Sharing-Aware Outlier Analytics over High-Volume Data Streams

Lei Cao[†], Jiayuan Wang[‡], Elke A. Rundensteiner[‡]
[†]IBM T.J. Watson Research Center Yorktown Heights, NY 10598, USA
[‡]Worcester Polytechnic Institute Worcester, MA 01609, USA
caolei@us.ibm.com, jwang1|rundenst@cs.wpi.edu

ABSTRACT

Real-time analytics of anomalous phenomena on streaming data typically relies on processing a large variety of continuous outlier detection requests, each configured with different parameter settings. The processing of such complex outlier analytics workloads is resource consuming due to the algorithmic complexity of the outlier mining process. In this work we propose a sharing-aware multi-query execution strategy for outlier detection on data streams called SOP. A key insight of SOP is to transform the problem of handling a *multi-query outlier* analytics workload into a *single-query skyline* computation problem. We prove that the output of the skyline computation process corresponds to the minimal information needed for determining the outlier status of any point in the stream. Based on this new formulation, we design a customized skyline algorithm called K-SKY that leverages the *domination relationships* among the streaming data points to minimize the number of data points that must be evaluated for supporting multi-query outlier detection. Based on this K-SKY algorithm, our SOP solution achieves minimal utilization of both computational and memory resources for the processing of these complex outlier analytics workload. Our experimental study demonstrates that SOP consistently outperforms the state-of-art solutions by three orders of magnitude in CPU time, while only consuming 5% of their memory footprint − a clear win-win. Furthermore, SOP is shown to scale to large workloads composed of thousands of parameterized queries.

Keywords

Outlier;Stream;Multi-query

1. INTRODUCTION

Motivation. Nowadays, high-speed data-intensive stream monitoring applications ranging from credit card fraud detection, network intrusion prevention, stock investment tactical planning to telephone fraud detection [16] necessitate the extraction of outliers from huge volumes of stream data in a near real-time fashion.

In recent years distance-based outlier methods [12, 18, 2, 5] have been widely adopted for the detection of outliers in high volume stream data due to their simplicity and insensitivity to concept drift. The distance-based outlier model initially proposed in [12] originates from the basic notion of capturing abnormal phenomena in data which can be traced back to the seminal work by Hawkins [10]. In [10] Hawkins states that outliers can be identified based on assessing the similarity among points in a dataset. More specifically an outlier is an object O with fewer than k neighbors in the dataset D, where a neighbor is defined to be any other object in D that is within a distance range r from object O.

To discover abnormal phenomena from streaming data using this model of distance-based outliers, a set of input parameters have to be specified by the analyst. For example, when monitoring the potential credit fraud in bank transaction streams, analysts may look for unusual transactions whose values *in recent days* significantly differ from those of *the majority of* transactions made by peers at similar income levels. To detect such outliers utilizing the distance-based outlier technique, several parameter values have to be appropriately set. These include the r parameter that defines the notion of *significance* in transaction value dissimilarity and the k parameter that determines the *majority* of the peer transactions. In addition sliding window semantics [3] need to be applied to ensure that the outliers are continuously detected based on the *most recent* portion of the input stream only. Out-of-date information, such as the bank transactions made a long time ago, are typically no longer relevant for the interpretation of the recent outlier detection results. Thus they should be purged from the active stream window. Therefore the sliding window specific parameters, such as the *window size* and the *slide size*, also have to be specified by the analyst.

A stream outlier detection system may need to handle a large number of such parameterized queries for a variety of reasons. First, multiple analysts monitoring the same input stream may submit their outlier search requests with different parameter settings. In the credit fraud monitoring application, with a number of analysts monitoring the same bank transaction stream [16] − each may customize their outlier search requests by tuning their parameter settings using their personalized interpretation of abnormity. Even a single data analyst may submit multiple queries with distinct parameter settings, because determining apriori the most effective input parameters is difficult − if not impossible − especially when faced with an unknown or widely fluctuating input stream. In streaming environments, the continuous nature of streaming data emphasizes the necessity of supplying the most appropriate input parameters settings early on. Otherwise, the opportunity to accurately discover outliers in the segment of the stream gone by may be permanently lost. Furthermore, each analyst may provide their own personalized understanding of how large the "most recent" portion of the data to be considered is essential. Hence they may submit multiple outlier detection requests with different window related

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2882920

parameter settings ranging from a short-term minute long view to a long-term weekly or even monthly perspective.

For these plethora of reasons, a large number of outlier detection queries with different pattern and window related parameter settings may simultaneously need to be processed by the system. Thus, a stream processing system must be able to accommodate a large outlier analytics workload composed of hundreds or more of requests covering many, if not all, major parameter settings of an outlier query, and thus striving to capture the most valuable outliers in the stream.

Limitations of the State-of-the-Art. Although efforts have been made in developing efficient algorithms for distance-based outlier detection on data streams [7, 13, 1], these algorithms focus on handling *one single outlier request* with a fixed parameter setting. The simultaneous execution of multiple requests with varying *pattern* and *window* specific parameters remains largely unexplored.

Unfortunately LEAP [7], the state-of-the-art solution for handling a *single* outlier detection request over data streams, is not sufficient when applied to a large analytics workload composed of hundreds or more of such requests. This is caused by the high demand on both computational and memory resources to support each query in the workload *independently*. Our experiments confirm that LEAP indeed cannot meet the real time responsiveness requirements when handing a large workload (Sec. 6). It takes around 100 seconds to support a medium size workload (hundred queries) on a medium speed data stream (100KB per second).

While [13] also focuses on distance-based outlier detection for a single parameter setting, it also discusses an initial multiple outlier query solution. However this work is limited in that it only considers queries for a variation of pattern-specific parameters. The window-specific parameters are completely overlooked. Even for this pattern-specific parameter case, [13] continuous to rely on conducting routinely expensive range query searches that acquire all possible neighbors for all queries. Yet as already shown in [7], such range-query based solutions struggle to handle even one single outlier request when dealing with high volume streams [16].

Challenges. To handle a large workload composed of hundreds or even thousands of outlier requests over data streams in real time, effective processing strategies must be devised that cleverly shared the system resources.

This is complicated by the fact that requests with different parameter settings may cause totally different outliers to be identified. First, given a data point *p*, the evidence needed to prove its outlier status, i.e., whether it is an outlier or an inlier, with respect to distinct outlier interpretations (parameter settings) can differ. Clearly, variations in pattern-specific parameters lead to distinct criteria in defining the affinity among data points. Furthermore, due to variations in window-specific parameters even the data subpopulation of the data stream each query is posed against can differ dramatically.

Our Proposed SOP Approach. In this work we propose a innovative approach, called SOP, that efficiently handles an outlier analytics workload composed of a large number of outlier detection requests with arbitrary parameter settings, while still guaranteeing that each data point is processed *only once*.

As foundation of our solution we make the important observation that a workload composed of multiple outlier requests with arbitrary parameter settings can be correctly answered by directly utilizing *one single skyband query* [17] − a generalization of the well known *skyline* concept. Better yet we prove that given one data point, the *skyband* points discovered by the skyband query are the *minimal* yet *sufficient* evidence required to prove its outlier status with respect to *all* outlier queries in the workload.

Second, we design a customized skyband query algorithm called K-SKY that leverages the unique *domination relationships* among the streaming data points that presents their relative importance to the detection of outliers. We prove that K-SKY is guaranteed to discover all skyband points by examining only the *minimal number* of data points. Furthermore by prioritizing the processing order of the data points in the stream based on their arrival time, K-SKY naturally takes advantage of the common data populations among the data points covered by the active windows of queries with *distinct window-specific parameters*. This way redundant skyband point computations are eliminated. In short our K-SKY algorithm is optimal in minimizing the number of data points evaluated when processing an outlier analytics workload.

Based on the K-SKY algorithm our integrated strategy called SOP (Sharing-Aware Outlier Processing) achieves full sharing of both CPU computations and memory utilization for the processing of a workload of outlier queries. Computation-wise, in each active window, SOP only requires a single pass through the batch of new data points. That is, for each point we run *one single skyband query to answer all queries* in the workload. Memory-wise, SOP integrates all skyband points into a single compact data structure called LSKY. This assures that only *one single copy* of the neighbors shared across all queries is maintained.

Our experimental studies on real data streams demonstrate that SOP successfully drives down the CPU costs by over *three orders of magnitude* with significantly less memory utilization compared to [13, 7] for a rich diversity of scenarios. Furthermore, it is the only known method that scales to huge workloads composed of thousands of outlier requests.

Contributions. Our contributions include:

1) Our SOP framework is the first to tackle the problem of shared execution of multiple outlier requests with arbitrary pattern and window specific parameters in the stream context.

2) The key innovation of SOP is to transform the multi-query outlier problem into a single-query skyband problem. The output of the skyband query is proven to be minimal yet sufficient for determining the outlier status of each point for any parameter setting on the workload.

3) Our customized skyband algorithm is tuned to process outlier requests with diverse parameter settings. K-SKY is proven to be optimal in the number of points being evaluated.

4) Leveraging the commonality and dominance among the data populations, we are able to utilize one specific skyband query to support multiple queries with varying window specific parameters. By this full sharing is achieved across the query windows.

5) Our extensive experiments demonstrate that SOP routinely achieves three orders of magnitude or more speed up over the state-of-the-art methods [13, 7].

2. PROBLEM DEFINITION

Distance-Based Outliers in Sliding Windows. We first review the notion of distance-based outliers [12]. We use the term *data point* or *point* to refer to a multi-dimensional tuple in a data stream. The function $dist(p_i, p_j)$ denotes the distance between data points p_i and p_j. Distance-based outlier detection uses a range threshold *r* to define the *neighbor* relationship between any two points p_i and p_j.

Definition 1. *Two points p_i and p_j in a dataset D are said to be **neighbors** if $dist(p_i, p_j) \leq r$ with r a range threshold.*

The function $nn(p_i, r)$ represents the number of neighbors a data point p_i has within range *r*.

Definition 2. ***Distance-based Outlier.*** *Given a distance threshold r and a count threshold $k(k > 0)$, a data point p_i is regarded as an* ***outlier*** *in dataset D if $nn(p_i, r) < k$.*

Distance-based outlier detection only works for data sets for which an appropriate distance function can be defined. While this is straightforward for numerical attributes, textual or categorical attributes would typically have to be transformed into numerical attributes using existing methods in the literature [19].

We use $q_i(r_i, k_i)$ to denote the outlier detection query q_i with r_i and k_i the range and count thresholds respectively.

