicate if the filter is put on the Target stream. Otherwise if filter is put on the Scope stream, then this predicate is Scope predicate. Again, in order to share the computations and results of the progressive pattern, it is crucial to exploit the similarities among those predicates specified by different queries to improve performance efficiency.

Figure 3.1 is the query template with the predicate parameters extended from the general outlier query template [7]. In this complete query template, the general formulation of a query with arbitrary predicate parameters as well as pattern-specific and window-specific parameters is fully illustrated. What need to be paid attention to is that Scope stream is not only restricted to simply one data stream. Disparate Scope streams can be pulled together through union operation into this template.

- Input: a query group *QG* with multiple outlier-detection queries on some input streams with arbitrary predicates and parameters.

Qi:

DETECT OUTLIERS
FROM TARGET <stream1> <var1> *[WHERE <cond on var...1>]*
WITH NEIGHBOR CONSTRAINT
FROM SCOPE <stream2> <var2> *[WHERE <cond on var...2>]*
USING COUNT = <k>
WITH DISTANCE FUNCTION <f(x)>
WITHIN RANGE = <r>
IN WINDOW = <w> AND SLIDE = <s>

- Goal: to minimize both the average processing time and the memory space needed by the system.

Figure 3.1: Query Template

3.2 A Running Example

Here we introduce a concrete example of how an outlier query with predicates can be formalized in the query template. The main purpose of the query is to detect users in HR Dept who behaves strangely in the latest hour and keep updating every one minute. The definition of strange behavior, in this query, is bound by the fact that their login or access times on 10 different machines or 10 different files should be more than 30 times.

From description above. Target stream is users. Scope stream is different machine login files and file access files. K is 30 times. R is 1o files. W is 60 minutes. S is 1 minute. Both target and scope predicate are HR Dept.

Chapter 4

Sharing Among Queries with Pattern and Window Parameters

We now propose our approach in optimizing the process of a workload of queries with arbitrary pattern-specific and window-specific parameters. Our sharing outlier processing (SOP) algorithm mainly is based on the minimization of neighbor search times, the maintenance of progressive pattern over sliding window among multiple queries and sharing on both intermediate and output results. By applying these strategies during outlier detection execution process, SOP can continuously generate an evolving result set and provide answers to queries with all possible combinations of different-pattern specific and window-specific parameters.

4.1 Varying Parameter - K

Consider the window-specific parameters and one of the pattern-specific parameters R are the same for the workload of many queries with different K values. This implies that all queries share the window size, slide size, range and require output at the same time, while

outlier data points that need to be reported for each query differ.

Assume that queries in the workload are ascendingly ordered in the light of the value of K from min to max. The total neighbor number of each data point in the workload is a constant number after the neighbor search stops. It has nothing related with different K values among queries. Based on this observation, we can infer that under the situation where only K is the variable parameter, we just need to maintain the number of neighbors equivalent to the largest K value. Once the largest K neighbors have been found, then it is sufficient to answer all the queries lined up whose K values are less. In this way, full share is thus achieved.

Status Sharing Lemma: *Given a workload WL of queries with arbitrary K parameter setting. After neighbor search stops, if data point $p_i \in I_s$ for queries with K value k_i (0¡i¡number of different k, k_{min}¡k_i¡=k_{max}) in WL, then safe status indicator of p_i will be indexed as k_i. If $p_i \in I_u$ for queries with K value k_j (0¡j¡number of different k, k_{min}¡k_j¡=k_{max}) in WL, then unsafe status indicator of p_i will be indexed as k_j. For those queries whose K value k_k¿k_j, $p_i \in$* D.

Proof: This Lemma holds because in arbitrary K case, there is an inclusion relationship based on the number of neighbors for ascendingly ordered arbitrary K queries. This pattern containment relationship enables the sharing by using a status indicator. Status indicator just records two indexes referring to threshold based on the value of K, safe status index I_s and unsafe status index I_u. As long as certain number of succeeding neighbors, say k_i, of p_i is maintained, then p_i is a safe inlier to queries whose K values are less than k_i. Therefore by setting up the safe status index at k_i, we are able to indicate the threshold that to which queries this data point is safe inlier and to which not. Otherwise, if less than k_j neighbors are collected, then based on the pattern containment relationship, status indicator can be used to at least show the divide of which queries can report this data point as an outlier through unsafe status index. More specifically, we set up I_u as k_j. Then for

all queries whose K values are greater than I_u, p_i would be outputted as an outlier, the rest of the queries in the workload will not.

Following this lemma, we just need to try to find enough neighbors for the query with greatest K value. This simplifies the multi-query sharing problem into single query problem. The only difference is for multi-query sharing, status indicator is additionally maintained. This leads to an important fact which is when to determine the termination of neighbor searching. This can be a quite decisive factor in saving system resources. This is so because if neighbor searching terminates too early, it is entirely possible that p_i can not find enough neighbors due to not touching every data point in the window. Error output might be caused by such early termination. On the contrary, if we keep searching neighbors even enough neighbors are collected, then redundant comparisons are wasting many precious resources. So late termination of neighbor searching also cause inefficiency. Therefore an inappropriate cutoff time actually significantly influences the efficiency of sharing strategy. Here we give the definition of our Least Search operation for arbitrary K case.

Definition: *Given a workload WL with arbitrary K, for each data point p_i, Least Search is the search that first search succeeding neighbors and then preceding neighbors. It will not stop searching neighbors until enough number of neighbors namely greater than or equal to k_{max} = max{k: for all k specified by queries in WL} within range R are found. Eventually, if it is unable to find k_{max} neighbors after all the data points in alive window have been compared with p_i already, then it terminates neighbor search automatically.*

The reason why SOP can use Least Search to exactly ensure the perfect intercept time is because when we are searching neighbors to collect minimum evidence [1] required, theoretically there are only two situations exist. One is that p_i have found enough neighbors. This means that minimum evidence, namely k_{max} neighbors, is collected. In this

case, those k_{max} neighbors are shared by the whole workload, therefore status indicator of p_i will show that p_i is an inlier for all queries. Continuation of neighbor search would be unnecessary. Another situation is that p_i cannot find k_{max} neighbors. In this case, we could use the status indicator to evaluate to which queries p_i is an outlier and to which ones p_i is not. In this way, neighbor search is forcefully terminated because neighbor search has probed all other data points in the window already.

Example 1 Given four queries Q_1, Q_2, Q_3, Q_4 with corresponding K values of 1, 2, 3, 4. R is 1, W is 6 and S is 1. We mainly analyze data point p_5. Figure 4.1 shows the distribution of all data points in the dataset from a distance-based perspective before and after window slides as well as all data points in the current window from a timestamp perspective.

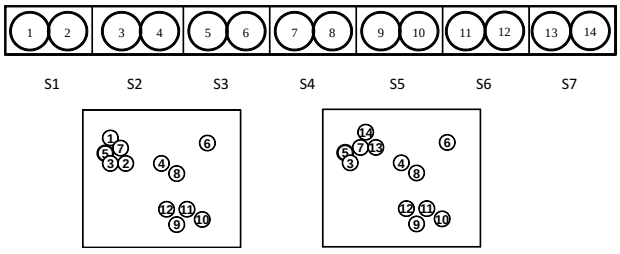


Figure 4.1: Distribution of data points

Before window slides, p_7 is the only succeeding neighbor of p_5, therefore —S(p_5)— is updated to 1 indicating the number of succeeding neighbors. Then neighbor search continues, p_1, p_2 and p_3 are found as preceding neighbors of p_5, therefore —P(p_5, s2)— is stored indicating there is a neighbor in the second slide and —P(p_5, s1)— is maintained indicating there are two neighbors in the first slide. At this moment, the search stops because the total number of neighbors satisfies 4, the greatest K value. This means it obtains sufficient evidence, which therefore makes p_5 exclude itself from the outlier list for Q_4. Accordingly, the safe status indicator is set to 4 showing that p_5 is safe for all four queries in the workload.

After window slides, p_13 and p_14 arrive as new data points. p_1 and p_2 are no longer counted as neighbors of p_5 due to the expiration of the first slide. Therefore —S(p_5)— updates to 3 and safe status indicator updates to 3, indicating that p_5 is a safe inlier for Q_1, Q_2 and Q_3. Meanwhile, unsafe status indicator is updated to 4, indicating that for Q_4 p_5 is an unsafe inlier. So again p_5 will not be outputted as an outlier for all queries.

The data structures of how we maintain those neighbor information are shown in Figure 4.2. For succeeding neighbors, only a total number is being updated all the time instead of indexes of some specific data points. However for preceding neighbors, in order to support the event-based mechanism [2] to efficiently schedule the checkups of which data points will be triggered as outliers due to window sliding, we maintain neighbor counts based on the unit of each slides. Status indicator is an additional data structure that indicates the output results that to which queries one data point is an outlier and to which is not by updating its safe inlier index and unsafe inlier index.

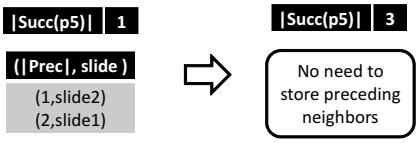


Figure 4.2: Arbitrary K: neighbor information

4.2 Varying Parameter - R

Consider the window-specific parameters and one of the pattern-specific parameters K are same for the workload of a bunch of queries with different R values. This indicates that the window size, slide size, count and require output at the same time are shared by all queries, while outlier data points that need to be reported for each query differ.

Assume that queries are descendingly ordered based on their value of R from max to min. Based on the premise that all the window specific parameters are fixed, as explained

in arbitrary K case that number of neighbors holds a inclusion relationship, this containment relationship still contributes to the sharing strategies in arbitrary R case. This means that number of neighbors found in some queries can also be shared by some other queries Q_i in the workload. More specifically, assume that there are two queries in which R value r_i of query Q_1 is less than R value r_j of query Q_2, then all the neighbors of data point p_i for Q_1 obviously can also be claimed as neighbors of pp_ii for Q_2. According to this observation, once enough neighbors is found within the smallest range, then it is sufficient to answer all the other queries with greater range value. Therefore, full sharing is achieved.

