2016-03-29

# Outlier Detection In Big Data

Lei Cao
*Worcester Polytechnic Institute*

# Outlier Detection In Big Data

by

Lei Cao

A Dissertation

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Doctor of Philosophy

in

Computer Science

by

March 28, 2016

APPROVED:

| | |
|---|---|
| Professor Elke A. Rundensteiner<br>Worcester Polytechnic Institute<br>Advisor | Professor Xiangnan Kong<br>Worcester Polytechnic Institute<br>Committee Member |
| Professor Mohamed Y. Eltabakh<br>Worcester Polytechnic Institute<br>Committee Member | Professor Tim Kraska<br>Brown University<br>External Committee Member |
| Professor Craig Wills<br>Worcester Polytechnic Institute<br>Head of Department | |

# Abstract

The dissertation focuses on scaling outlier detection to work both on huge static as well as on dynamic streaming datasets. Outliers are patterns in the data that do not conform to the expected behavior. Outlier detection techniques are broadly applied in applications ranging from credit fraud prevention, network intrusion detection to stock investment tactical planning. For such mission critical applications, a timely response often is of paramount importance. Yet the processing of outlier detection requests is of high algorithmic complexity and resource consuming. In this dissertation we investigate the challenges of detecting outliers in big data – in particular caused by the *high velocity of streaming data*, *the big volume of static data* and *the large cardinality of the input parameter space* for tuning outlier mining algorithms. Effective optimization techniques are proposed to assure the responsiveness of outlier detection in big data.

In this dissertation we first propose a novel optimization framework called LEAP to continuously detect outliers over data streams. The continuous discovery of outliers is critical for a large range of online applications that monitor high volume continuously evolving streaming data. LEAP encompasses two general optimization principles that utilize the rarity of the outliers and the temporal priority relationships among stream data points. Leveraging these two principles LEAP not only is able to continuously deliver outliers with respect to a set of popular outlier models, but also provides near real-

time support for processing powerful outlier analytics workloads composed of large numbers of outlier mining requests with various parameter settings.

Second, we develop a distributed approach to efficiently detect outliers over massive-scale static data sets. In this big data era, as the volume of the data advances to new levels, the power of distributed compute clusters must be employed to detect outliers in a short turnaround time. In this research, our approach optimizes key factors determining the efficiency of distributed data analytics, namely, communication costs and load balancing. In particular we prove the traditional frequency-based load balancing assumption is not effective. We thus design a novel *cost-driven* data partitioning strategy that achieves load balancing. Furthermore, we abandon the traditional one detection algorithm for all compute nodes approach and instead propose a novel *multi-tactic* methodology which adaptively selects the most appropriate algorithm for each node based on the characteristics of the data partition assigned to it.

Third, traditional outlier detection systems process each individual outlier detection request instantiated with a particular parameter setting one at a time. This is not only prohibitively time-consuming for large datasets, but also tedious for analysts as they explore the data to hone in on the most appropriate parameter setting or on the desired results. We thus design an interactive outlier exploration paradigm that is not only able to answer traditional outlier detection requests in near real-time, but also offers innovative outlier analytics tools to assist analysts to quickly extract, interpret and understand the outliers of interest.

Our experimental studies including performance evaluation and user studies conducted on real world datasets including stock, sensor, moving object, and Geolocation datasets confirm both the effectiveness and efficiency of the proposed approaches.

# Acknowledgements

The growth of my knowledge over the last few years culminating with my dissertation is to a huge part due to the inspiration and guidance I received from my advisor, Professor Elke A. Rundensteiner. She gave me the freedom to explore any topic in data analytics research, provided sound directions at every turn, and the prompt feedback that pushed my research envelope. I have been fortunate to have her as my advisor. I express my sincere thanks for her support, advice, patience, and encouragement throughout my graduate studies. Her excellence in teaching as well as tackling complex research problems will always be my inspiration to continue my research in the future.

I sincerely thank the members of my Ph.D. committee, Professor Mohamed Y. Eltabakh, Professor Xiangnan Kong, and Professor Tim Kraska for providing me valuable feedback during all milestones in my Ph.D studies. Their insight and critique helped me improve my research and the content of this dissertation. I would like to thank Professor Emmanuel O. Agu for his guidance during my research qualification. I am thankful for the financial support I have received from my advisor Prof. Elke A. Rundensteiner via National Science Foundation (NSF) research grants (IIS-1018443 and 0917017) and Computer Science department. My thanks also goes to the National Science Foundation (NSF) for providing funding for the computing resources used in my dissertation.