We focus on periodic sliding window semantics as proposed by CQL [3] and widely used in the literature [13, 7]. Such semantics can be either time or count-based. In both cases, each query q_i has a window size $q_i.win$ (either a time interval or a tuple count) and a slide size $q_i.slide$. For time-based windows each window W_c of q_i has a starting time $W_c.T_{start}$ and an ending time $W_c.T_{end}$ = $W_c.T_{start}$ + $Q.win$. Periodically the current window W_c slides, causing $W_c.T_{start}$ and $W_c.T_{end}$ to increase by $q_i.slide$. For count-based windows, a fixed number (count) of data points corresponds to the window size $q_i.win$. The window slides after the arrival of $q_i.slide$ new data points. The arrival time of point *p* is denoted as $p.time$. If $W_c.T_{start} \leq p.time < W_c.T_{end}$, *p* falls into W_c.

Outliers will be generated based on the points that fall into the current window W_c, also called the population of W_c. A point p in W_c might have a different outlier status (outlier or inlier) in the next window W_{c+1} if it is still alive in W_{c+1}, since each window has a different population.

Definition 3. ***Distance-Based Outlier Detection In Sliding Windows.*** *Given a stream S, a streaming distance-based outlier detection query $q_i(r_i, k_i, win_i, slide_i)$, q_i continuously detects and outputs the outliers in the current window W_c after the window slides from the previous to the current window W_c.*

Multiple Outlier Detection Optimization. Outlier detection requests on the same input stream can have arbitrary settings on all four parameters *r*, *k*, *win*, and *slide*. The set of outlier requests q_i that must be concurrently processed is denoted as query group $\mathbb{Q}$. Each query q_i in $\mathbb{Q}$ is called a member query of $\mathbb{Q}$. Our goal is to minimize both the processing time and the memory space needed to answer all queries in a large outlier analytics workload.

3. VARYING THE DISTANCE-BASED OUTLIER PARAMETERS

In this section we first introduce our transformation of processing a workload composed of queries with varying *r* but fixed *k* parameters into a skyband query. Then we present our K-SKY algorithm that supports such skyband query with optimality. Next we extend K-SKY to handle outlier detection queries with arbitrary *k* and *r* parameters. In this section we assume all queries share the same sliding window parameters *win* and *slide*.

3.1 K-SKY: Varying Parameter - r

Given a query group $\mathbb{Q}$ with *varying r* but *fixed k* parameters, the goal is to design an approach that supports all member queries in $\mathbb{Q}$ with each point *p* of data stream *S* processed only once in each current window W_c. The key insight here is that given such a query group $\mathbb{Q}$ and one data point *p* in current window W_c of stream *S*, the output of one single customized K-skyband query is sufficient yet necessary to determine the outlier status of *p* with respect to all queries in $\mathbb{Q}$.

3.1.1 From Multi-Query Outlier Workloads to Single Query Skyband Processing

K-skyband query is a generalization of the well known *skyline* concept. As defined in [17] a K-skyband query reports all points that are *dominated* by no more than K points. The case K = 0 corresponds to a conventional skyline. The key idea underlying this skyline concept is to define the *domination relationship* between any two data points. As a simple example consider a dataset *D* composed of *n* one dimensional data points, namely *n* distinct values $\{p_1, p_2, p_3, ..., p_n\}$. Assume the *domination relationship* between any pair of data points p_i and p_j ($1 \leq i, j \leq n$) is defined as p_i *dominates* p_j if $p_i > p_j$. Then the K-skyband (K=2) query on dataset *D* returns the top-3 largest points in *D*: $\{p_{max}, p_{max-1}, p_{max-2}\}$. p_{max-2} is dominated by the two data points p_{max} and p_{max-1}, while all other data points in *D* are dominated by at least these three top-3 points of *D*.

To map our problem of determining the *outlier status of a given point p* to the K-skyband problem, we have to similarly define the *domination relationship* between any pair of data points in the dataset D_{W_c}, i.e., the population of the current window W_c. The key observation here is that given any two points p_i and p_j, two key factors, namely their relative *arrival time* and the *distance* to the point *p* under evaluation, determine whether p_i is more important than p_j in terms of evaluating the outlier status of *p*.

Let us introduce a query group $\mathbb{Q}$ used in the remainder of this section. Assume we have a query group $\mathbb{Q}$: $\{q_1(r_1), q_2(r_2), ..., q_m(r_m), q_{m+1}(r_{m+1}), ..., q_n(r_n)\}$ [1], where r_m represents the *r* parameter of query q_m. The *r* parameter of q_1, q_2,...,q_n monotonically increases, that is, $r_1 < r_2 < ... < r_m < r_{m+1} < ... < r_n$.

Distance Dimension. In distance-based outlier definition (Def. 2), points in a dataset *D* are classified either as outliers or inliers. Thus, the process of identifying outliers in *D* is equivalent to the process of finding and eliminating inliers from it. By Def. 2, *p* is guaranteed to be an inlier once *k* neighbors are acquired in *D*. Given two points p_i and p_j, assume $dist(p_i, p) < r_m < dist(p_j, p) < r_{m+1}$. Then p_i is the neighbor of *p* with respect to query subset $\mathbb{Q}_i = \{q_m, ..., q_n\}$, while p_j is the neighbor of *p* only with respect to query subset $\mathbb{Q}_j = \{q_{m+1}, ..., q_n\}$. $\mathbb{Q}_i \supset \mathbb{Q}_j$. In other words p_i satisfies the neighbor requirement of more queries than p_j. For the evaluation of *p*, p_i is more important than p_j, because p_i makes the outlier status of *p* closer to be determined with respect to all queries in $\mathbb{Q}$ than p_j. In this perspective p_i *dominates* p_j.

On the other hand, assume $r_m < dist(p_i, p) < dist(p_j, p) < r_{m+1}$. Then p_i and p_j are both neighbors of *p* for the same set of queries $\{q_{m+1}, ..., q_n\}$. In this scenario p_i and p_j equally affect the outlier status of *p* although $dist(p_i, p) \neq dist(p_j, p)$. Based on this observation we now are ready to re-define the distance function $dist(p, p_i)$ so to normalize the distance between data points. The *original* distance function is denoted as $dist_o(p, p_i)$ instead.

Definition 4. *Given a query group $\mathbb{Q}$: $\{q_1(r_1), q_2(r_2), ..., q_m(r_m), q_{m+1}(r_{m+1}), ..., q_n(r_n)\}$ with $r_1 < r_2 < ... < r_m < r_{m+1} < ... < r_n$, $dist(p, p_i) = m + 1$ if $r_m < dist_o(p, p_i) \leq r_{m+1}$ for $0 \leq m \leq n$ with r_0 defined as $-\infty$ and r_{n+1} defined as ∞.*

By Def. 4, $dist(p_i, p) = dist(p_j, p)$ if $r_m < dist_o(p_i, p) < dist_o(p_j, p) < r_{m+1}$. This new *normalized distance* calculated using Def. 4 now accurately represents the importance of each data point to *p*.

[1]For the ease of readability, we only list those parameters in the query notation $q_i(r, k, win, slide)$ that vary. In this case $(k, win, slide)$ would be removed from q_i, since only parameter *r* is a variable.

Time Dimension. In the streaming context the presence of the time dimension further complicates matters. In particular we cannot simply claim that one data point p_i closer to p impacts the status of p more than the other points. Instead the arrival time of the data points also has to be taken into consideration. A point p_i that arrived later in the window may have a more *decisive* impact on the outlier examination process compared to an earlier arriving p_j even if p_i is not closer to p than p_j. This is so because the *younger* a data point p_i is, the longer its neighbor relationships (if any) with p will persist into the future.

Domination Relationship. We now define the *domination relationship* between the pair of points in dataset D_{W_c} that takes both the distance and time dimensions into consideration.

Definition 5. ***Domination Relationship.*** *Given a query group $\mathbb{Q}$: $\{q_1(r_1), q_2(r_2), ..., q_m(r_m), q_{m+1}(r_{m+1}), ..., q_n(r_n)\}$ with $r_1 < r_2 < ... < r_m < r_{m+1} < ... < r_n$, point p_i **dominates** p_j with respect to point p if: (1) $p_i.time > p_j.time$; (2) $dist(p, p_i) \leq dist(p, p_j)$ ($p_i, p_j \in D_{W_c} - p$) and $p \in D_{W_c}$; (3) $dist(p, p_i) \leq n$, with dist() the normalized distance of $\mathbb{Q}$ defined in Def. 4.*

In other words, given a data point p_i, p_i dominates another point p_j only if p_i expires later than p_j from window W_c (Condition 1) and it is not further away from p than p_j (Condition 2). The third condition in the domination rule filters out any data point p_i that is not a neighbor of p for any query in $\mathbb{Q}$. As otherwise this p_i would never be influencing the outlier status of p.

Based on the domination relationship defined in Def. 5, the outlier status of p with respect to all queries in $\mathbb{Q}$ can now be correctly answered based on the skyband points delivered by one single $(k-1)$-skyband query denoted as Q^s, namely the K-skyband query with K specified as k-1 [2].

Lemma 1. *Given a query group $\mathbb{Q}$, for any data point p, the output of the skyband query Q^s corresponding to $\mathbb{Q}$, denoted as $\mathbb{S}_p$, is **sufficient** and **necessary** to continuously determine the outlier status of p with respect to all queries in $\mathbb{Q}$.*

While the formal proof of Lemma 1 can be found in Appendix A, below we sketch the key ideas of the proof.

Sufficiency. The sufficiency of this mapping is based on two observations, namely the *KNN* observation and the *K-distance* observation as explained below.

KNN Observation. First, Q^s always returns the *k nearest neighbors* of p as part of the skyband points. The k nearest neighbors of p denoted as $kNN(p)$ are k points in D_{W_c} that do not have larger distance to p than any other point in D_{W_c}. The proof of this observation is intuitive. Given any point $p_i \in kNN(p)$, at most $k-1$ points in D_{W_c} are closer to p than p_i. By the domination relationship defined in Def. 5, at most $k-1$ points in D_{W_c} dominate p_i. Therefore p_i is a skyband point of our skyband query Q^s.

K-distance Observation. Second, once $kNN(p)$ is discovered, the outlier status of p with respect to each query in $\mathbb{Q}$ can be determined by examining the distance between p and its kth-nearest neighbor called $k\text{-}distance(p)$. If $r_m <$ k-distance(p) $\leq r_{m+1}$, then p is guaranteed to be an **outlier** for queries $\{q_1,q_2,...,q_m\}$ and an **inlier** for queries $\{q_{m+1},...,q_n\}$.

Justifying this observation is straightforward. If k-distance(p) $\leq r_{m+1}$, then all points in $kNN(p)$ are neighbors of p for queries $\{q_{m+1},...,q_n\}$. Therefore p is an inlier for such queries. On the other hands, since k-distance(p) $> r_m$, p does not have k neighbors for queries $\{q_1,q_2,...,q_m\}$. Otherwise the points in $kNN(p)$ would not be the k nearest points to p in D_{W_c}. Thus p is an outlier to queries $\{q_1,q_2,...,q_m\}$.

[2]For simplicity this notation does not reflect p and $k-1$.