Status Sharing Lemma: *Given a workload WL of queries with arbitrary R parameter setting. After neighbor search stops, if data point $p_i \in I_s$ for queries with R value r_i (0¡i¡number of different r, r_{min}¡r_i¡=r_{max}) in WL, then safe status indicator of p_i will be indexed as r_i. Accordingly, if data point $\in I_u$ for queries with R value r_j (0¡j¡number of different r, r_{min}¡r_j¡=r_{max}) in WL, then unsafe status indicator of p_i will be indexed as r_j. For those queries whose R value r_k¿r_j, $p_i \in$* D.

Proof: This Lemma holds because in arbitrary R case, the inclusive relationship of neighbor number still holds as in arbitrary K case. For descendingly ordered arbitrary R queries, this pattern containment relationship enables the sharing strategy in number of neighbors within different range. This means if number of neighbors of one data point in the most restricted range r_{min} equivalent to k is maintained, then it is sufficient to answer the queries with R value greater than r_{min}. Otherwise, if less than k neighbors are collected for the most restricted range, then based on the pattern containment relationship, status indicator can be used to at least show the divide between which queries can report this data point as an outlier and which queries this data point is an unsafe inlier or safe inlier. More specifically, for all queries whose R value is less than or equal to safe status index, data point is safe. For all queries whose R values fall between safe status index and unsafe status index, data point is an unsafe inlier. It has a potential to become an outlier

due to expiration of its preceding neighbors at some point. Otherwise, queries whose R values less than unsafe status index will output data point as an outlier.

However, there is a difference between arbitrary K case and arbitrary R case in status sharing strategy. For arbitrary K case, neighbor sharing is bidirectional. This means queries with larger K values can share neighbors with queries with smaller K values and vice versa. Yet in arbitrary R case the neighbor sharing is unidirectional. This means only neighbors found by queries with smaller R values can share number of neighbors with queries whose R values are greater. Since the sharing in the opposite way is not applicable for arbitrary R case, data structures used in arbitrary K case can not be inherited to arbitrary R case directly. We adapt data structure that only maintains number of succeeding neighbors in a fixed range to a relation that maintains the number of succeeding neighbors in disparate range for different queries. The same adaptation can be made to the data structure that maintains preceding neighbors. So we maintain number of preceding neighbors within disparate range based on the unit of slide. Utilizing this relation, we are able to know how many neighbors we still need to find within some specific range. Also we are able to look up the exact number of succeeding neighbors being shared by certain queries. As for status indicator, it remains the same.

When we utilizing this lemma in our outlier detection process, the same as arbitrary K case, when to determine the exact timing of neighbor search termination is a key factor that significantly influences the sharing strategy efficiency. Below defined the rules of how SOP handles the search termination in arbitrary R.

Definition: *Given a workload WL consists of all queries with same window specific parameters and count K parameter but arbitrary R, it always searches succeeding neighbors and preceding neighbors later. For each data point p_i, its neighbor search will not stop until enough number of neighbors namely greater than or equal to K neighbors within range r_{min} = min{r: for all r specified by queries in WL} are found. Otherwise, after*

all data points in alive window have been compared with p_i and it is unable to find k neighbors within range r_{min}, then neighbor search terminates automatically because so far all data points have been touched.

The reason why this definition of least search process is optimal for SOP resembles previously explained reason in the arbitrary K case. After neighbor search stops, there are only two possibilities. One possibility is that pi finds k neighbors within the smallest range. Under this condition, those k neighbors are shared by the whole workload, and the status indicator of p_i will show that p_i is an inlier for all queries. Another case is that p_i cannot find k neighbors within smallest range. Under this circumstance, all data points in the window have already been touched therefore the neighbor search terminates automatically.

Example 2 Given four queries Q_1, Q_2, Q_3, Q_4 with corresponding R value of 1, 2, 3, 4. K is 3, W is 6 and S is 1. Again, this time we mainly focus on p_5. Distribution of data points is the same as in Example 1 shown in Figure 2.

Before the window slides, the succeeding neighbors of p_5 in four disparate ranges are shown in Figure 4.3. After compared with all succeeding data points, for Q_3 and Q_4 who share the same neighbors, p_5 is a safe inlier and for Q_1 and Q_2, p_5 is still an outlier, therefore the neighbor search does not terminate. So it turns back to the preceding data. When it stops, enough neighbors have been collected for Q_1. Hence p_5 is no longer an outlier for all queries. Figure 4.3 shows the preceding neighbors information. Afterwards the safe status is updated to 3 and the unsafe indicator 1.

Figure 4.3: Arbitrary R: neighbor information

After the window slides, p_{13} and p_{14} arrive as new data points. Caused by the expiration of the first slide, p_1 and p_2 can not be regarded as neighbors to any data point in the current window. Therefore p_5 loses two of its preceding neighbors, which makes it become an outlier for Q_1 and Q_2. So it keeps neighbor search until it finds two neighbors p_{13} and p_{14} within the smallest range. At this moment, four queries in the workload share the same number of neighbors and p_5 is excluded in the outlier list. Then the safe indicator is updated to 1.

4.3 Varying Parameter - K and R

Now we consider when both pattern-specific parameters change yet window-specific parameters remain the same for all queries in the workload. This means that the window size, slide size and require output at the same time are shared by all queries, while outlier data points that need to be reported for each query differ.

Because the sharing mechanism and least search process of arbitrary K and arbitrary R case use the same sharing idea and the data structure of those two cases are orthogonal. Therefore we can naturally combine previous two cases to actualize the sharing mechanism and least search process for arbitrary K and R case.

However, just combining those two cases can lead to extreme situation that consumes unnecessary resources. For instance, there is one query specifying K value as 1 and R value as 1 and there is another query specifying K value as 100 and R value as 100. In this case, least search will require to find 100 neighbors within range 1, which is the most restricted condition. This potentially creates an authentic new query with the most restricted parameter specifications. If we evaluate which data points are outliers for this new created query, extra memory and CPU resources are inevitably wasted.

To settle this newly-aroused problem, an extra table to organize different R and K

value is maintained. This table analyzes all arbitrary K and R specifications from all the queries at the first place. Then considering different R values as the primary key of table, corresponding greatest K values are then paired up to them. Accordingly, for each entry in the table, namely one R and its greatest K, it can be processed as a single arbitrary K case. For each column, it can be regarded as a single arbitrary R case. Therefore, without generating new non-existed query and exploiting extra resources consumed by that query is achieved.

Below shows the overall pseudo-code of this combination case in Algorithm 1 and 2.

Algorithm 1 SOP(p_i,D) Arrival

```
Require: Data point p_i, Dataset D, R //D is Data points in the current window, R is a set of different value of R
for each q ∈ p_i.succPoint do
    for (r_i ∈ R) do
        if (true == p_i.isNeighbor(q, r) then
            for (r_j ¿= r_i) do
                p_i.—Succ(p_i, r_j)— ++;
                if (p_i.—Succ(p_i, r_i)— ¿= p_i.getKmax(r_i)) then
                    p_i.updateStatusIndicator;
                    break;
                end if
            end for
        end if
    end for
end for
while p_i.precSlides ! = NULL and !p_i.isSafe do
    slide = getSlideWithLargestLifespan(p_i.precSlides(D));
    for each q ∈ slide do
        for (r_i ∈ R) do
            if (true == p_i.isNeighbor(q, r) then
                for r_j ¿= r_i do
                    p_i.—Prec(p_i, r_j)— ++;
                    slide.updateTriggeredList(p_i);
                    if ( (p_i.—Prec(p_i, r_j)— + p_i.—Succ(p_i, r_i)— ¿= p_i.getKmax(r_i)) then
                        break;
                    end if
                    p_i.updateTriggeredSlide(slide);
                    p_i.updateStatusIndicator;
                end for
            end if
        end for
    end for
end while
```

We take the previous example to elaborate this algorithm. It establishes a table with two entries in the first place. One entry is R equal to 1 and K equal to 1. Another one is R equal to 100 and K equal to 100. Utilizing this methodology not only prevents the process of query with extremely restricted parameter specifications, but also takes the advantage

Algorithm 2 SOP(p_i,D) Departure

Require: Data point p_i, Dataset D, R //D is Data points in the current window, R is a set of different value of R

```
1: for each p_i ∈ expSlide.triggereList do
2:     slide = getUncomparedSlide(p_i);
3:     for each q ∈ p_i.succPoint do
4:         for (r_i ∈ R) do
5:             if (true == p_i.isNeighbor(q, r) then
6:                 for (r_j ¿= r_i) do
7:                     p_i.—Succ(p_i, r_j)— ++;
8:                     p_i.updateStatusIndicator;
9:                     if (p_i.—Succ(p_i, r_i)— ¿= p_i.getKmax(r_i)) then
10:                        break;
11:                    end if
12:                end for
13:                p_i.updateTriggeredSlide(slide);
14:                p_i.updateStatusIndicator;
15:            end if
16:        end for
17:    end for
18: end for
```

of sharing strategy and least process to output outliers in real-time response for different queries with less resources being used.

In the line 7 of the Algorithm 1, a data point updates its attached status indicator each time a neighbor is found. Line 14 in Algorithm 1 starts a loop to ensure that neighbor search keeps looking into the fresh data point until enough neighbors has been found.

Figure 4.4 shows another example when the workload consists of four different queries and how the table is established after pre-analyzing.

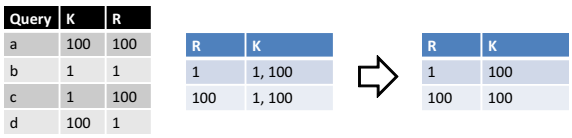


Figure 4.4: Matching table

4.4 Varying Parameter - W

Assuming all the queries start simultaneously, consider the pattern-specific parameters and one of the window-specific parameters S are the same for the workload of a bunch of queries with different W values. This implies that all the queries share the count, range,

slide size and require output at the same time, while outlier data points that need to be reported for each query differ.

Assume that queries are ascendingly ordered based on the value of W from min to max, how to maximize the sharing among queries with different window size now becomes the main focus. In terms of the definition of streaming system, window of larger size contains the window of smaller size. This means that all slides constituting smaller window are also the slides included in the larger window. The lifetime of data points in the smaller window never terminates in the larger window unless the smaller window slides and no longer keeps them anymore. It is entirely possible that for queries whose have smaller window sizes than w_{max}, p_i is outputted as an outlier due to no enough accumulating neighbors in all slides contained by those windows. Yet for queries with larger window sizes or window size being equal to w_{max}, it is entirely possible that there are more neighbors of p_i residing in other slides, which makes p_i an inlier for those queries. Based on this observation, if number of neighbors in each slide contained in the window of larger size is maintained, answering the queries with smaller W value is adequate as well.