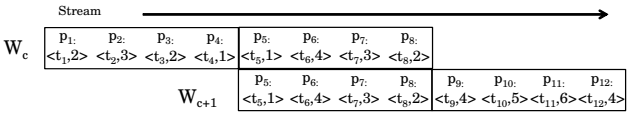


Figure 1: Sliding window stream

Next we illustrate these two observations with an example.

Example 1. *Given a query group $\mathbb{Q}$: {q_1(1),q_2(2),q_3(3)} with the k parameter set as 3 and the dataset D_{W_c} composed of points p_i represented in the arrival time and distance space ($< t_i, d_i >$): {$p_1 :< t_1, 2 >$,$p_2 :< t_2, 3 >$, $p_3 :< t_3, 2 >$,$p_4 :< t_4, 1 >$, $p_5 :< t_5, 1 >$, $p_6 :< t_6, 4 >$, $p_7 :< t_7, 3 >$, $p_8 :< t_8, 2 >$ } as shown in Fig. 1. Here t_i indicates the arrival time of p_i ($t_1 < t_2 <... <t_8$) and d_i indicates the distance of p_i to p. The $(k-1)$-skyband query $\mathbb{Q}^s$ with k = 3 will return {$p_4 :< t_4, 1 >$, $p_5 :< t_5, 1 >$, $< p_7 : t_7, 3 >$, $< p_8 : t_8, 2 >$} as the skyband points in window W_c. A subset of this result, namely {$p_4 :< t_4, 1 >$, $p_5 :< t_5, 1 >$, $p_8 :< t_8, 2 >$} is the kNN of p. The k-distance of p thus is 2. By the k-distance observation we can correctly derive the outlier status of p. Namely p is an outlier for q_1, while being an inlier for q_2 and q_3.*

Necessity. Note in the above example since the skyband point $p_7 :< t_7, 3 >$ is not in the kNN(p) set of W_c, p_7 is not utilized to evaluate p in W_c. However p_7 arrived later than p_4 and p_5 in kNN(p). Potentially it might still benefit the evaluation of p in the future windows.

As shown in Fig. 1 when the window slides from W_c to W_{c+1}, $< t_4, 1 >$ will expire. Since all new arrivals p_i ({p_9, p_{10}, p_{11}, p_{12}}) in W_{c+1} are far from p, namely $dist(p, p_i) > 3$, now p_7 will be in kNN(p) = {$< t_5, 1 >$, $< t_7, 3 >$, $< t_8, 2 >$} of W_{c+1}. As the third nearest neighbor of p, the distance between p_7 and p $dist(p_7, p) = 3$ will be utilized to determine the outlier status of p. Now p is an outlier for q_1 and q_2, while being an inlier only for q_3.

3.1.2 The K-SKY Algorithm

Although the traditional *K-skyband* algorithms could be applied to support our Q^s query [17, 21], we now design a customized algorithm called K-SKY that more efficiently supports the multiple outlier detection queries compared to existing algorithms [17, 21]. K-SKY encompasses two optimization principles, namely *time-aware prioritization* and *least examination*, that leverage the unique properties of the domination relationship among the streaming points shown in our outlier detection context. K-SKY is proven to be optimal in minimizing the number of data points to be evaluated in the skyband point discovery process.

Time-Aware Prioritization Principle. Given a data point p_i only two attributes are considered in the domination relationship of our skyband problem, namely the distance to a certain point p and the arrival time of p_i as defined in Def. 5. Furthermore, in sliding window streams the data points are naturally ordered by their arrival time. In other words, all data points can effectively been considered to be sorted on their arrival time attribute upon arrival. Therefore K-SKY effectively only needs to consider one attribute (distance to p) in the skyband point discovery process. By the definition of the domination relationship, later arrivals will never be dominated by the earlier arrivals. Leveraging this property we prioritize the order in which the K-SKY algorithm processes the data points. More specifically K-SKY always conducts the search with a later arriving data points first order. By this if one data point is not dominated by more than k points in the distance attribute and thus considered to be a skyband point, then it is not necessary to eval-

uate it again. This is so, because it will be guaranteed to never be dominated by other points evaluated later. Thus all skyband points can be discovered in one pass over the data set.

Better yet, given a data point p_i with $dist(p_i, p)$ no larger than the smallest *r* value r_1 in $\mathbb{Q}$, if p_i has already been dominated by *k* points when evaluated, K-SKY can be terminated immediately. This is so because all remaining (unevaluated) points would be dominated by at least these *k* points that dominate p_i. Therefore K-SKY can safely terminate without even examining all points.

Least Examination Principle. Second, in the sliding window context, the K-SKY search is applied in two situations. First, any *new* point p that just arrived in the current window W_c needs K-SKY to figure out its skyband points in the current window. Second, an existing point p needs K-SKY to update its skyband points when the stream slides to the current window W_c. In the first situation, for a newly arriving point p, K-SKY has to be conducted *from scratch* to search for the needed information of p. Instead in the second situation the key observation here is that given the skyband points of the window W_{c-1}, to acquire the skyband data points of a new window W_c, only a small fraction of data points in W_c need to be evaluated, namely the new arrivals and the unexpired skyband points of W_{c-1}.

This is so because any existing data point p_i in W_c could not possibly be a skyband point in window W_c if p_i is not also a skyband point in W_{c-1}. If p_i is not listed in the *skybandPoints* set of W_{c-1}, p_i must be dominated by at least k data points p_j in *skybandPoints*. By the domination rule defined in Def. 5, if p_j dominates p_i, $p_j.time > p_i.time$. This indicates p_j would not expire earlier than p_i. If p_i is still valid in window W_c, p_j would also remain valid. Therefore in W_c, p_i could not possibly be a skyband point, since it is still dominated by at least *k* data points.

Algorithm 1 K-SKY($p, W_c.plist, p.skyband, \mathbb{Q}$)

```
Output: skybandPoints //the k-1-skyband point set
1: if p.skyband == NULL then
2:     W_c.input = W_c.plist; // New point;search from scratch
3: else
4:     expireSkyband(p.skyband)
5:     W_c.input = p.skyband + W_c.pList.new; //Old point; search in new
       arrivals and unexpired skyband points
6: end if
7: for each p_i ∈ W_c.input from W_c.input.tail to W_c.plist.head do
8:     d = dist(p, p_i);
9:     if (TRUE == p_i.skyEvaluate(d,skybandPoints, ℚ)) then
10:        p.updateOutlierStatus(ℚ);
11:    else
12:        if d ≤ ℚ.r_min then
13:            break;
14:        end if
15:    end if
16: end for
```

K-SKY Algorithm. Next we show how K-SKY detects the (k-1)-skyband points in each window W_c. The skyband is computed every time when the window moves. In other words, the K-SKY algorithm is called after we receive a batch of new points based on the slide size. As shown in Alg. 1, the points of window W_c are stored in a list structure $W_c.plist$. When the streaming data arrives, the later arrivals are appended at the tail of $W_c.plist$. Therefore the points in $W_c.plist$ are naturally ordered by their arrival time. If *p* is a new point of W_c, then the search has to be conducted from scratch (Lines 1,2). Otherwise based on our *Least Examination* optimization principle K-SKY only search in the new arrivals and the unexpired skyband points of *p* (Lines 3-5).

Then guided by our *time-aware prioritization* optimization principle, K-SKY evaluates the data points of the input list $W_c.plist$ in the order from tail to head, i.e., via a "last come, first served" order (Line 7). After calculating the distance between *p* and some data point p_i, K-SKY evaluates whether p_i is dominated by at least *k* already discovered skyband candidate points. If not, p_i will be inserted into the candidate point set *skybandPoints* (Line 9). Otherwise if p_i is not a skyband point and $dist(p_i, p)$ is not larger than the smallest r parameter r_1 in $\mathbb{Q}$, K-SKY terminates (Lines 12,13). This is so because again by our *time-aware prioritization* principle, the remaining points does not have chance to be skyband points.

Leveraging the time-aware prioritization and least examination optimization principles, K-SKY is able to discover all skyband points by scanning the data set at most once. In other words, K-SKY is a *one pass* algorithm. Furthermore, it may terminate without even seeing all data points. We now show that it is *optimal* in minimizing the number of points being evaluated in the execution process.

Lemma 2. ***Optimality.*** *K-SKY correctly discovers the (k-1)-skyband points in window W_c by examining only the minimum number of data points.*

The proof of Lemma 2 can be found in Appendix B.

The skyEvaluate Algorithm. The complexity of K-SKY relies on the number of points being evaluated and on the cost of evaluating each point, that is, the cost to determine whether a given point p_i is a skyband point or not. This decision is computed by the subroutine $skyEvaluate$ of K-SKY. Since the number of points examined by K-SKY has already been proven to be minimal, the reduction of the second cost per point is now critical for high-performance of K-SKY. For this we must design an efficient $skyEvaluate$ algorithm.

Algorithm 2 skyEvaluate(d,skybandPoints, $\mathbb{Q}$)

```
Output: isSkyband; //Boolean: skyband point or not
1: layer = skybandPoints.getLayer(d);
2: count = 0;
3: for i =1; i++; i ≤ layer do
4:     count += skybandPoints.layerCount(i);
5: end for
6: if count ≤ k - 1 then
7:     skybandPoints.map(p_i);
8:     return true;
9: else
10:    return false;
11: end if
```

First we introduce the *core data structure* of the K-SKY algorithm called *LSky*. *LSky* is a layered data structure that stores the skyband points acquired in the execution process of K-SKY. It plays a critical role in assisting $skyEvaluate$ to effectively determine whether a point p_i is a skyband point.

In *LSky*, skyband points are organized into a layered two dimensional structure that preserves the order among the skyband points in both the distance and the time dimensions. As shown in Fig. 2, the points in each layer have the same distance to point *p* based on the normalized distance function in Def. 4. The points in the upper layer always have a smaller distance to *p* than the points in lower layers. Furthermore, in each layer the points are ordered based on their arrival time with the earliest arrival being at the head. By this, skyband points can be quickly expired when the window slides forward in time.

As shown in Fig. 2, given a data point p_i, skyEvaluate first calculates which layer it belongs to (Line 1). More specifically, given a query group $\mathbb{Q}$: $\{q_1(r_1), q_2(r_2), ..., q_m(r_m), q_{m+1}(r_{m+1}), ..., q_n(r_n)\}$ with $r_1 < r_2 < ... < r_m < r_{m+1} < ... < r_n$, a point p_i should be mapped to the layer corresponding to r_m (bucket B_m) if $r_{m-1} < dist(p, p_i) \leq r_m$. This can be done in logarithmic time in the number of buckets using a binary search.

Next, $skyEvaluate$ evaluates whether a point p_i is a skyband point. Since K-SKY processes data points in the "last come, first served" order, a point p_i to be inserted into LSky is guaranteed to be dominated by the points falling in the same layer with p_i and the points within its upper layers. If in total there are fewer than *k* such points in LSky when p_i is processed, then p_i will be a skyband point (Lines 6 - 8). This can be easily determined by explicitly maintaining the cardinality of each layer (Lines 3 - 5).

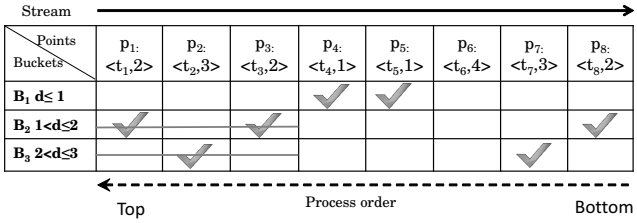


Figure 2: Skyband Point Search With LSky

Next we utilize an example to demonstrate how K-SKY detects the skyband points with the assistance of the LSky structure.

Example 2. *Given the stream and the queries in Example 1, point p_8 is processed first by K-SKY as shown in Fig. 2. Since 1 $< dist(p, p_8) = 2 \leq 2$, by Def. 4, p_8 is hashed into bucket B_2. The next point processed by K-SKY is p_7. Correspondingly p_7 is inserted into bucket B_3 because $2 < dist(p, p_7) = 3 \leq 3$. Point p_6 will be excluded from the skybandPoints set immediately since $dist(p, p_6) = 4$ is greater than the largest r parameter in query group $\mathbb{Q}$. Points p_5 and p_4 instead will be inserted into bucket B_1. By Def. 4, p_3, p_2, and p_1 should be hashed into buckets B_2, B_3, and B_2 correspondingly. However all of them are excluded from the LSky structure, since they are dominated by at least 3 data points. For example when we hash p_3 into bucket B_2, there are already 2 points in B_1 and 1 point in B_2. Therefore p_3 is dominated by 3 points and thus it is not a skyband point. In this example the skyband points are $\{< t_4, 1 >, < t_5, 1 >, < t_7, 3 >, < t_8, 2 >\}$.*

Complexity Analysis. With the assistance of the $LSky$ structure the overall complexity of K-SKY is $O(LB)$ with *L* the number of the points examined in K-SKY and *B* the number of the layers (buckets) visited when evaluating whether a point is a skyband point. We have already proven that K-SKY is optimal in *L*. Now let us assume that the points are uniformly distributed among all layers of $LSky$, then on average *B* equals to $\frac{|r|}{2}$, where $|r|$ represents the number of unique *r* parameter values in query group $\mathbb{Q}$. Given a window with $|W|$ data points, the complexity of processing the whole window therefore is $O(|W|L\frac{|r|}{2})$.

3.2 Handling Various K and R Parameters

We now relax our problem to consider varying not only k but also r parameters. One simple approach to handle a set of outlier detection queries with arbitrary pattern related parameters k and r would be to divide this workload into groups, each of which contains queries with the identical k parameter value. This then would simplify our problem into a *multi-skyband query problem* with only the k parameter varying. Intuitively our problem then could be handled by directly applying K-SKY on each group of queries. However this solution requires the independent identification and maintenance of the skyband points for each group of queries. Since a large number of skyband points are likely to be shared across these skyband queries, this naive solution inevitably leads to significant wastage of CPU and memory resources. We now tackle this shortcoming.

3.2.1 Sharing-Aware Multi-Skyband Solution

Next we propose a sharing-aware solution that efficiently solves this *multi-skyband query problem*. By maintaining the skyband points in one integrated LSky structure, given a point p_i, only one single skyband point evaluation operation is required to correctly answer all skyband queries. This way we assure that multiple skyband queries are supported by K-SKY, while still guaranteeing that each data point is evaluated exactly *only once*.

Given a query group $\mathbb{Q}$, $\mathbb{Q}$ is partitioned into sub-groups $\mathbb{Q}_j$: $\{(r_1, k_j), (r_2, k_j), ..., (r_n, k_j)\}$ $(1 \leq j \leq max, k_1 < k_2 ... < k_{max}, r_1 < r_2 ... < r_n)$. The member queries in each sub-group $\mathbb{Q}_j$ share the same k parameter value (k_j). Therefore each $\mathbb{Q}_j$ corresponds to one skyband query Q_j^s.

The key idea here is that our K-SKY algorithm can handle any number of skyband queries with distinct *k* parameter values with only some slight adjustment in the criteria used to determine whether a point p_i is a *skyband point* of at least one Q_j^s .

Definition 6. ***Skyband Point Rule.*** *p_i is a **skyband point** if:*
(1) p_i is hashed into some bucket B_m;
(2) $k' = \sum_{j=1}^{m} | B_j | < k_{max}$; and
(3) dist(p,p_i) $\leq max\{r_n \, of \, \mathbb{Q}_j \mid \forall k_j > k'\}$.

The first two conditions in Def. 6 correspond to the *examination rule* of the single query case except for replacing the k parameter of the single query with k_{max} (the largest k parameter value in $\mathbb{Q}$). However not all points satisfying these two conditions would be in the skyband point set. Now p_i is dominated by k' points. By the definition of k-skyband query, p_i would not be a skyband point of query Q_j^s unless k' is smaller than the *k* parameter k_j of queries in $\mathbb{Q}_j$ (the query sub-group corresponding to Q_j^s). Furthermore, any point p_i will be discarded by query Q_j^s if $dist(p, p_i)$ is larger than r_n of $\mathbb{Q}_j$ by the *domination relationship* defined in Def. 5. The above two conditions are captured by Condition 3 in Def. 6.

Optimality. Based on the discussion of our *time-aware prioritization optimization* principle it can be easily shown that K-SKY discovers all skyband points of multiple skyband queries in one pass over the data set. Here we provide the intuition. The processing order of K-SKY guarantees that the points added to *skybandPoints* during the execution of K-SKY would not be replaced later. Therefore similar to the case of processing a single skyband query, K-SKY correctly discovers the (k-1)-skyband points for *all* skyband queries with only the *minimum* number of points evaluated.

Complexity Analysis. Similar to handling a single skyband query, the complexity of K-SKY handling multiple skyband queries is determined by the number of the points being evaluated (*L*) and the number of the layers (*B*) that exist in $LSky$. Given a window with $|W|$ data points, the complexity of processing the whole window therefore is $O(|W|LB)$.

3.2.2 Outlier Detection With K-SKY

After acquiring the skyband points, these points then can be utilized to determine the outlier status of *p* with respect to each member query in $\mathbb{Q}$. For example, this can be done by first calculating the k-distance of *p* with respect to each query sub-group $\mathbb{Q}_j$ and then applying the *k-distance observation* (Sec. 3.1.1). However the key observation here is that to determine the status of *p*, this extra process is superfluous. In fact, we observe that the outlier status of *p* can be naturally derived as part of the *skyband point discovery* process as explained below.

Inlier Rule. Suppose we have a query sub-group $\mathbb{Q}_j$: $\{q_1(r_1, k_j), q_2(r_2, k_j), ..., q_m(r_m, k_j), q_{m+1}(r_{m+1}, k_j), ..., q_n(r_n, k_j)\}$. When

evaluating whether p_i is a skyband point, if point p_i is found to be dominated by $k_j - 1$ points and mapped into bucket B_m of LSky, then *p* is guaranteed to be an inlier for a subset of queries in $\mathbb{Q}_j$: $\{q_m, q_{m+1}, ..., q_n\}$. This is so because all points dominating p_i are as close as p_i to p. Since p_i is mapped to bucket B_m, $dist(p_i, p) \leq r_m$. Therefore p_i along with all points dominating p_i (in total k_j points) are neighbors of *p* for $\{q_m, q_{m+1}, ..., q_n\}$. By the outlier definition in Def. 2, *p* is thus an inlier for these queries.

As shown inliers are naturally recognized during the process of evaluating whether a point is a skyband point without introducing any extra overhead. This logic can be seamlessly applied in K-SKY (see Line 11 in Alg. 1) to mark *p* as inlier for the corresponding queries. Eventually *p* will be reported as outlier for those queries that do not mark *p* as inlier after K-SKY terminates.

Safe Inlier in Sliding Stream Windows. Furthermore, given a point *p* its outlier status might not always need to be evaluated in *each window* and against *every query* in $\mathbb{Q}$. Potentially skyband points discovered in the current window might provide sufficient evidence to prove that *p* is an inlier during its *entire remaining* life for a particular subset of queries in $\mathbb{Q}$, regardless of the characteristics of the future incoming stream. In this case, we would name *p* a guaranteed *safe inlier* with respect to a query subset $\mathbb{Q}_{safe}$ of query group $\mathbb{Q}$. This property arises due to the time order relationship among stream data points.

Safe Inlier Condition. We observe that *p* is guaranteed to be a *safe inlier* if the point p_i that triggers the above *inlier rule* arrives later than *p*. By the *domination relationship* definition in Def. 5, for each of the $k_j - 1$ points p_j that dominate p_i, $p_j.time > p_i.time > p.time$. In other words, all k_j neighbors of *p* would have arrived later than *p* and in turn would not expire before *p*. Therefore the neighbor relationship between *p* and p_i persists during the entire life of *p*. *p* is thus a safe inlier.

Evaluating the above *safe inlier condition* in K-SKY is straightforward. Namely when monitors the satisfaction of the *inlier rule* (Line 11 in Alg. 1), we also compare the arrival order of p_i and *p*.

Once *p* is determined to be a safe inlier, it is no longer necessary to evaluate *p* for $\mathbb{Q}_{safe}$: $\{q_m, q_{m+1}, ..., q_n\}$ in any future window. Thus the discovery of safe inliers can significantly improve the CPU and memory efficiency of K-SKY.

Next we demonstrate with an example how K-SKY determines whether a given point *p* is an outlier with respect to query group $\mathbb{Q}$.