Status Sharing Lemma: *Given a workload WL of queries with arbitrary W value. After neighbor search stops, if data point $p_i \in I_s$ for queries with W value w_i (0¡i¡number of different w, w_{min}¡w_i¡=w_{max}) in WL, then safe status indicator of p_i will be indexed as w_i. Accordingly, if data point $p_i \in I_u$ for queries with W value w_j (0¡j¡*
number of different w, w_{min}¡w_j¡=w_{max}) in WL, then unsafe status indicator of p_i will be indexed as w_j. For those queries whose W value w_k¡w_j, $p_i \in$ D.

Proof: This Lemma holds because in arbitrary W case, there is an inclusion relationship among the number of neighbors for ascendingly ordered queries. As long as pattern-specific parameters are same for all queries, then the definition to find outlier is universal in the workload. This means we just need to consider how to chop progres-

sive neighbor numbers into different slides for different window to share. Therefore, we maintain all progressive patterns, especially the number of succeeding neighbors that are used to be maintained based on unit of window, in the unit of slide. This serves to avoid duplicate neighbor search. Accordingly, once enough number of neighbors of one data point in the window with smaller window size w_{min} is maintained, then it is sufficient to answer the queries with W value greater than w_{min}. Otherwise, if less than k neighbors are collected for query with window size w_{min}, then based on the pattern containment relationship, status indicator can be used to at least show the divide of which queries can report this data point as an outlier. In this way, full share is achieved.

When following status sharing lemma in the process, identical to arbitrary case, for the arbitrary W case, sharing direction is also unidirectional. This is so because window with larger size contains slides that are not in slides that compose the smaller window. Thus only queries with smaller W value can share neighbors in each slide with the ones with greater W value. Accordingly, if there are two queries in which query Q_1 whose W value is specified as w_i containing m slides and query Q_2 whose W value is w_i containing n slides, assume w_i ¡ w_j and m ¡ n, then for data point p_i, intuitively, m slides of w_i are part of the n slides of w_j. Hence the accumulated number of succeeding neighbors of p_i found in slides m_1, m_2, ... and m_i can all be reused by w_j of Q_2 through one execution of neighbor searching driven by Q_1. Neighbors sharing on the other way does not work.

However, even status sharing lemma serves to significantly reduce resources by sharing efficiently computation among multiple queries, a bad timing of neighbor search termination still causes either defective output or over-comparisons. Therefore how to evaluate the timing of termination affects the efficiency after all.

Definition: *Given a workload WL consists with arbitrary W, for each data point p_i, its neighbor search will not stop until enough number of neighbors namely greater than or equal to K neighbors within range R in the window size w_{max} = max{w: for all w*

specified by queries in WL}. Otherwise, after all the data points in the window whose size is w_{max} compared with p_i, it is still unable to find k neighbors. Then neighbor search terminates automatically.

This optimizes the process of neighbor search is because when neighbor search stops, there are only two possibilities. One case is that p_i finds enough neighbors in the smallest window size. Under this condition, those neighbors are shared by the whole workload, and status indicator of p_i will show that p_i is an inlier for all queries. Another case is that p_i cannot find k neighbors in the smallest window size. Under this situation, based on status sharing rule, status indicator still can be used as a measure to evaluate for which queries p_i is an outlier and which ones p_i is not. In other words, all queries whose W values are smaller than the unsafe indicator will output p_i as an outlier and the rest of the queries in the workload will not.

Example 3 Given four queries Q_1, Q_2, Q_3 with corresponding W value of 2, 3, 6. K is 2, R is 1 and S is 1. We concentrate on data point p_3 and analyze how sharing strategy works under arbitrary window case. Geographical distance distribution of all data points and the window view are the same as Example 1.

Before the window slides, the succeeding neighbor of p_3 in three disparate ranges are shown in Figure 4.1. After compared with all succeeding data points in the window of Q_1, there are no neighbors. Therefore it looks back to compare with data points in the first slide. After found two neighbors which make p_3 an unsafe inlier for Q_1, for Q_2 and then Q_3 it still has some succeeding data points not compared. So it keeps searching in the non-overlapped slides contained by Q_2 and then Q_3 in order until it hits the end of the largest window. Meanwhile, for Q_3 p_3 becomes a safe inlier. All the succeeding neighbors information and preceding neighbors information are shown in Figure 4.5.

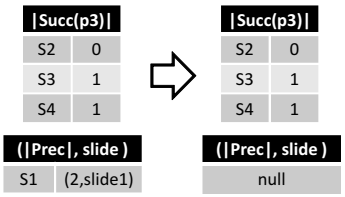


Figure 4.5: Arbitrary W: neighbor information

After the window slides, p_13 and p_14 arrive as new data points and p_1 and p_2 expire. For Q_1, we just need to add the neighbor numbers from slide 2 and 3 to see if p_3 is an outlier. The same rule is applied to Q_2. As for Q_3 with the largest number, p_3 is already a safe inlier, therefore no need to do the search again. Accordingly, the safe indicator is updated to 2 and unsafe is updated 2 indicating for Q_1 p_3 is an outlier.

4.5 Varying Parameter - S

Assume all queries start simultaneously, consider the pattern-specific parameters and one of the window-specific parameters W are the same for the workload of a bunch of queries with different S values. This implies that all queries share the count, range, window size and require output at the same time, while outlier data points that need to be reported for each query differ.

Not like the previous cases where neighbors can be shared among all the queries in the workload, in this case, different slide sizes only influence the moving unit of each window sliding and the output timing of the outlier results. According to the characteristics of window mechanism, a window is triggered by certain time duration or certain number of arriving data points, then slides forward. Hence, each time how much a window slides depends on the slide size of the query specified by different users. Therefore in order to evaluate an appropriate value to enable the sharing among a workload with queries of arbitrary slide size, a greatest common divisor is calculated based on these different S

values. This greatest common divisor is then used as the smallest unit which we call slice for a window to move forward.

In terms of the features of greatest common divisor, each distinct slide size is divisible by the slice size. This important characteristic allows us to simply use a counter to measure number of times the slice size a window has slided and then to evaluate if the corresponding slide size has been hit. Once it comes up to the time that one specific slide size has been slided over, then outliers are outputted for queries with that slide size. All corresponding maintenances also update at this time. Each time window slides, counter increases its value up one and to see if there is any need to output. Also, if the counter is accumulated to the greatest slide size in the whole workload, reset is triggered and counting starts from the scratch.

4.6 Varying Parameter - W and S

Now we consider when both window-specific parameters change yet the pattern-specific parameters remain the same for all queries in the workload. This means that the count, range and require output at the same time are shared by all queries, while outlier data points that need to be reported for each query differ.

Case of multiple slide sizes and case of multiple window sizes are orthogonal structures, so naturally combining those two strategies and data structures together does not cause heavy workload. This is because, for case of multiple window sizes, we can simplify all the maintenance down to the progressive patterns in each slide, status indicator for the whole workload and the update trigger overheads. For case of multiple slide sizes, we only need to decide the timing we are required to output outliers and when to update corresponding intermediate results. Consequently, as to each different slide size, we exploit their greatest common divisor as slice size, making slice the smallest unit we use to store

neighbor information. Therefore, case of arbitrary W and S can be regarded as arbitrary W case with fixed slide size whose value is slice size.

The pseudocode for the core routines is shown in Algorithm 3 and 4.

Algorithm 3 SOP(p_i,D) Arrival

```
Require: Data point p_i, Dataset D, W, slice //D is Data points in the current window, W is a set of different value of W
for each q ∈ p_i.succPoint(slice) do
    if (true == p_i.isNeighbor(q) then
        p_i.—Succ(p_i, slice)— ++;
        p_i.updateStatusIndicator;
        if (p_i.—Succ(p_i, w_m in)— ¿= k) then
            break;
        end if
    end if
end for
while p_i.precSlides ! = NULL and !p_i.isSafe do
    slide = getSlideWithLargestLifespan(p_i.precSlides(slice));
    for each q ∈ slide do
        if (true == p_i.isNeighbor(q) then
            p_i.—Prec(p_i, w_m in)— ++;
            slide.updateTriggeredList(p_i);
            if ( (p_i.—Prec(p_i, w_m in)— + p_i.—Succ(p_i, w_m in)— ¿= k) then
                break;
            end if
            p_i.updateTriggeredSlide(slide);
            p_i.updateStatusIndicator;
        end if
    end for
end while
```

Algorithm 4 SOP(p_i,D) Departure

```
Require: Data point p_i, Dataset D, W, slice //D is Data points in the current window, W is a set of different value of W
for each p_i ∈ expSlide.triggereList do
    slide = p_i.getUncomparedSlide(slice);
    for each q ∈ p_i.succPoint(slide) do
        if (true == p_i.isNeighbor(q) then
            p_i.—Succ(p_i, slice)— ++;
            p_i.updateStatusIndicator;
            if (p_i.—Succ(p_i, w_m in)— ¿= k) then
                break;
            end if
        end if
    end for
    p_i.updateTriggeredSlide(slide);
    p_i.updateStatusIndicator;
end for
```

The first loop in the Algorithm 4 shows that we use slice as the smallest unit to slide the window. And every time we update the number of neighbors, the timing is based on size of slice. The body of the first loop in Algorithm 5 is for triggered potential outliers to find new neighbors. During the process, slice is the smallest unit all the time and be used

as a basic slide size.

4.7 Varying Parameter - K, R, W and S

Now we consider when both window-specific parameters and pattern-specific parameters change for all queries in the workload. This means that the count, range, required output at the same time and outlier data points that need to be reported for each query all differ. This is the general case that frequently happens in the real application.

We actually can utilize a combination of previously introduced techniques to achieve the maximum sharing. This is so because for the case of arbitrary pattern-specific parameters only, the maintenance and data structure it involves in mainly is related to the number of neighbors and corresponding update triggers. This means holding all the patterns identified by them in a containment relationship. Yet for the case of arbitrary window-specific parameters only, all it refers to mainly is the size of the snapshot of the data streaming we are analyzing and the sliding frequency. Therefore, they are orthogonal from each other and can be integrated together without modification of sharing strategies and data structure.