Example 3. *Given two query groups QG_1 and QG_2 in Fig. 3, on stream in Fig. 4 we demonstrate how K-SKY supports outlier detection queries with varying k and r parameters. As shown in Fig. 4, p_8 is processed first and hashed to bucket B_2 of LSky. p_7 is processed next and inserted into bucket B_3. p_7 is dominated by one point in B_2. In other words, p_7 is dominated by $k_1 - 1$ points, where k_1 is the k parameter of QG_1. By the inlier rule this may cause p to be recognized as inlier for some queries in QG_1. By comparing $dist(p, p_7)$ against the r parameters in QG_1, p is confirmed to be inlier for queries $< k_1, r_3 >$ and $< k_1, r_4 >$, since $dist(p, p_7) \leq r_3 < r_4$. Then K-SKY proceeds to hash p_6 into bucket B_4. p_6 is dominated by two points p_7 and p_8. This triggers the inlier status check for queries in QG_2 (k_2 = 3). Since $dist(p, p_6) \leq r_4$, p is inlier for query $< k_2, r_4 >$. Next p_5 is processed and inserted into bucket B_1. When K-SKY processes p_4, p_4 is dominated by 2 (k_2-1) points. Furthermore, $dist(p, p_4) \leq r_{min}$ of QG_2 (r_2 = 2). By the termination condition of K-SKY, QG_2 (k_2 = 3) is terminated now, since all queries in QG_2 classify p as inlier. Next p_3 will be excluded from LSky, since p_3 (in B_3) is dominated by four points and therefore is not a skyband point for any query group. After p_2 is evaluated, there are two points p_2 and p_5 in B_1 whose distance to p is not larger than r_{min} = r_1 = 1 of QG_1 (k_1 = 2). This satisfies the termination condition of QG_1. In turn p is classified as inlier by all queries in QG_1. This leads to the termination of the outlier status evaluation process for point p, because both query groups have been completed. The earliest arrival p_1 is not evaluated.*

	r / k	r_1= 1	r_2 = 2	r_3=3	r_4=4
QG1	k_1 = 2	X		X	X
QG2	k_2 = 3		X	X	X

Figure 3: Queries with varying *k* and *r* parameters

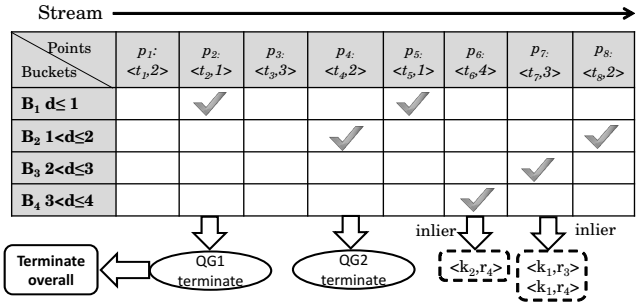


Figure 4: K-SKY for multiple queries

The Overall Outlier Detection Approach. The overall process of utilizing the K-SKY algorithm to continuously detect outliers from the sliding window stream is shown in Alg. 3.

Algorithm 3 detectOutlier($W_c.plist$,$\mathbb{Q}$)

```
Output: outliers; //the outlier sets
1:  for each p ∈ W_c.plist do
2:      if (IsSafeInlier(p) == TRUE) then
3:          break;
4:      else
5:          K-SKY(p, W_c.plist, p.skyband, ℚ);
6:          ℚ_in = markInlierStatus(p, ℚ);
7:          markSafeInlier(p, ℚ);
8:          if ℚ - ℚ_in ≠ ∅ then
9:              insertOutlier(outliers,p,ℚ - ℚ_in);
10:         end if
11:     end if
12: end for
```

Given a point *p* in the current window W_c, Alg. 3 first checks whether p is a safe inlier (Line 2). The K-SKY algorithm will only execute on the points that are not marked as safe inliers (Lines 4, 5). Then based on the output of K-SKY, we mark *p* as inlier for the corresponding queries $\mathbb{Q}_{in}$ (Line 6). The safe inlier status of *p* will also be updated (Line 7). Finally *p* is inserted into the output set *outliers* if not all queries in $\mathbb{Q}$ classify *p* as inlier (Lines 8 - 10). Each element in the *outlier set* records one point *p* along with the member queries $q_i \in \mathbb{Q}$ - $\mathbb{Q}_{in}$ that classify *p* as outlier.

4. VARYING SLIDING WINDOW PARAMETERS

Next, we study the case that the window parameters can vary. The key observation here is that such multiple queries can be supported by utilizing one single customized skyband query. Therefore full sharing of both CPU and memory resources is achieved over sliding windows.

4.1 Varying the Window Parameter - Win

Here we first examine the scenario when the window sizes vary, while the slide size remains stable. Therefore all queries slide to a new window at the same time. In other words, they are *synchronized*. All queries require output at exactly the same moment, i.e., at time $W_c.end$ in Fig. 5. This observation leads to an important characteristic. Given a query group $\mathbb{Q}$ with member queries having the same slide size but arbitrary window sizes, $\mathbb{Q}$ can be supported with one skyband query with respect to q_{max} denoted as q^s_{max}, namely the member query with largest window in $\mathbb{Q}$ as in Fig. 5. Intuitively this is so because the largest window covers all smaller windows. Therefore skyband points discovered in the largest window can be utilized to answer all queries in the group.

Therefore by employing the K-SKY algorithm and collecting the skyband points for this special skyband query q^s_{max}, this outlier query group $\mathbb{Q}$ can be correctly answered with each point *p* in the data stream *S* evaluated only once in each window that contains *p*.

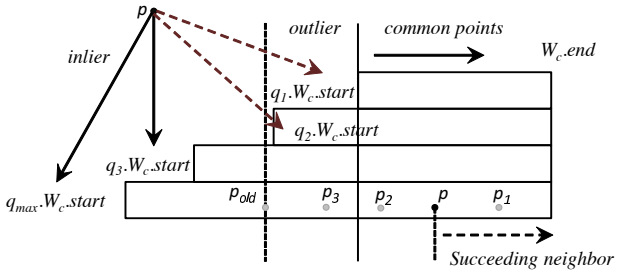


Figure 5: Queries with varying window sides

As shown in Sec. 3.1.2, K-SKY gives more preference to the points arriving later than the points arriving earlier. That is, K-SKY always processes the later arrivals first. On the other hand the later arrivals in the stream happen to be the common points among the data populations covered by the current windows of different queries (Fig. 5). Therefore K-SKY naturally leverages the data commonality among the windows of distinct queries. Redundant computations during the skyband point discovery process are eliminated.

Outlier Status Evaluation. After K-SKY terminates, as shown in Sec. 3.1.2, the outlier status of $p \in q_{max}.D_{W_c}$ with respect to q_{max} has already been determined. If q_{max} marks *p* as outlier, then *p* is guaranteed to be an outlier for any other query q_i in Q. Therefore it is not necessary to evaluate the outlier status of q_i anymore. However this is not the case if p instead is marked as an inlier for q_{max}. This is so because the neighbors of a particular q_i are only a subset of the neighbor set of q_{max}. Even if q_{max} has acquired enough neighbors to prove the inlier status of *p*, this does not guarantee that *p* has enough neighbors for other queries with smaller windows. Therefore an *extra outlier status evaluation step* is necessary for other member queries q_i in $\mathbb{Q}$ besides q_{max}. Fortunately, as we demonstrate below, to determine the outlier status of *p* with respect to the member queries in $\mathbb{Q}$, it is not necessary to examine all points in $q^s_{max}.skyband$.

Lemma 3. *Given a query group* $\mathbb{Q}$: $\{q_1, q_2, ..., q_m, q_{m+1}, ..., q_{max}\}$ *(*$q_1.win < q_2.win < ... < q_{max}.win$*), p is an outlier for queries* $\{q_1, q_2, ..., q_m\}$*, if* $q_{m+1}.W_c.start < p_{old}.time < q_m.W_c.start$*, where* p_{old} *is the oldest point in* $q^s_{max}.skyband$.

The formal proof of Lemma 3 can be found in Appendix C.

By Lemma 3 when *p* is an inlier of q_{max}, to determine the outlier status of *p* with respect to all other queries in $\mathbb{Q}$ we only need to evaluate one single skyband point p_{old}, namely the last skyband point acquired exactly when K-SKY terminates. This is achieved by locating the query q_m with largest window size whose current window W_c has not started yet at the time when p_{old} arrives. As shown in Fig. 5, queries q_1 and q_2 are outliers, because the oldest neighbor of *p*, namely p_{old} arrived earlier than the start time of the windows of q_1 and q_2.

Let us assume the queries in $\mathbb{Q}$ are ordered by their window sizes. In that case, the delimiter query q_m can be located in $O(\log \mid \mathbb{Q} \mid)$ time with a binary search style algorithm.

Safe Inlier. A point *p* declared as inlier by query q_{max} is not necessarily an inlier for all queries in $\mathbb{Q}$. However this is not the case when *p* is a *safe inlier*. Once *p* is confirmed as a safe inlier by q_{max}, *p* is guaranteed to be a safe inlier for all queries in $\mathbb{Q}$.

Since all queries in $\mathbb{Q}$ have the same slide size, their current window W_c ends at the same time point. Therefore all queries in $\mathbb{Q}$ share the same succeeding points of *p* in W_c, namely the points arriving later than *p*, but earlier than the end of the window as depicted in Fig. 5. This indicates that the *k* succeeding neighbors of *p* discovered by q_{max} are shared by all queries. Hence *p* is a safe inlier with respect to all queries in $\mathbb{Q}$.

Based on this **Safe-For-All** property, once *p* is determined by K-SKY to be a safe inlier of q_{max}, *p* can also be declared to be a safe inlier for all queries without requiring any further evaluation. It is then safe to exclude *p* from further evaluation in any future window. Therefore significant CPU and memory resources are saved.

In summary, we conclude that we only need to detect and maintain the skyband points for one single skyband query with respect to the outlier query with the largest window size. This then is sufficient to answer all outlier queries in the query group. Clearly, full sharing is achieved.

4.2 Varying the Slide Parameter

Next, we consider the case where all queries have the same window size, while their slide sizes vary. Unlike the previous varying window size case, these queries are not synchronized. That is, their windows move at a different pace. Therefore no stable relationship holds across the data populations covered by the active windows with respect to different queries. In other words there is no such query whose active window continuously contains the windows of other queries. Therefore the above strategy supporting queries with various window sizes does not handle this case. However independently generating output for this set of queries at different moments is not practical for large volume streams.

To solve this problem, given a query group $\mathbb{Q}$, we build a single *swift query* q_{sft} that correctly answers all member queries of $\mathbb{Q}$. Q_{sft} has the same window size as all member queries in Q, while its slide size is set as the greatest common divisor on the slide sizes of all the queries in $\mathbb{Q}$.

Intuitively by definition of the greatest common divisor, $\forall q_i \in \mathbb{Q}$, we have $q_i.slide \bmod q_{sft} = 0$. Therefore at any time t_j when $q_i \in \mathbb{Q}$ produces an outlier result $q_i.outlier$, q_{sft} would also be producing result $q_{sft}.outlier$. Furthermore, since $q_i.win = q_{sft}.win$, the points covered by the window of q_i and q_{sft} would be identical at t_j. Therefore at any t_j $q_i.outlier = q_{sft}.outlier$. Hence Q_{sft} is sufficient to represent all queries in $\mathbb{Q}$.

Therefore a query group $\mathbb{Q}$ with varying slide sides can be supported by one special skyband query with respect to this special outlier query q_{sft}. It is straightforward to determine at runtime when to output the outlier detection results and what query the out-

put corresponds to by tracking for each query when the window slides.

Safe Inlier. Although potentially q_{sft} slides its window more frequently than any q_i in $\mathbb{Q}$, this swift query solution does wastes neither CPU nor memory resources. This is because q_{sft} is able to discover *safe inliers* and to terminate the outlier detection process at the earliest possible moment. This observation relies on the **Safe For All** property of q_{sft} similar to q_{max} in the varying window size case. Namely given a point *p*, if *p* is recognized as a safe inlier for the swift query q_{sft}, then *p* is a safe inlier for all $q_i \in \mathbb{Q}$. Next we briefly justify this observation.