4.8 Complexity Analysis

Computational Costs Computationally, there are two major actions that contribute to the cost of neighbor searching. We recall that, first range query compares all the data points in the window no matter if these comparisons are necessary or not for each data point. Instead of expensive cost of range query, neighbor search of SOP stops once the query with the most restricted parameter specification is satisfied. Moreover, when one data point is searching neighbors in the window, all the other data points being compared

with also maintain the neighbor information, which reduces more CPU computation costs. From this perspective, neighbor search for each data point is a constant operation. Second the cost of sharing for each data point only requires on neighbor search. For each data point, status indicators are set to indicate which queries will output the data point as an outlier or not. Thus the cost of multiple queries in the workload can be reduced from O(nk) (k is the number of queries, and n is the number of data points in the window) to O(n). Third, as for the general case, we maintain a minimum set, an organized table, to figure out the minimum k for each different r so no extra computation would be occupied.

Memory Costs The memory costs of SOP depends mainly on two factors, the relation we maintain for different range which depends on the number of queries and the intermediate computation results from the comparison of two different data points. Complexity wise, compared to non-sharing method, memory requirements are multiplied by the number of queries. SOP significantly decreases it via status sharing lemma to just one neighbor search. As for the sharing method in [2], most of the intermediate computation and event-based trigger update and maintenance are obviously reduced via least searching lemma.

Conclusion As discussed above, SOP structure maintains a minimum object set and also achieves least computation. Evidently, we do not need to hold the number of comparisons of data points equivalent to the complete window size at any stage for computing if two data points are neighbors or not, rather once the data point is a safe inlier for all, comparison stops. This is a clear win over the exiting methods for multiple queries computations that need to utilize range query from scratch.

However, we observe that the resource requirements of SOP grow with N_{values}, the number of different parameter specifications. More specifically, since SOP always searches to meet the most restricted parameters specification and maintains a relation of different R value to keep number of neighbors in different ranges, its memory and CPU consump-

tion grow with the number of queries.

Chapter 5

Sharing Among Queries with Predicate Parameters

In the preliminary section, a complete query template with outlier parameters and predicate parameters is fully demonstrated. Sharing approach on pattern and window specific parameters have been deliberated in the last section already. Integration of capability of handling predicate parameters into SOP becomes the next problem so that SOP can become more robust and pragmatic in the real application. Therefore, in this section, we shift our focus to another half of our problem, i.e. the design of processing and optimization strategies for the shared predicates in a workload of outliers detection queries. We begin with the concept of two different sets. Next, we present the intuition and the methodology of the sharing strategies for each case.

Assume that we have a workload WL consisting of a set of outlier detection queries. Each outlier query have arbitrary selection predicates on both target and scope. According to the particular role of each of these two screened sets via application of two predicate parameters, as described in previously section, we classify these two sets into two categories. For the one to see if these data points in it are outliers or not, we call the set of

these data points *Target Set*. For the one that data points are selected separately against which to be compared with data points in *Target Set* to evaluate whether they are neighbors of data points in *Target Set* or not, we call the set constituted by these data points *Scope Set*.

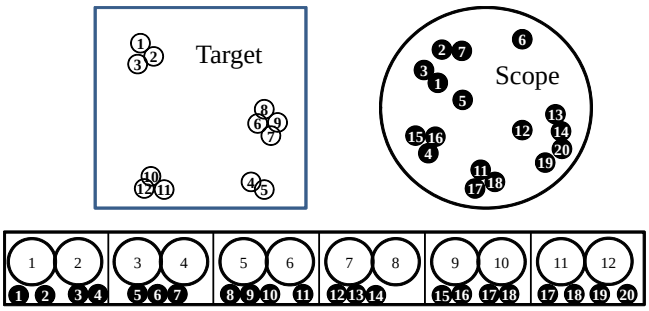


Figure 5.1: Geographic Distribution and Window View

Figure 5.1 shows geography distribution and window view of both *Target Set* and *Scope Set*. These two sets are composed of all data points specified by the corresponding predicates given by the users. *Target Set* and *Scope Set* do not have to be pertaining to each other. Nevertheless, from the perspective of the distance function, *Target Set* and *Scope Set* should be related in some ways where connections are built on their meaning in the real world determined by the analysts. This can be perceived from the running example in problem formalization section. On the other hand, they can be the same data stream.

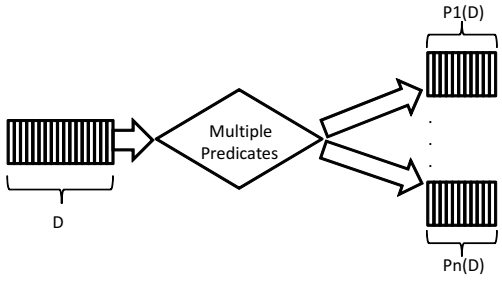


Figure 5.2: Unshared Predicates

Assume no sharing strategy is applied, then Figure 5.2 shows how method introduced earlier would process these requests. This helps to demonstrate conceptually that brute-force method causes huge inefficiency. In the first place each data point is sequenced by its time stamp when arriving system. Then based on different predicate, the input data stream D are divided into n subsets. All data points in one subset satisfies one particular predicate of one query. Thereafter, the outlier detection algorithm SOP is applied to each set of data points separately. Then for each query, the system outputs the corresponding correct result.

Simply applying SOP to each subset actually can meet the minimum demand that system can handle outlier detection queries with predicate parameters. However, usually most of predicate specifications have certain percentage of overlaps. If the workload contains a huge number of queries whose selections on this data stream differ subtly, then same computation of many times definitely wastes huge amounts of resources. Therefore, our goal in this section is to introduce sharing strategies that reduce these unnecessary costs.

5.1 Intuitions and Approaches

Given a continuous input stream, different queries in the workload will select disparate data points in it based on their own predicates. Therefore within a fixed window size, each query maintains two lists of indexes pointing to the data points in *Target Set* and *Scope Set*. These two lists dynamically update data points in it according to the expiration and arrival of data points each time window slides.

Arbitrary Target Predicate The main intuition for how we tackle the sharing problem for varying Target predicate is the following. Namely, we utilize the predicates p_1, p_2, p_3, ..., p_n to partition the data points in a window of the input stream into disjoint sub-

sets, called fragments in [3]. For each data point in each fragment, neighbors searching would be applied. In other words, a set of data points in a window of the input stream, is partitioned into F_0, F_1, ..., F_k, a set of k + 1 disjoint fragments: D= $F_0 \cup F_1 \cup$ ... $\cup F_k$. Each fragment F_i is associated with a subset of the workload WL that denoted by WL (F_i) ¡ $2^{|WL|}$ where every data point in the fragment F_i satisfies the predicates of every query in WL (F_i), and no other query. Among all these fragments, the workload WL (F_0) is an empty set. This means that all data points in F_0 satisfy none of the predicates. Thus none of these data points should participate in any query and accordingly can safely be ignored.

Meanwhile, a signature that identifies the precise subset of queries is associated with each data point. In other word, this signature of a data point encodes the fragment that it falls in and accordingly the query. This can be realized by using the bitmap to contain one bit for each of the n queries in the workload. Accordingly, when it comes to outputting timing for each query, outliers in different fragments can be aggregated in terms of the bitmap for different queries.

Example. Given a set of three queries Q_1, Q_2 and Q_3 with disparate Target predicate predicates p_1, p_2 and p_3, how queries are related with each other by 8 fragments based on these three predicates are shown in Figure 5.3. Also signatures for each fragment are also designated under the number of fragments.

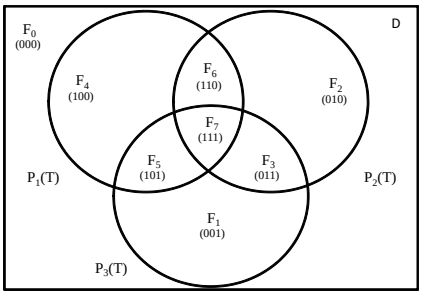


Figure 5.3: Possible Fragments

First we partition the *Target Set* into fragments. SOP can be applied in individual fragment to detect outliers. Then, we use an array to maintain this list of outliers in each fragment. Afterwards a simple add-up operation whose function is like an aggregation to accumulate all outliers in different fragments together for each query through looking up the signatures attached on those outliers. Therefore each data point just needs to apply neighbor search once for queries with different *Target Set* but same *Scope Set*, which significantly reduces the inefficiency aroused by repeated neighbor search. The basic idea of pipelining general outlier detection and aggregation is shown below.

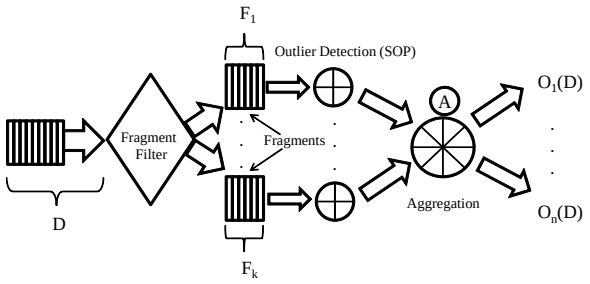
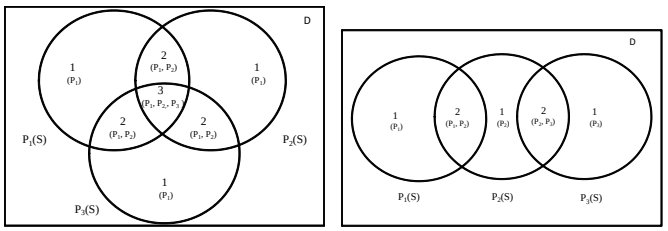


Figure 5.4: Conceptual View of Fragments Sharing

Arbitrary Scope Predicate In arbitrary Scope predicate case, we basically have a similar intuition as in arbitrary Target predicate case. Given predicates p_1, p_2, p_3, ..., p_n indicating different Scope predicates of queries in the workload, we partition the Scope data points from the input stream into disjoint subsets, called blocks. Namely these sets of data points in a window of the input stream are partitioned into a set of k disjoint blocks: D = $B_0 \cup B_1 \cup$ ... $\cup B_k$. Each block is associated with a subset of the workload WL that is denoted by WL (B_i) ¡ $2^{|WL|}$ where every data point in the block B_i satisfies the predicates of each query in WL (B_i), and no other query.