We use $succ(p, q)$ to denotes the points arriving later than *p* in the current window W_c of query q. The **safe-for-all** property follows immediately from the fact that in any future window $succ(p, q_{sft})$ is a subset of $succ(p, q_i)$ for any query $q_i \in \mathbb{Q}$. This is so because in any future window when query q_i is scheduled to produce outlier status for *p*, all points succeeding to *p* in the current window of q_{sft} will not expire (since *p* has not expired). Furthermore, q_i will also get some additional new points into the future window.

Since q_{sft} is potentially scheduled more frequently than any query in $\mathbb{Q}$, the safe inliers will be discovered quicker. q_{sft} is able to discover and prune the safe inliers earlier than any query in $\mathbb{Q}$. Therefore CPU and memory resources are saved.

In conclusion, this swift query solution achieves full sharing by utilizing only one single skyband query to answer all member queries in $\mathbb{Q}$. Furthermore safe inliers are also discovered and discarded earlier than any actual member query, leading to additional saving in CPU and memory resources.

4.3 Varying Both Win and Slide Parameters

We now describe our solution for the case when both window parameters, namely win and slide, vary. This solution is a straightforward combination of the techniques introduced in the last two sections. In particular, we simply build one *single swift query* that has the largest window size among all member queries and its slide size as the greatest common divisor of the slide sizes of all member queries. A specific skyband query with respect to this single swift query will then be employed to collect skyband points, namely the evidence to prove the outlier status of a given point *p*. Similar to the varying slide size case the timing when each query is required to produce output is determined at runtime (see Sec. 4.2). Then Lemma 3 introduced in Sec. 4.1 is applied to decide the outlier status of each point for the queries requiring output based on the results of the swift skyband query. In short, this case of arbitrary window and slide sizes can be regarded as an arbitrary window size case with a fixed slide size whose value is the greatest common divisor of the slide sizes of all member queries.

5. VARYING ALL PARAMETER SETTINGS

Finally, we consider the most general case with arbitrary pattern and window-specific parameters. Although sharing among a group of totally arbitrary queries appears hard at first sight, we now demonstrate that this problem can be tackled utilizing the skyband query technique. This is possible, because as shown in Sec. 4.3, the skyband query technique designed for processing the outlier analytics workloads with varying *pattern-specific* parameters can be leveraged to answer multiple queries with arbitrary *window-specific* parameters.

SOP Outlier Detection Framework. As depicted in Fig. 6, SOP first employs a query parser to divide the queries in a query group $\mathbb{Q}$ into sub-groups $\mathbb{Q}_i$ based on their k parameters. Queries with the same k parameter are grouped into one sub-group $\mathbb{Q}_i$. The queries in each sub-group $\mathbb{Q}_i$ are then sorted based on their r parameters. The queries with same r parameters are further sorted based on their window sizes. Then the query parser will create one skyband query Q_i^s for each outlier query sub-group $\mathbb{Q}_i$. Its window size is set as the the largest window size among the member queries in $\mathbb{Q}_i$. Its slide size is then set as the greatest common divisor of the slide sizes of the member queries.

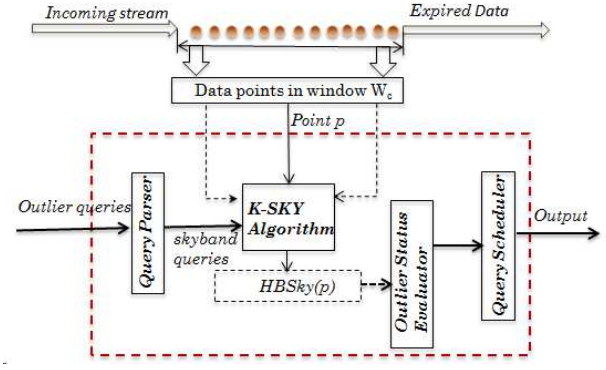


Figure 6: SOP Framework

After the query parser transforms the outlier detection queries into the skyband queries, the K-SKY algorithm for multiple skyband queries introduced in Sec. 3.2 will be applied to detect the skyband points. Then the outlier status evaluator determines the outlier status of each data point with respect to the outlier queries using the inlier rule introduced in Sec. 3.2.2.

Similar to the varying window sizes case if a data point *p* is classified as an outlier for the queries in some sub-group $\mathbb{Q}_i$, then p is guaranteed to be an outlier for all queries in $\mathbb{Q}_i$ no matter what their window sizes are. On the other hand if *p* is declared to be an inlier for some queries, Lemma 3 has to be applied to evaluate whether *p* is indeed an inlier for these queries by checking their window sizes as shown in Sec. 4.1.

Once the outlier status of *p* is determined for certain queries, the query scheduler determines whether it is time to output the outliers for these queries based on their slide sizes (per Sec. 4.2).

Conclusion. Computation-wise, SOP only requires a single pass through new data points, each collecting the minimum evidence to prove its outlier status with respect to all queries. Memory-wise, the evidence which proves the outlier status of each data point with respect to multiple queries is maintained only once. In short, SOP achieves full sharing for multiple distance-based outlier queries over sliding windows in terms of both CPU and memory resources.

6. PERFORMANCE EVALUATION

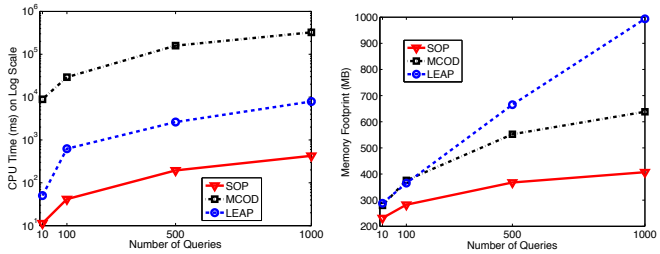
6.1 Experimental Methodology

We conducted experiments on a PC with 3.4G HZ Intel i7 processor and 6GB memory, running Windows 7 OS. All algorithms are implemented in JAVA on HP CHAOS stream engine [8].

Real Data Sources. We use the Stock Trading Traces Data (STT) [11]. It has one million transaction records throughout the trading hours of one day. All data has the same format of name, transId, time, volume, price, and type.

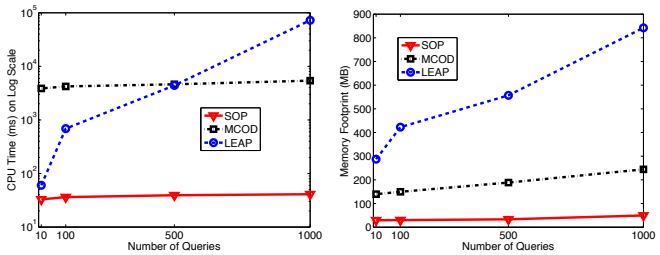
Synthetic Data. We also implement a data generator to create a dataset containing 100M points. This dataset is composed of Gaussian distributed data points as inlier candidates and uniform distributed ones as outliers. The outliers are randomly distributed in each time segment of the data stream.

Alternative Algorithms. We compare our proposed SOP algorithm with the two state-of-the-art solutions from the literature [13, 7]. Since MCOD [13] does not support variations in window-specific parameters, we have extended MCOD by inserting our



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

Figure 7: Varying r values for queries on synthetic dataset



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

Figure 8: Varying k values for queries on synthetic dataset

window-specific techniques into MCOD. We now use this enhanced algorithm to compare against SOP. In addition, we also compare our sharing strategy against the state-of-the-art single query strategy LEAP [7]. Multiple queries are supported by applying LEAP independently to process each query in the query group.

Metrics. We measure two metrics common for stream systems, namely the average processing time (CPU time) per window and the peak memory consumption (MEM). The CPU time per window corresponds to the total amount of system time resources used to process the queries on the data in one window. The consumed memory metric corresponds to the memory required to store the information for each active object (i.e. the skyband points) and the outliers of all queries in the current window. All results are collected and calculated at the unit of one window at a time. Then they are averaged over all windows processed in the given experiments. All experiments are reported using the count-based window, with time-based window processing achieving similar results.

We also conduct scalability tests to validate the performance of the proposed algorithms with an increasing number of queries in the workload.

All in all our study covers important combinations of the four query parameters. They range from varying one specific parameter only at a time to the more general cases of varying all 4 of them among the queries populating the workload as shown in Table 1. The varying ranges of the parameters are listed in Table 2.

Workload	Pattern		Window	
	R	K	W	S
(A)	arbitrary	fixed	fixed	fixed
(B)	fixed	arbitrary	fixed	fixed
(C)	arbitrary	arbitrary	fixed	fixed
(D)	fixed	fixed	arbitrary	fixed
(E)	fixed	fixed	fixed	arbitrary
(F)	fixed	fixed	arbitrary	arbitrary
(G)	arbitrary	arbitrary	arbitrary	arbitrary

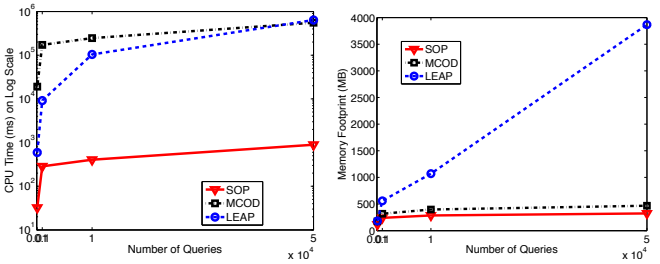
Table 1: Combinations of different workloads

Type	Name	Value
Pattern	K	[30,1500)
	R	[200,2000)
Window	W	[1Ks,500Ks)
	S	[50s,50Ks)

Table 2: The ranges of the parameters

6.2 Varying Pattern Specific Parameters

Our general methodology is to prepare four workloads with 10, 100, 500, 1000 queries respectively by randomly choosing the values of the pattern-specific parameters in a range for each query, while fixing the window-specific parameters. The synthetic dataset



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

Figure 13: Varying K, R, W and S values on synthetic dataset

is utilized in this set of experiments to make sure the outlier rate is small ($< 5\%$) when varying the k and r parameters.

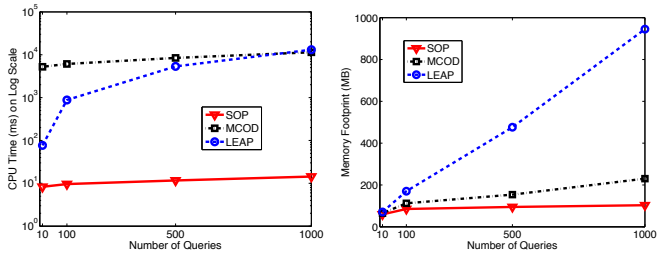
Arbitrary R Case. In the first experiment, we evaluate the performance of our SOP compared with the state-of-the-art MCOD [13] and LEAP [7] when only varying parameter r. We fix the window size to 10K, slide size to 0.5K and k parameter value to 30, while r is randomly selected in the range from 200 to 2000.