Nevertheless, in arbitrary Target case, it does not matter which fragment should be looked at first and which later. This is because all data points in different fragments have to be examined once if they are outliers or not. In arbitrary Scope case, we should prior-



(a) Case One (b) Case Two

Figure 5.5: Possible Blocks in Two Different Case

itize the principle that once we find enough neighbors, then we stop searching neighbors to reduce the resources occupied by comparisons between two different data points to the minimum. If the bitmap signature used in arbitrary Target case is applied here, then like all data points in different fragments need to do neighbor search once, all data points in the blocks need to be probed against once. Therefore a different technique is utilized to distinguish different blocks.

In order to reduce the probing times by different queries over the same blocks, we tag each block a priority. This priority functions clarifying the overlap level of different queries in the workload. In other words, each block has a priority value starting at zero. This value increases by one if one query is found to be related to it. This means more queries a block involves, higher priority it is assigned. However, only with the priority, we can not connect different blocks with their corresponding queries. Hence each block maintains a list of indexes pointing to the queries whose predicate parameters of *Scope Sets* consist of itself. Below shows the example different scenarios of block and its priority establishment.

Example Given a set of three queries Q_1, Q_2 and Q_3 with disparate Scope predicates p_1, p_2 and p_3, two possible cases of different block formulation with different priority and query lists are shown in Figure 5.5.

From the Figures above, we can infer that the number of block is not determined by the number of queries in the workload, yet is based on the condition how predicate parameters overlap. Then priority can be generated to tag each block accordingly. Moreover, for the

simplicity, the priority value is exactly the number of queries in each block's query list.

Block Search Lemma: *Given a workload WL consists of all queries with the same outlier parameters and target parameter but arbitrary scope, for each data point* p_i*, the order of which block to probe in is based on its priority value. Assume each block has its own unique priority, then the selection sequence is sorted by the descending order, starting with the highest one and ending with the lowest one. The neighbor probing select block from this sequential list from the highest to lowest accordingly. If some blocks happen to have the same priority, then they would be linked together based on their own priority values. During the process of neighbor probing, selection order of those blocks does not influence the output results. In other words, if block* B_i *has priority value equal p, then all blocks with priority values of p should be probed.*

This lemma holds because to find enough neighbors, we just need to collect k neighbors in range r. Thus once sufficient evidence is satisfied. Further search is meaningless, but consumes limited yet precious resources. If one block with unique priority contains enough evidence to make one data point an inlier, then for all queries, it is an inlier. Search does not need to proceed. Otherwise it has to go to the next block with less higher priority. However, for blocks with same priority, search has to be applied in each block separately because these blocks are associated with different Scope predicates of different queries. Therefore we can only ensure the search lemma to stop as early as possible and then go to the parallel blocks.

Here is a concrete demonstration on how SOP shares different blocks in the whole workload. We take the case where there is a block being involved with all three queries in the workload as an example. The block in the middle has priority value equal three. Thus it no doubt has the highest priority. When we probe for neighbors, we need to determine which block to look into first. As explained previous, in order to eliminate the repeated computing on same blocks, we always choose blocks with higher priority. Therefore,

block in the center would be the first block we search neighbors in. If within the range of this block, enough neighbors is found, then there is no need to look into other blocks with lower priority. However, under some circumstances, more data points in the lower priority need to be probed. So accordingly, the probing continues in the next lower priority until sufficient evidence shows that data point in *Target Set* become safe. During the walking down neighbor search in priority, it is entirely possible that different blocks are designated with same priority yet their query lists are different, like the three different blocks with priority value equal to two. If this is the case, then all these three blocks need to be probed into since they are related to different lists of queries. This means for the block with the same priority value, the order of which block to choose and then probe does not affect the final output result.

General Case The most general case is that given a workload of queries that all differ in both *Target Set* and *Scope Sets* parameter specifications, yet we now note that these specified target and scope values have some percentage of the overlaps. Due to the fact that data points in *Target Set* and *Scope Sets* can be irrelative with each other, therefore the sharing strategies introduced above are orthogonal. So after combining those two algorithms and maximize the sharing of the overlaps among predicate parameters, SOP can achieve high efficiency for outlier detection with arbitrary predicate parameters. The pseudocode for the core routines is shown in Algorithm 5.

Algorithm 5 SOP(p_i,*D*)

```
Require: Data point p_i, Dataset D, R //D is Data points in the current window, R is a set of different value of R
    for each frag_i ∈ slides.getLatest do
        for each p_i ∈ frag_i do
            for each prior_i in Prioritys.getInorder do
                p_i.neighborSearch;
 5:             p_i.updateStatusIndicator;
                if ! p_i.isOutlier then
                    break;
                end if
            end for
10:     end for
    end for
```

Here is a brief demonstration on how SOP works after put sharing approaches of two

different predicate parameters together. First we partition the set of data points D into fragments, then for each data point in each fragment we efficiently probe the block with the highest priority. If sufficient evidence is met in that block, then for all queries in both Q (F_i) and Q (B_i), the data point is regarded as an inlier. Accordingly, no further search is needed among other blocks. However, if for that data point, the number of neighbors is not sufficient to be an inlier at this point, then the neighbor search process must continue for this particular target data point probing into the blocks with the next lower priority until it can be determined that this data point satisfies the definition of an inlier. Again if more neighbors need to be found to prove that the data point at hand is not an outlier, then this it will find other fragments that its query are involved with and probe into the corresponding blocks with the sharing approach of arbitrary Scope case. Finally, either it finds enough neighbors or the query will report it out as an outlier.

5.2 Complexity Analysis

Computationally, there are two major factors that influence the cost of neighbor searching. One is the percentage of Target sharing. Another is the percentage of Scope sharing. To be more specifically, it indicates the number of fragments and the number of blocks with different priority values. Neighbor search for each data point is a constant operation. Therefore O(nk) (k is the number of queries, and n is the number of data points in the window) is the complexity of the LEOC. The best case in this scenario is the sharing is 100 percent. This means the complexity can be reduced to O(n).

Chapter 6

Performance Evaluation

6.1 Experiment Setup and Methodology

Our experiments are conducted on a PC WITH 3.4G HZ Intel Core-i7 processor and 6GB memory, running Windows 7 OS. All algorithms are implemented in JAVA on CHAOS Stream Engine.

Real Data We used real streaming data set, namely, the Stock Trading Traces Data (STT). It has one million transaction records throughout the trading hours of one day. All data has same format of name, transId, time, volume, price and type.

Synthetic Data We also implemented a data generator to create dataset containing 100M objects produced by a data generator. This dataset is composed of Gaussian distributed data points as inlier candidates and uniform distributed ones as noises. Certain percentage of random noises is distributed in each segment of the data stream.

Alternative Algorithms We compare our proposed algorithm SOP with two alternative methods. One is the state-of-the-art ACOD [1] whose method was the first mention and then to provide a preliminary attempt at supporting multiple outlier detection requests each with different parameter settings. However, [1] mainly focuses on outlier detection

of single query method, though briefly sketched a preliminary idea of shared computation for the shared K or shared R parameters in only about few paragraphs of the manuscript. They thus do not consider different window or slide sizes. Neither do they consider the generalized outlier queries with Scope and Target clauses supported by our work. Furthermore, even for the 2 parameters that they do consider, namely, K and R, they do not introduce the notion of searching only the minimum resources essentially to share outlier computation, namely, the minimal set of neighbors to be probed for each data point.

In addition, we also compare against the best known single-query strategy LEOC [2], which has been shown to be optimal in distance-based outlier detection. The method however is applicable to the outlier detection request with a single fixed parameter setting only. We choose this method not only because it is the best so far, but also because several of its core principles such as minimal probing principle and lifespan-aware prioritization principle also are able to be applied and adapted in the multiple parameter setting contexts.

Methodology We measure two common metrics for stream systems, namely the average processing time (CPU time) per window and the average memory consumption per window. The CPU time per window corresponds to the total amount of system time used to process one window before it expires. The consumed memory corresponds to the memory required to store the information mainly for each active object (i.e., preceding and succeeding neighbors), the heap size used for the events prioritization (i.e., triggered outliers), and the outliers of all the queries in one live window. All data results are collected and calculated on the unit of one window, and then have been averaged over all windows. All experiments are reported using time-based mechanism, while count-based one supports similar results.

We conduct scalability experiments to validate the performance of the proposed algorithms with increasing number of queries in the input workload. We study the performance by covering the important combinations of the six query parameters, varying

Type	Name	Value
Pattern	K	[30,1030)
	R	[200,1200)
Window	W	[1Ks,50Ks)
	S	[50s,50Ks)
Predicate	T	
	S	Location

Table 6.1: Parameters Setting of SOP

Workload	Pattern		Window		Predicate	
	K	R	W	S	T	S
(A)	arbitrary	fixed	fixed	fixed	fixed	fixed
(B)	fixed	arbitrary	fixed	fixed	fixed	fixed
(C)	arbitrary	arbitrary	fixed	fixed	fixed	fixed
(D)	fixed	fixed	arbitrary	fixed	fixed	fixed
(E)	fixed	fixed	fixed	arbitrary	fixed	fixed
(F)	fixed	fixed	arbitrary	arbitrary	fixed	fixed
(G)	arbitrary	arbitrary	arbitrary	arbitrary	fixed	fixed
(H)	fixed	fixed	fixed	fixed	arbitrary	fixed
(I)	fixed	fixed	fixed	fixed	fixed	arbitrary
(J)	fixed	fixed	fixed	fixed	arbitrary	arbitrary
(K)	arbitrary	arbitrary	arbitrary	arbitrary	arbitrary	arbitrary

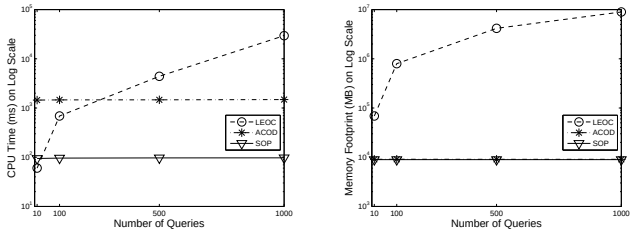
Table 6.2: Combinations of different workload

from focused specific ones to more general cases as shown in Table 1. In particular, we evaluate performance of SOP with predicates parameters by differing sharing percentage in targets and scopes. Core scenarios of our study are summarized in Table 2.

6.2 Evaluation of SOP for Varying Pattern and Window Parameters

6.2.1 Varying Pattern-Specific Parameters

We prepare four workloads with 10, 100, 500, 1000 queries respectively by randomly varying pattern-specific input parameters values (in the range shown in Table 1) for each



(a) CPU (log scale) (b) Memory (linear scale)

Figure 6.1: Varying K values on Synthetic Dataset

query, while using fixed parameters settings for all the other query parameters.

Arbitrary K In the first experiment, we analyze the effect of our proposed algorithm when compared to the-state-of-art algorithm LEOC and ACOD under different numbers of queries. We use a fixed window size of 10Ks and a slide size of 0.5Ks on our synthetic data. We keep range parameter R at 700. An appropriate value in range of R setting is shown in Table 9. K is randomly generated from the possible values in the range from 30 to 1029 for each query.

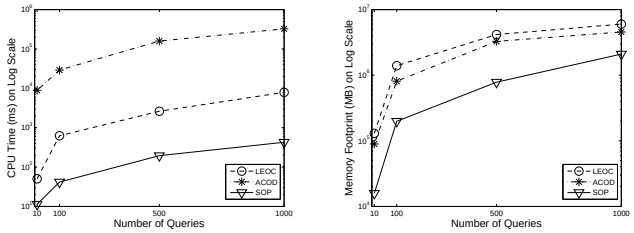
Figures 6.1 show the CPU time and memory space on logarithmic scale on y-axis by the three algorithms. Clearly, the CPU performance of our proposed method is superior to the other two. This win is because after SOP finds enough neighbors for a data point so that it is qualified to be excluded as an outlier during its life cycle for all queries, neighbor search immediately stops for that data point. As explained in the previous section, if a data point gets enough neighbors for the query with largest K value, namely the most restricted condition, for other queries specifying smaller K value, this data point definitely is qualified as a safe inlier. Therefore there is no need to keep searching. However, ACOD would compare each data point with all the other data points in the window even if that data point has already been confirmed to be safe with respect to each of the queries in the workload. As to LEOC, because of its repeatedly detecting outliers over and over for each query from the scratch, with the number of queries increasing, our gain in CPU obviously becomes more and more significant.

The trend in CPU resources utilized by our SOP is almost straight. This is because the value of k is randomly selected from the same fixed range as depicted in the parameter settings table Table 9. For each data point, only when the greatest k neighbors are found, it is capable to be labeled as a safe inlier and then exempted from further neighbor search. At least one of randomly selected k is likely to get fairly close to the upper ceiling value in the range given sufficient number of searches. And, as is apparent from method description (Section 5), the highest k value determines the overall CPU costs consumed while all smaller contained k values are gotten as by-product nearly for free. Therefore as the number of queries increases, the CPU cost in increasingly larger workload tends to be similar over time.

For the same reason, a similar gain can also be observed in memory usage, especially when compared with LEOC. However, SOP does not save much compared to ACOD. This can be explained by the fact that even though ACOD applies expensive range query searches, it does not store more than k largest neighbors for each data point. In other words, except that ACOD stores extra intermediate results for neighbor pattern update and maintenance after neighbor search stops caused by range search, essentially it stores almost the same amount of information as SOP. Yet because of the least search process, SOP still wins by a narrow margin from the perspective of memory.

Arbitrary R In this experiment, we evaluate the performance of SOP compared with ACOD and LEOC under case with parameter R varying only. In order to keep the dataset with even outlier distribution, we fix the window size to 10Ks, slide size to 0.5Ks and K to 30, while R is randomly generated from 200 to 1199.

As shown in Figure 6.2, both the CPU and memory usage of our algorithm are significantly less than ACOD and LEOC. In particular, SOP is achieving up to three magnitude times improvement compared to ACOD. Based on the sharing mechanism that if sufficient neighbors are found in the most restricted condition which means the smallest range



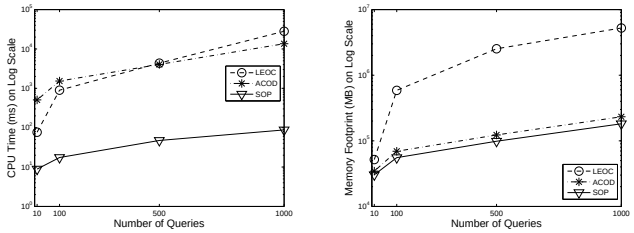
(a) CPU (log scale) (b) Memory (linear scale)

Figure 6.2: Varying R values on Synthetic Dataset

for one data point, it will be labeled as an inlier for query specifying that range. Obviously as well for all queries specifying smaller range, namely more relaxed condition, in workload. The status indicator of this data point will become showing its safe afterwards. A safe status indicator represents enough number of neighbors is gathered, capable to be shared by all queries and no more need to keep on searching. Then SOP terminates the neighbor search. However, ACOD does not stop searching until all data points in current window have been touched. As a result, extra CPU cost used to update neighbor information like number of neighbors and triggered outliers decrease its efficiency.

In addition, the trend of CPU cost is climbing as the number of queries increases. The reason is because in the process of neighbor search, in order to track number of neighbors falling in different range, for each data point, a table will be maintained by both ACOD and SOP. The maintenance of this table correlates with the size of this table decided by the number of different value of R. Consequently, CPU processing is burdened heavier as the number of queries increases.

Similar available under the memory consumption is also due to the fact that ACOD keeps neighbor search throughout the whole window. Therefore after the sufficient evidence for each data point is collected, neighbor finding process continues, which causes extra pattern maintenance and update. This maintenance and update are supposed to be small. However, we also have to update the table kept to look up the number of neighbors in different R. Every time we find a neighbor, an update of the table is executed. Because



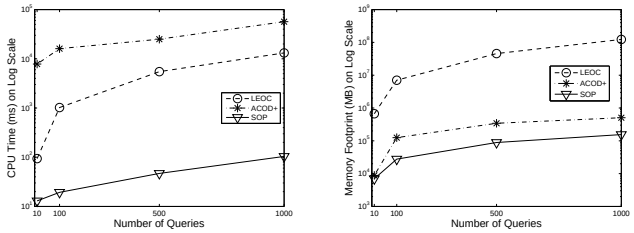
(a) CPU (log scale) (b) Memory (linear scale)

Figure 6.3: Varying K and R values on Synthetic Dataset

this is proportional to the size of different R values, more often update happens, more extra memory is used. Therefore ACOD in arbitrary R case costs much more memory space than SOP.

Arbitrary K and R In this experiment, we assess the impact of SOP with case of varying both pattern-specific parameters compared with ACOD and LEOC. Again, to keep reasonable outlier distribution in dataset, we fix the window size to 10Ks, slide size to 0.5Ks, while both k and r is randomly generated from 30 to 1029 and 200 to 1199 individually.

Figure 6.3 depicts the performance of those three algorithms via CPU cost and memory consumption. We observe that SOP utilizes less processing time and memory usage than the other two state-of-arts. This is not only caused by the reason explained in previous two single cases, but also because SOP keeps an optimization of neighbor search requirement in combining K and R. In other word, SOP analyzes the relation between R and K specified by all different queries so that when it searches neighbors for each data point, it does not aim to find greatest K within smallest R like ACOD does. Instead, from given workload, it maintains a meta data to match different R and its corresponding largest K. Served by this meta data, SOP is enabled not to apply neighbor search based on the most restricted query conditions, namely the largest K and smallest R during the outlier detection algorithm execution. Therefore more CPU cost and memory are saved. However, ACOD always applies the most restricted criteria in each range query, even that



(a) CPU (log scale) (b) Memory (linear scale)

Figure 6.4: Varying W values on Synthetic Dataset

combination of K and R might not even exist, which is always the case. As a result, ACOD wastes more CPU and memory on that compared to SOP.

6.2.2 Varying Window-Specific Parameters

Next, we focus on workloads with 10, 100, 500, 1000 queries respectively by randomly varying window-specific input parameter value (in the range shown in Table 9) for each query, while using fixed parameter settings for the other query parameters.

Arbitrary W In this experiment, we study the performance of SOP compared with LEOC and ACOD+ in the case that only w is arbitrary. We show the result with fixed value of slide size at 0.5Ks, r 200 and k 30, while the window size is varied from 1Ks to 500Ks shown in Table 1.

In Figure 6.4, our algorithm still shows a better result on CPU time and memory consumption. Our sharing mechanism on different window sizes mainly is concentrating on maintaining number of neighbors in each slide, especially for succeeding neighbors. This is quite different from LEOC since single query only requires number of succeeding neighbors based on unit of window. However, this sharing mechanism benefiting us almost three folds faster is due to the fact that if a data point is safe for the smallest window size, it certainly turns out to be a safe inlier represented by its status indicator. Obviously, once a data point satisfies its safe identity for the most restricted condition which is the smallest window size, there is no doubt that for those larger window size specified by

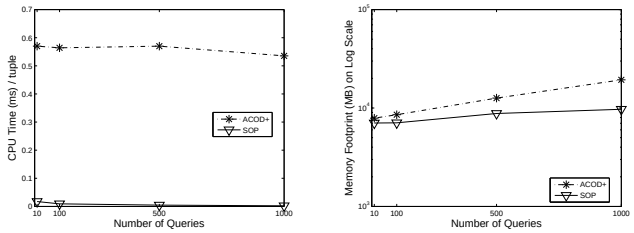
other queries, it is a safe one. Hence its number of neighbors and its status indicator can be shared in workloads and no need to search and update its neighbor information further. However, for ACOD+, even after the data point is labeled as a safe inlier, neighbor search continues until comparisons with all the other untouched data points have been finished. In this scenario, needless CPU resources are consumed.

Regarding to memory space, applying main mechanism of SOP into ACOD+ does help save memory to avoid recalculation of expensive range query. However, the side effect of the range query, namely exhausted neighbor search still costs extra memory space to update neighbor information and triggered outliers. From another perspective, though sharing strategy helps, but ACOD+ does not maximize benefit brought by this sharing strategy because when enough sharing information has gathered, search does not halt afterwards.