As shown in Fig. 7(a), SOP significantly outperforms MCOD and LEAP up to 3 orders of magnitude in CPU time. By mapping the multiple outlier query problem to the skyband query problem, SOP only needs to collect *minimum information* to prove the outlier status of each data point with respect to *all* queries. Instead, MCOD relies on routinely conducting a range query to detect outliers. That is, in each case it will compare each data point with all the other data points in each window and collect all the points satisfying the neighbor condition of any user query. On the other hand LEAP repeatedly detects outliers for each query from scratch. Its CPU performance thus degrades quickly as the number of queries increases. We thus confirm that the CPU efficiency of both MCOD and LEAP is significantly worse than that of SOP.

SOP is also superior in memory usage as shown in Fig. 7(b). This is because in each window given a data point p, MCOD keeps all data points satisfying the neighbor conditions of p with respect to any query. SOP instead determines that it is not necessary for the evaluation of the outlier status of p. On the other hand, LEAP, without leveraging the sharing opportunities across multiple queries, maintains the neighbors of each point independently for each query.

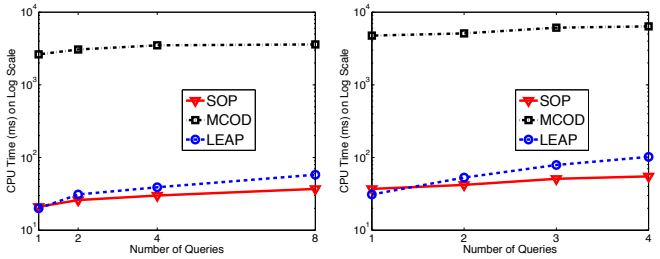
Arbitrary K Case. In this experiment, we analyze the performance of SOP by varying k parameter values of the queries. We use a fixed window size of 10K and a slide size of 0.5K. Parameter r is fixed at 700. A value for k is randomly selected in the range from 30 to 1500 for each query.

As shown in Fig. 8(a), similar to the case of varying r, the CPU performance of SOP outperforms the other two alternatives up to 4 orders of magnitude. Since the case of varying k can be treated as a special case of the case with arbitrary k and r values, this experiment demonstrates the effectiveness of our K-Sky algorithm when handling multiple skyband queries.



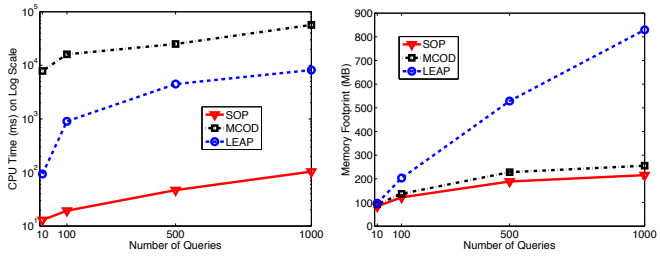
(a) CPU (log scale) (b) Memory (log scale)

Figure 9: Varying k and r values on synthetic dataset



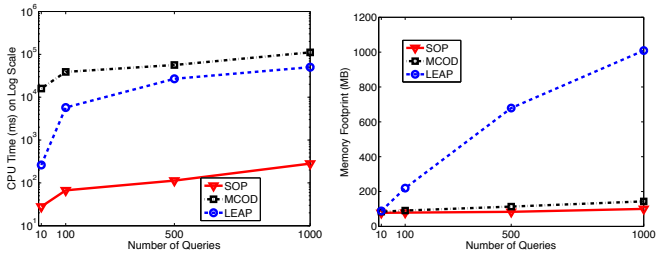
(a) Same attribute (b) Different attributes

Figure 10: CPU (log scale): small workload



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

Figure 11: Varying Win for queries on STT dataset



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

Figure 12: Varying W and S values on STT dataset

The CPU resources utilized by SOP are very stable as the number of queries increases. This is because for each workload the k value is randomly selected in the same range. In each workload at least one of the randomly selected k is likely to get fairly close to the upper ceiling value in the range. In other words the maximum k in each workload is similar on average. Therefore this experiment demonstrates that the performance of SOP relies on the largest k value instead of on the number of queries in the workload. Therefore SOP scales to a potentially huge workload composed of a large number of queries. A similar trend can also be observed in memory utilization as shown in Fig. 8(b).

Arbitrary K and R Case. In this experiment, we assess the performance of the algorithms when varying both k and r. We fix the window size to 10K, slide size to 0.5K, while the values for both k and r are randomly generated in the range respectively from 30 to 1500 and from 200 to 2000 for each query.

Fig. 9 depicts the performance of the three algorithms in terms of CPU costs and memory consumption. We observe that SOP consistently outperforms MCOD and LEAP up to 3 orders of magnitude. This confirms that K-SKY not only effectively shares the computation among the queries with an identical k parameter, but it also achieves full sharing across multiple skyband queries with respect to different query groups with distinct k values. MCOD instead solves this case by *simulating* an outlier query using the largest k and smallest r values in the workload as its pattern parameters. Such a query can have much more restricted neighbor requirements and in turn more expensive range queries than any of the actual outlier queries. Huge CPU and memory resources may be wasted compared to SOP.

Small Workload. In this set of experiments, we test the performance of SOP when processing small workload. It is composed of two experiments. In the first experiment, all queries utilize the same set of attributes in the detection of outliers. We vary the size of the workload by containing 1, 2, 4, 8 queries. Again it confirms that SOP performs well even in this small workload case as shown in Fig. 10(a). In particular when the workload contains only one single query, SOP does not perform worse than the state-of-the-art single query approach LEAP. This shows that no much extra overhead is introduced by SOP.

Furthermore, we also evaluate the performance of SOP when handling queries utilizing different set of attributes. In this experiment, the queries are divided into 3 groups. The queries in the same group utilize the same set of attributes. We vary the number of queries in each group from 1 to 4. To support such workload SOP is slightly extended using a simple divide and conquer approach. As depicted in Fig. 10(b) our extended SOP approach continues to perform well. More specifically SOP is at least 150 times faster than MCOD and two times faster than LEAP even in this small number of queries case.

6.3 Varying Window Specific Parameters

Next, we focus on workloads composed of 10, 100, 500, 1000 queries respectively for the case when varying window-specific parameters, while using a fixed pattern-specific parameter setting. In this set of experiments the stock data [11] is utilized to evaluate how our SOP solution performs when handling real datasets.

Arbitrary Win Case. In this experiment, we study the performance of SOP for window sizes ranging from 1K to 500K. We fix the slide size as 0.5K, r as 200, and k as 30.

As shown in Fig. 11, SOP features significantly better performance on both CPU and memory consumption compared to MCOD and LEAP. Since MCOD leverages the sharing opportunities across the windows of multiple queries by adopting our swift query strategy, its CPU and memory usage is relatively stable compared to LEAP as the number of queries increases. However MCOD is still outperformed by SOP by at least 2 orders of magnitude in CPU time as shown in Fig. 11(a). This is because based on our *safe-for-all* observation in Sec. 4.1, SOP terminates and excludes p from any future evaluation process immediately once p is classified as safe inlier by the skyband query corresponding to the outlier query with the largest window size. As stated earlier, MCOD instead relies on a range query to detect these outliers. Even if a data point is recognized as safe inlier, this neighbor search continues completing the comparisons with all other untouched data points. Therefore a huge amount of CPU resources can be wasted.

Arbitrary Win and Slide Case. In this experiment, we investigate the performance of SOP when varying both window-specific parameters. We fix k as 30 and r as 200. The window and slide sizes are arbitrarily selected for each query from the range of 1K to 500K and from 50 to 50K respectively.

As illustrated in Fig. 12, the average CPU time consumed by SOP increases only from 28ms to 282ms (10 folds) as the number of queries increases from 10 to 1000 (100 folds). This continues to outperform the alternative algorithms by at least two orders of magnitudes. Clearly, results shown in Fig. 12 demonstrate the effectiveness of the swift query strategy for handling arbitrary win and arbitrary slide case.

6.4 Varying Pattern and Window Parameters

In this most general case, we prepare four workloads composed of 100, 1000, 10,000, 50,000 queries respectively by varying all window-specific and pattern-specific parameters.

We observe from Fig. 13(a) that similar to the cases of independently varying pattern-specific parameters and window-specific parameters SOP achieves tremendous gain in CPU utilization compared to (augmented) MCOD and (the non-shared) LEAP. Furthermore SOP shows excellent scalability in the cardinality of the workload. As the number of queries rises from 1000 to the extremely large cardinality of 50,000 queries, the CPU costs of SOP only increase from 32ms to 892ms. As the number of the queries increases, the sharing opportunities among the given set of queries also increase. Since SOP achieves full sharing across queries, it effectively reduces the CPU burden caused by the huge workload.

The memory usage of SOP also consistently outperforms the alternatives solutions as shown in Fig. 13(b). As previously stated, the reason is that LEAP detects outliers on the same streaming data for each query independently. Hence the memory consumed by the workload queries accumulates as the number of queries grows. On the other hand, MCOD always detects outliers by discovering and maintaining all neighbors for each point. However, SOP only requires minimal information to prove the outlier status of each point. As a result, SOP effectively avoids the usage of unnecessary space by purging redundant intermediate results. Therefore significant memory utilization is reduced by SOP.

7. RELATED WORK

Distance-based Outliers on Streaming Data. With the emergence of digital devices generating data streams, outliers on streaming data are one type of anomalies recently studied [7, 13, 1]. Existing work [7, 13, 1] focuses primarily on processing a single outlier detection request. In particular [1] leverages the observation that the neighbors p_i of a point p that arrived after p do not expire before p expires. They make a distinction between the preceding neighbors of p, P_p, i.e., those that will expire before p, and the succeeding neighbors of p, S_p, those that will persist during the entire lifetime of p. They first introduce the idea of a "safe inlier" as a point p with $\geq$ k succeeding neighbors.

[13] further improves on [1] by leveraging the *safe inlier* concept of [1]. That is, it organizes the data points into a queue based on the number of their succeeding neighbors, so that it can efficiently schedule the necessary checks that have to be made when the window slides. However it still relies on full range query searches to process the newly arriving points. Therefore it cannot provide real time responsiveness when applied to high velocity streaming data. Besides this single query technique, [13] also touches on supporting multiple outlier detection queries for the case of varying pattern-specific parameters. Given a data point p they first utilize a range query to find all points that satisfy the neighbor condition of all queries in the query group. Then a postprocessing step is applied to filter the unnecessary points from this large neighbor set to reduce the maintenance costs. In our work by directly transforming the multi-query outlier problem into the single query skyband problem, we only collect the necessary evidence that is sufficient to answer multiple outlier queries, therefore significantly outperforming this method as confirmed in our experiments.

Due to a deeper understanding the temporal relationships among stream data points, [7] overcomes the limitation of prior methods [1, 13] of undertaking full range query searches by discovering as early as possible safe inliers in the scan process. This allows it to better satisfy the performance requirements of modern streaming applications. However multiple outlier detection requests are not supported in [7].