Arbitrary S In this experiment, we concentrate on comparing the scalability of the algorithms when varying the slide sizes only. Window size is set at 50Ks, k is 30 and r is 200, while slide size is varied from 50s to 50Ks as shown in Table 1. Figure only shows the performance of SOP and ACOD. This is because case of arbitrary S only influences the outlier output timing and the moving unit each time window slides. Therefore, only slight effect on the maintenance and update is between LEOC and SOP.

As presented in Figure 6.5, the outcome again clearly shows that with respect to the CPU consumption, SOP only takes 0.01742s to process each object on average, while A-COD+ needs 0.57s for each object. This is as expected because SOP eliminates redundant comparisons among different data points via keeping safe inliers from expensive neighbor search.

In particular, the CPU time of SOP on each data point decreases from 0.0116 to 0.0021 as the number of queries increases. The reason is evident. Larger size introduces more different slide sizes. More different values of slide size are there in one workload, the



(a) CPU (log scale) (b) Memory (linear scale)

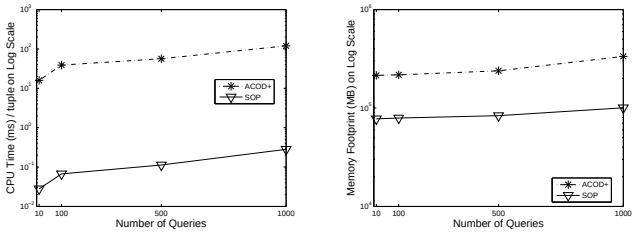
Figure 6.5: Varying S values on Synthetic Dataset

smaller greatest common divisor is. Our sharing strategy will use this greatest common divisor to process window sliding mechanism on streaming data. In other words, value of the greatest common divisor determines the basic unit for SOP to process and store preceding neighbors and triggered outliers. Considering that triggered outliers are stored by basic unit and the fact that smaller unit size gives rise to less number of triggered outliers. Trend of CPU cost per data point will apparently go down as number of queries increases.

Again our method is not only more desirable in CPU but also in memory utilization. With the increase in number of queries, more storage resources will be distributed on when to output outlier and how many basic units, namely the greatest common divisor, we need to move forwards to avoid extra neighbor searching. Therefore, this rising trend is exactly what we anticipate.

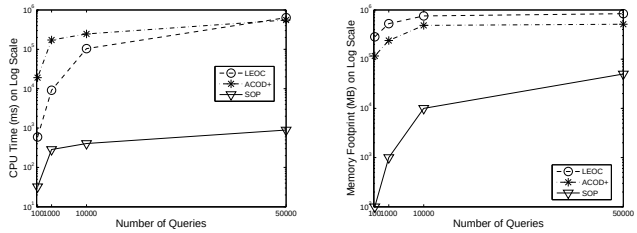
Arbitrary W and S In this experiment, we investigate the effectiveness of SOP under the case of varying both window-specific parameters compared with ACOD+. We do not compare with LEOC is because for LEOC, CPU process time consumed by each data point are almost the same, which is available to be referred in other experiments as a constant value. Therefore we remove it from this experiment. To enable outliers to be even distributed, we use k as 30 and r as 200, while window size and slide size are arbitrarily selected from the range of 1Ks to 500Ks and 50s to 50Ks respectively.

As illustrated in Figure 6.6, the CPU time consumed by SOP per tuple increases from



(a) CPU (log scale) (b) Memory (linear scale)

Figure 6.6: Varying W and S values on Synthetic Dataset



(a) CPU (log scale) (b) Memory (linear scale)

Figure 6.7: Varying K, R, W and S values on Synthetic Dataset

0.028 to 0.282 as number of queries increases from 10 to 1000. This cost is still three orders of magnitudes less than cost of alternative algorithms. Clearly, results shown in these two Figures are combined from two previous cases of arbitrary w and arbitrary s. Because of the fact that these two cases are orthogonal as previously explained, impact of the combination will not influence with each other.

6.2.3 Varying Pattern and Window Specific Parameters

In this general case, we prepare workloads with 10, 100, 1000, 10000, 50000 queries respectively by selecting all input parameters (in the ranges shown in Table 1 for each query. We examine the behavior of SOP compared with ACOD+ and LEOC in this experiment.

Observations can be drawn from Figure 6.7 that our approach SOP significantly outperforms the unshared LEOC and state-of-art ACOD+ in processing multiple queries. As previously shown, SOP achieves a tremendous gain in CPU utilization. Its constantly hundreds times faster than ACOD benefiting from the fact that not all objects need to be investigated once they become safe. The status indicator can only be turned on as safe for

all queries only if for the most restricted condition, which is greatest K, smallest R and smallest window size, the data point is qualified as an inlier. An inlier shares all neighbor information to all queries in the workload, therefore no more search need be conducted by SOP as ACOD+ does. Based on this mechanism, SOP saves much CPU processing.

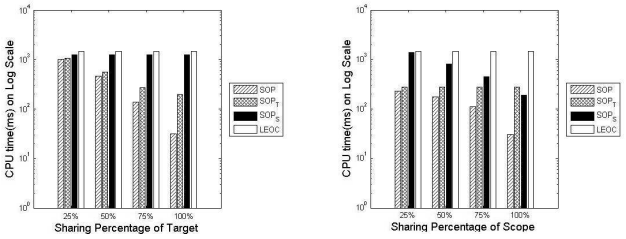
As the number of queries rises, overlap among different queries will increases. Therefore the neighbor information of each data point can be shared to a larger percentage, meaning more running time is saved by our sharing strategy. In addition, as previous explained, our two sharing strategies on pattern-specific and window-specific are orthogonal. Therefore their combination contributes to excellent scalability of our algorithm.

The memory usage also consistently exhibits stable improvement compared to the alternatives solutions. The reason is as previously stated that LEOC has to specifically detect outliers over the same streaming data over and over for every query, hence memory used by different queries adds up as the number of queries grows. Consequently, as the number of queries grows, the space consumption saved could be more significant. With respect to ACOD, it always executes the distance computations among all data points each time range query is executed. Accordingly it stores preceding neighbors for all data points as well as update triggered outliers. However, SOP only stores the number of preceding neighbors in each slide for unsafe inliers and outliers, and shares enough number of neighbors for all queries. As a result, SOP avoids unnecessary space to keep intermediate and redundant results. Memory utilization is reduced in this strategy.

6.3 Evaluation of SOP For Varying Predicates

6.3.1 Varying Target Sharing Percentage

In this experiment, we prepare four workloads with around 25%, 50%, 75% and 100% Target sharing percentage respectively varying target predicates over the real dataset.



(a) Target (b) Scope

Figure 6.8: Varying Predicate Sharing Percentage on Synthetic Dataset

Meanwhile, Scope sharing percentage is fixed at 50%, K, R, W and S are fixed at 30, 700, 10Ks and 0.5Ks individually. This is mainly achieved by varying the location which is used for filtering the streaming data. We compare the execution time across 4 algorithms (SOP, SOP_T, SOP_S, LEOC) based on the sharing approach discussed in the previous section, in which SOP_T only shares target and SOP_S only shares scope.

As shown in Figure 6.8 (a), as the sharing percentage of Target increases, CPU processing time of both SOP_S and LEOC keep the same, while SOP_T and SOP decrease. This is what we expected. Because for each query with different Target predicates specification, LEOC and SOP_S always have to execute our base outlier detection algorithm over the whole dataset again and again. The time of outlier detection execution equals the number of queries. Therefore, sharing strategy on Target has nothing to do with CPU efficiency of LEOC and SOP_S. However, if Target are shared as SOP_T and SOP do, higher Target sharing percentage, more overlapping area of Target, hence for data points in Target, more queries those data points can be shared to. Therefore, this sharing strategy greatly reduces the time of outlier detection execution over the whole workload. Especially when the sharing percentage is up to 100%, this means all queries share the same Target areas. As a result, if same Scope area is shared, CPU processing is almost the same as outlier detection on single query.

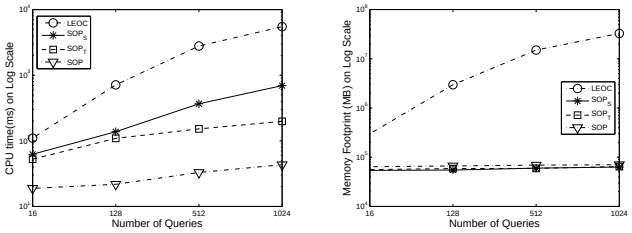
6.3.2 Varying Scope Sharing Percentage

In this experiment, we prepare four workloads with around 25%, 50%, 75% and 100% Scope sharing percentage respectively by varying the scope predicates over the real dataset. Meanwhile, Target sharing percentage is fixed as 50%, K, R, W and S are fixed at 30, 700, 10Ks and 0.5Ks individually. This is mainly achieved by varying the location which is used for filtering the streaming data. We compare the execution time across 4 algorithms (SOP, SOP_T, SOP_S, LEOC) based on the sharing approach discussed in previous section.

It is obvious to observe from Figure 6.8 (b) that as now we share the Scope, SOP_T and LEOC who do not apply sharing strategy have a constant CPU processing time, independent of the Scope sharing percentage. This is because even for queries specifying the same Scope area, they always have to execute outlier detection several times, which is the number of queries. Consequently, their CPU time have no relevance to the varying sharing percentage on Scope. On the contrary, for SOP_S and SOP that use Scope sharing strategy, their CPU process time decrease as their Scope sharing percentage increase. The reason is because the higher sharing percentage on Scope, more overlapping area of Scope, hence more queries can share the same searching area. Under most scenarios, data points will find enough neighbors by only searching highest priority area in Scope and then are labeled as safe inliers. Consequently, this safe status indicators are shared by all queries whose Scope consist of the highest priority area. Therefore CPU time is saved from this sharing strategy.

6.3.3 Scalability on Predicates

We now consider workloads with 16, 128, 512 and 1024 queries respectively representing our main problem. In these workloads, we examine the performance of SOP compared



(a) CPU (log scale) (b) Memory (linear scale)

Figure 6.9: Varying Predicates on Synthetic Dataset

with SOP_T, SOP_S and LEOC under case with both Target and Scope sharing percentage are fixed at 50%, K, R, W and S are fixed at 30, 700, 10Ks and 0.5Ks individually.

As shown in Figure 6.9, SOP is always the best performer on CPU time. Then it comes to SOP_T, SOP_S and LEOC accordingly. The reason is well explained in detail in previous experiments as to the comparisons between SOP and SOP_T, SOP and SOP_S, SOP and LEOC. The reason that why SOP_T costs less CPU processing time than SOP_S is because the former shares data points needing to find enough number of neighbors, and the latter shares the area that needed to be searched by target data points. This means sharing Target reduces the time of each base outlier detection algorithm execution, however, sharing Scope can only share the searching result, reducing the searching times in the same Scope. In the other word, sharing on Target has more effect on CPU time than sharing on Scope. Another observation drawn from Figures is that as the number of queries increases, all four algorithms show an increasing CPU consumption time. This is as expected because at a fixed sharing percentage of Target and Scope, more number of queries means more target data points are specified to search neighbors in Scope and more information to distribute the results of outliers to different queries need to be calculate. As a result, all these extra processing cause the increases in CPU cost.

Chapter 7

Related Work

With the occurrence of streaming data generated by accurate digital equipments, outlier detection on streaming environments has been extensively studied [6,7,11]. Most outlier detection methods that have been developed through previous research efforts, nevertheless, are focused on processing single mining requests [7,17,18] for streaming pattern detection only. The existing state-of-the-art algorithms [2] handle multiple queries of arbitrary pattern-specific parameters, but their naive sharing strategies demand high CPU and memory resources. Moreover, predicates, essentially for outlier specifications, have never been paid attention to.

In the beginning, [9, 10, 11] extend the outlier detection domain from static data to streaming data. They propose to use a simpler threshold variation in distance-based outliers. They all consider about the lifetime [20] of each data point and the impact of each data point on other alive data points. [9] leverages the fact that all neighbors of data point p_i that arrive after p_i will not expire before p_i is removed from the alive window. For some neighbors, their influence as a neighbor never disappears, while for other neighbors, they naturally depart before the entire lifetime of p_i ends. Based on this observation, three types of data can be categorized, namely outliers, unsafe inliers and safe inliers.

This status can be decided by checking the number of different types of neighbors found within a specified range.

Later, in order to further improve the CPU cost and reduce the memory consumption, four algorithms in [2] of continuous outlier monitoring over streaming data are introduced. They use range query to search neighbors for each data point in the current stream window and then identify outliers from the dataset. They integrate an event-based rationale to efficiently schedule and reduce the number of checks when a data point expires. However, the expensive range query search applied to every single new data tuple still gives rise to performance degradation.

Seeing the optimization opportunity in [2], [1] casts light on several critical insights to drive down the CPU costs by over three orders of magnitude with almost the same memory utilization. The concept first is to exploit the literal notion of what constitutes an outlier. Outliers are only very small numbers of objects against the entire huge data set. Hence, this limited resources can be concentrated on serving the minority of outliers and then unnecessary computation can dramatically be reduced. Furthermore, they take advantage of the property of streaming data that later data points always have a more decisive influence than data points that have arrived earlier. Consequently, they present two principles, which are "minimal probing" and "lifespan-aware prioritization", assisting in abandoning the exhaustive range query in the process of searching. Nonetheless, their methods are restricted to a single outlier query, not keeping up with requirements from modern streaming systems that similar outlier queries always come concurrently in the regular scenario. Therefore, standing on the foundation that this algorithm is the optimal one in the outlier detection of single query, we propose to extend this approach to multiple queries. Afterwards, we optimize it to share resources among different query specifications to the maximum.

As for existing multiple queries of outlier detection, [2] develops an algorithm from its

single query outlier detection method. It maintains certain amount of neighbors that can satisfy the most restricted condition which is the greatest k and smallest r. Then it filters the results with respect to each query to provide the corresponding outliers. However, these provided approaches are only for cases with arbitrary r and k in distance-based outlier specification, without cases with arbitrary w and s related to the streaming system. Moreover, another limitation of this algorithm is that the base algorithm of the single query it is extended from is outperformed by the latest algorithm presented in [1].

[24] presents shared execution strategies for processing a huge workload of the same type neighbor-based pattern mining requests with arbitrary parameter settings. It first proposes an incremental pattern representation specified by queries with different pattern-specific parameters in a single compact structure to enable integrated pattern maintenance for multiple queries. Then it introduces a meta query strategy that compacts multiple queries with different window-specific parameters into a single query by leveraging the overlaps among sliding windows. When combining those two strategies together, it also executes the range query search during the sharing process, which has been proved to be extremely expensive and unnecessary when considering the rarity property of outlier. However, the techniques it presents still can not be used in our problem.

[3] presents an algorithm to share resources by exploiting similarities in the streaming aggregate queries with differing periodic windows and arbitrary selection predicates. In the varying selection predicates part especially, it introduces the idea of fragments where streaming data is divided and categorized. Then it uses a signature to identify the uniqueness of each fragment and maintain the associated queries for each fragment. The signature is implemented by a bitmap containing one bit for each queries in the workload to confirm the relations between fragments and queries. However, though its mechanism is insightful in sharing predicates in streaming environments, the domain it can be applied to is exclusively concerning the aggregation, therefore incapable to be exploited in

the outlier detection.

Chapter 8

Conclusion

Outlier detection is increasingly used in data streaming systems as critical infrastructure for monitoring application. It serves as a helpful and difficult technique accordingly. We use least searching and status sharing strategies for efficient shared processing of a huge workload of outlier detection queries over streaming windows. Besides the common sharing part, namely pattern-specific and window-specific parameters in multiple queries, SOP also integrates the predicate as one of the parameters. Also it achieves its significant resource sharing by analyzing the parameter settings at the query level and assigning the signature to every unique fragment and block. Our experimental studies based on both real and synthetic streaming data exhibit the clear superiority of SOP to the state-of-the-art algorithms. Also SOP is confirmed with excellent scalability in terms of capability of handling thousands of queries under high speed input streams in our experiments. An intriguing future direction is merging more predicates into the parameters that can be varied so that queries of outlier detection can have a more relaxed restriction on its format and more resources can be shared.

Chapter 9

Bibliography

[1] L. Cao, Q. Wang, J. Wang and W. Yu, E. A. Rundensteiner, "Scalable Distance-Based Outlier Detection over High-Volume Data Streams,", In VLDB, 2013.

[2] M. Kontaki, A. Gounaris, A. N. Papadopoulos, K. Tsichlas, and Y. Manolopoulos. "Continuous monitoring of distance-based outliers over data streams," In ICDE, pages 135−146, 2011.

[3] S. Krishnamurthy, C. Wu, M. J. Franklin. "On-the-Fly Sharing for Streamed Aggregation," In SIGMOD, 2006.

[4] A. Shastri, D. Yang, E. A. Rundensteiner and M. O. Ward, "MTopS: Scalable Processing of Continuous Top-K Multi-Query Workloads", In CIKM, 2011, pp. 362−274.

[5] E. M. Knorr and R. T. Ng, "Algorithms for mining distance-based outliers in large datasets," In VLDB, pages 392− 403, 1998.

[6] F. Angiulli and F. Fassetti, "Detecting distance-based outliers in streams of data," In CIKM, pages 811−820, 2007.

[7] D. Yang, E. A. Rundensteiner, and M. O. Ward, "Neighbor-based pattern detection for windows over streaming data, In EDBT, pages 529−540, 2009.

[8] S. Ramaswamy, R. Rastogi, and K. Shim, "Efficient algorithms for mining outliers

from large data sets" in SIGMOD Conference, 2000, pp. 427−438.

[9] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, "A framework for clustering evolving data streams" In VLDB, pages 81−92, 2003.

[10] S. Thomspon. The twitter eurovision results. http://www.
wallblock.co.uk/.

[11] M. Kontaki, A. Gounaris, A. N. Papadopoulos, K. Tsichlas, and Y. Manolopoulos, "Continuous monitoring of distance-based outliers over data streams" In ICDE, pages 135−146, 2011.

[12] Wang, S., Rundensteiner, E. A., Ganguly, S., and Bhatnagar, S. 2006. State-Slice: New paradigm of multiquery optimization of window-based stream queries. In Proceedings of the VLDB Conference. 619−630.

[13] Zhang, R., Koudas, N., Ooi, B. C., and Srivastava, D. 2005. Multiple aggregations over data streams. In Proceedings of the ACM SIGMOD Conference. 299−310.

[14] Hammad, M. A., Franklin, M. J., Aref, W. G., and Elmagarmid, A. K. 2003. Scheduling for shared window joins over data streams. In Proceedings of the VLDB Conference. 297−308.

[15] LI, J., Maier, D., Tufte, K., Papadimos, V., and Tucker, P. A. 2005. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. SIGMOD Rec. 34, 1, 39−44.

[16] Arasu, A. and Widom, J. 2004. Resource sharing in continuous sliding-window aggregates. In Proceedings of the VLDB Conference. 336−347.

[17] Cao, F., Ester, M., Qian, W., and Zhou, A. 2006. Density-based clustering over an evolving data stream with noise. In Proceedings of the SDM Conference.

[18] Chen, Y. and Tu, L. 2007. Density-based clustering for real-time stream data. In Proceedings of the ACM KDD Conference. 133−142.

[19] Arasu, A., Babu, S., and Widom, J. 2006. The cql continuous query language:

Semantic foundations and query execution. VLDB J. 15, 2, 121−142.

[20] STREAMINSIGHT, M. 2012. Microsoft streaminsight query engine. http://msdn.microsoft.com/en-us/library/ee362541.aspx.

[21] D. M. Hawkins. Identification of Outliers. Springer, 1980.

[22] F. Angiulli and F. Fassetti. Dolphin: An efficient algorithm for mining distance-based outliers in very large datasets. TKDD, 3(1), 2009.

[23] S. D. Bay and M. Schwabacher. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In KDD, pages 29−38, 2003.

[24] D. Yang, E. A. Rundensteiner, and M. O. Ward, "Shared Execution Strategy for Neighbor-Based Pattern Mining Requests over Streaming Windows", In VLDB, 2009, pp. 224−238.