Outliers on Sensor Streams. [20] proposes an online technique to detect outliers in sensor stream data. This work primarily focuses on how to reduce the message transmission and in turn reduce the power consumption of the sensors. First, it utilizes kernel density estimation to model the distribution of the sensor data. Then given a point p, [20] approximates whether p is an outlier by estimating the number of its neighbors using the density distribution function $f(p)$. However unlike our work focusing on detecting *exact* distance-based outliers, this *approximation* no longer assures complete results. Furthermore, they only support a single outlier detection query. No multi-query technique is discussed.

kNN Queries on Streams. Continuous kNN queries over sliding windows have indeed been studied in [15, 6]. Both works use a grid to index the stream data. To improve response time, they either postpone the processing of the new points which are not likely to be in kNN set [6] or eagerly pre-compute the possible kNN set for each future window as new data arrives [15].

However to determine the outlier status of p it is not necessary to always discover the full kNN of p. Rather one algorithm should stop evaluating p as long as any k neighbors of p have been discovered. Therefore the use of the streaming kNN algorithm to continuously detect outliers is not an efficient approach. Instead our customized skyband algorithm K-SKY always discovers the minimal information necessary to determine the outlier status of p.

General Multi-Query Optimization. Multiple query sharing has been widely studied as a general optimization problem in streaming environments. Previous research on sharing computations studied traditional SQL queries such as selection, join, and aggregation [4, 9, 22, 14]. Their methods include rewriting queries to expose common subexpressions, sharing indices, or segmenting input into partitions and sharing partial results over the partitions. However the key problem we address in this work, namely correctly answering multiple outlier detection queries by only collecting minimal information, is different from the more general purpose optimization effort required by the traditional SQL query sharing.

8. CONCLUSION

In this work, we present the first solution, called SOP, for efficient shared processing of a large number of distance-based outlier detection requests with diverse parameter instantiations over sliding window streams. SOP requires only one single pass over the data points to support a huge workload composed of a large number of outlier detection requests with arbitrary input settings for all pattern and window specific parameters. Our experimental study using both real and synthetic streaming datasets confirms that for the rich diversity of tested scenarios SOP outperforms the alternatives on average three orders of magnitude in CPU time, while using only 5 % memory space of its counterparts.

9. REFERENCES

[1] F. Angiulli and F. Fassetti. Distance-based outlier queries in data streams: the novel task and algorithms. *Data Min. Knowl. Discov.*, 20(2):290–324, 2010.

[2] F. Angiulli and C. Pizzuti. Fast outlier detection in high dimensional spaces. In *PKDD*, pages 15–26, 2002.

[3] A. Arasu, S. Babu, and J. Widom. The cql continuous query language. *VLDB J.*, 15(2):121–142, 2006.

[4] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, pages 336–347, 2004.

[5] 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.

[6] C. Böhm, B. C. Ooi, C. Plant, and Y. Yan. Efficiently processing continuous k-nn queries on data streams. In *ICDE*, pages 156–165, 2007.

[7] L. Cao, D. Yang, Q. Wang, Y. Yu, J. Wang, and E. A. Rundensteiner. Scalable distance-based outlier detection over high-volume data streams. In *ICDE*, pages 76–87, 2014.

[8] C. Gupta, S. Wang, I. Ari, M. C. Hao, U. Dayal, A. Mehta, M. Marwah, and R. K. Sharma. Chaos: A data stream analysis architecture for enterprise applications. In *CEC*, pages 33–40, 2009.

[9] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *VLDB*, pages 297–308, 2003.

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

[11] I. INETATS. Stock trade traces. *http://www.inetats.com/*.

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

[13] 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.

[14] S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD Conference*, pages 623–634, 2006.

[15] K. Mouratidis and D. Papadias. Continuous nearest neighbor queries over sliding windows. *IEEE Trans. Knowl. Data Eng.*, 19(6):789–803, 2007.

[16] A. Nazaruk and M. Rauchman. Big data in capital markets. In *SIGMOD Conference*, pages 917–918, 2013.

[17] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1):41–82, 2005.

[18] S. Ramaswamy, R. Rastogi, and K. Shim. Efficient algorithms for mining outliers from large data sets. In *SIGMOD Conference*, pages 427–438, 2000.

[19] G. E. Rosario, E. A. Rundensteiner, D. C. Brown, and M. O. Ward. Mapping nominal values to numbers for effective visualization. In *InfoVis*, pages 113–120, 2003.

[20] S. Subramaniam, T. Palpanas, D. Papadopoulos, V. Kalogeraki, and D. Gunopulos. Online outlier detection in sensor data using non-parametric models. In *VLDB*, pages 187–198, 2006.

[21] Y. Tao and D. Papadias. Maintaining sliding window skylines on data streams. *IEEE Trans. Knowl. Data Eng.*, 18(2):377–391, 2006.

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

APPENDIX

A. PROOF OF PROBLEM MAPPING

Lemma 1. *Given a query group $\mathbb{Q}$, for any data point p, the output of Q^s, denoted as $\mathbb{S}_p$, is **sufficient** and **necessary** to continuously determine the outlier status of p with respect to all queries in $\mathbb{Q}$, where Q^s represents the $(k-1)$-skyband query with respect to p with k as the neighbor threshold parameter defined in $\mathbb{Q}$.*

Proof. "⇒" we prove the sufficiency by showing that Q^s always returns the *k nearest neighbors* of *p* kNN(p) [3] as part of the skyband points. Assume the data points in D have been partitioned into t subsets S_1, S_2, ..., S_t. Each subset S_i contains all data points with the same distance to *p*. Assume here that the subsets S_1, S_2 ,..., S_t are sorted in an ascending order by their distance to *p*. Assume $\mid S_1 \mid + \mid S_2 \mid +, ..., + \mid S_x \mid < k$ and $\mid S_1 \mid + \mid S_2 \mid + ...+ \mid S_x \mid + \mid S_{x+1} \mid \geq k$. All data points in $S_1 \cup S_2 \cup ... \cup S_x$ along with (k - ($\mid S_1 \mid + \mid S_2 \mid +, ..., + \mid S_x \mid$)) data points with largest timestamp in S_{x+1} will be returned by Q^s, since by the domination rule in Def. 5 they are dominated by at most k -1 data points. These k data points satisfy the $kNN(p)$ definition, since no other data point is closer towards *p* than they are. Then by the *k-distance* observation, kNN(p) is sufficient to prove the outlier status of *p* with respect to $\mathbb{Q}$. The sufficiency is proven.

"⇐" By contradiction. Given a point p_i ($p_i \in \mathbb{S}_p$ - kNN(p)), although by the *k-distance* observation, $\mathbb{S}_p$ - $\{p_i\}$ is still sufficient to determine the status of *p* in the current window W_c, potentially *p* may be erroneously evaluated in the future window W_{c+x}. Assume that when the stream evolves to w_{c+x}, all points in $\mathbb{S}_r$ arriving earlier than p_i have expired. If all points p_j arriving after W_c are far from *p* ($dist(p, p_j) > r_i$), p_i should now be in kNN(p). Since the outlier status of *p* is determined by kNN(p), to correctly determine the outlier status of *p* in any future window, p_i has to be kept. The necessity is proven. ■

B. OPTIMALITY PROOF OF K-SKY

Lemma 2. *K-SKY correctly discovers the (k-1)-skyband points in window W_c by examining only the minimum number of data points.*

Proof. We prove Lemma 2 by showing that: (1) Any point inserted to skybandPoints during the execution of K-SKY is guaranteed to be a true *(k-1)-skyband* point; (2)No data point that could not be a *(k-1)-skyband* point is examined during the execution of K-SKY.

Proof of (1). In K-SKY the later arrivals are always evaluated earlier than the earlier arrivals. Therefore any point p_i already added into *skybandPoints* has a larger timestamp than any point p_j remaining to be evaluated. That is, $p_j.time < p_i.time$. By the domination rule defined in Def. 5, p_j cannot *dominate* p_i. Therefore p_i would not be replaced by any point evaluated later. Condition (1) holds.

Proof of (2). Proof in two steps. First, K-SKY stops immediately once the termination condition is satisfied. Namely K-SKY terminates immediately once one point p_i is dominated by k points if the distance between p_i and p is not larger than the smallest r parameter r_{min} in $\mathbb{Q}$. Therefore the remaining points that will not be in $skybandPoints$ are not evaluated.

Second, we prove any data point evaluated during the execution of K-SKY is potentially a *(k-1)-skyband* point. Data point p_i is evaluated by K-SKY if and only if the termination condition has

[3]This kNN is based on the normalized distance function.

not yet been satisfied. In other words when p_i was evaluated, at most k - 1 data points p_j with dist(p,p_j) $\leq r_{min}$ existed at that time. Therefore p_i should be listed in $skybandPoints$ if dist(p,p_i) $\leq r_{min}$. That is, if we were not to consider p_i, then potentially an incorrect skyband point set may be reported. Furthermore, in K-SKY if p $\in W_c$ is an point that survived the stream data expiration, only the new arrivals in W_c and its unexpired unexpired skyband points in last window W_{c-1} will be examined. By our *least examination optimization* principle these points are the only points that will appear in the skyband point set of the new window W_c. This confirms that any point evaluated by K-SKY is indeed necessary to guarantee the correctness of the Q^s query. ■

C. PROOF OF LEMMA 3

Lemma 3. *Given a query group* $\mathbb{Q}$*:* $\{q_1, q_2, ..., q_m, q_{m+1}, ..., q_{max}\}$ *(*$q_1.win < q_2.win, < ..., < q_{max}.win$*), p is an outlier for queries* $\{q_1, q_2, ..., q_m\}$*, if* $q_{m+1}.W_c.start < p_{old}.time < q_m.W_c.start$*, where* p_{old} *is the oldest point in* $q^s_{max}.skyband$*.*

Proof. q^s_{max} is a special skyband query with respect to the single outlier query $q_{max}(k, r)$. Since *p* is an inlier with respect to q_{max}, the skyband set $q^s_{max}.skyband$ contains *k* neighbors of *p*. p_{old} is dominated by k -1 points in $q^s_{max}.skyband$, because $p_{old}.time < p_i.time$ ($\forall p_i \in q^s_{max}.skyband$). Since $q_{m+1}.W_c.start < p_{old}.time < q_m.W_c.start$, p_{old} falls in the current window W_c of $\{q_{m+1}, ..., q_{max}\}$, but is out of the W_c of $\{q_1, q_2, ..., q_m\}$. Therefore *p* has at most k - 1 neighbors in $q^s_{max}.skyband$ for $\{q_1, q_2, ..., q_m\}$. Furthermore, any point p_j out of $q^s_{max}.skyband$ cannot be a neighbor of *p* for $\{q_1, q_2, ..., q_m\}$ in their current window W_c. Otherwise p_{old} would also be dominated by p_j. Hence it would in total be dominated by *k* points. This contradicts the fact that p_{old} is a skyband point of q^s_{max}. ■