I would like to thank my collaborators Dr. Di Yang, Dr. Yanwei Yu, Yizhou

Yan, Mingrui Wei, Caitlin Kuhlman, Yingmei Qi, Dr. Di Wang, Medhabi Ray, Xiao Qin, and Qingyang Wang. My thank you also goes to all other previous and current DSRG members – in particular Dr. Chuan Lei, Dr. Mo Liu, Han Wang, Hao Loi, Xika Lin, and Dongqing Xiao for their insightful discussion, helpful feedback, and friendship.

I would like to thank my spouse Ying Zhang for her patience, support and love during the past few years. Her confidence in me and encouragement was in the end what made this dissertation possible. My parents, Shoushan Cao and Yuehua Song, receive my most sincere gratitude. Their passion to achieve bigger and better things ingrained in me is a drive to reach excellence.

# My Publications

## Publications Contributing to this Dissertation

In this context I have achieved research advances that are selectively included in this dissertation as detailed below.

### Topic I: Continuous Outlier Detection Over Data Streams

Topic I of this dissertation addresses the problem of continuously detecting outliers over high-volume streaming data.

1. Lei Cao, Di Yang, Qingyang Wang, Yanwei Yu, Jiayuan Wang, and Elke A. Rundensteiner, *Scalable Distance-Based Outlier Detection over High-Volume Data Streams*, **ICDE**, 2014, pages 76-87.

   *Relationship to this dissertation:* In this work, we propose LEAP – an optimized streaming outlier detection framework to support all major distance-based outlier variations in a unified way. Chapters 3 to 7 in Part I of this dissertation are based on this work.

2. Lei Cao, Qingyang Wang, Elke A. Rundensteiner, *Interactive Outlier Exploration In Big Data Streams*, **PVLDB 7(13)** 2014, pages 1621-1624 (Demo).

*Relationship to this dissertation:* In this demonstration paper, we present our VSOutlier system for supporting interactive exploration of outliers in big data streams. VSOutlier not only supports a rich variety of outlier types such as distance-based outliers, density-based outliers, angle-based outliers by leveraging innovative outlier detection strategies, but also provides a rich set of interactive interfaces to explore outliers in real time.

3. Lei Cao, Yanwei Yu, Elke A. Rundensteiner, *Detecting Moving Object Outliers In Massive-Scale Trajectory Streams*, **KDD** 2014, pages 422-431.

   *Relationship to this dissertation:* In this paper, we first propose several classes of novel trajectory outlier definitions that model the anomalous behavior of moving objects for a large range of real time applications. Furthermore, we propose a general strategy for efficiently detecting outliers that fall into these new outlier classes. This work features fundamental optimization principles designed to minimize the detection costs.

4. Lei Cao, Jiayuan Wang, Elke A. Rundensteiner, *Sharing-Aware Outlier Analytics over High-Volume Data Streams*, **SIGMOD** 2016, accepted.

   *Relationship to this dissertation:* In this paper we propose the first sharing-aware multi-query execution strategy for outlier detection on data streams called SOP. SOP supports a large variety of continuous outlier detection requests with real time responsiveness by minimizing the utilization of both computational and memory resources for the processing of such complex outlier analytics workload. Chapters 8 to 12 in Part II of this dissertation are based on this work.

## Topic II: Distributed Outlier Detection

Topic II of this dissertation develops highly scalable algorithms leveraging modern distributed infrastructures to detect outliers in massive datasets.

5. Lei Cao, Yizhou Yan, Caitlin Kuhlman, Elke A. Rundensteiner, and Mohamed Y. Eltabakh, *Distance-Based Outlier Detection At Scale,* in submission to a major conference.

   *Relationship to this dissertation:* In this work, we present DOD, the first distributed solution for distance-based outlier detection, designed using the MapReduce paradigm. DOD detects outliers in one single pass empowered by: (1) an efficient cost-driven data partitioning strategy to achieve load balancing, and (2) a multi-tactic outlier detection methodology for minimizing computation costs. Chapters 13 to 17 in Part III of this dissertation are based on this work.

6. Lei Cao, Yizhou Yan, Caitlin Kuhlman, and Elke A. Rundensteiner, *From K nearest neighbors to Local Outlier Factor Using MapReduce,* in preparation for submission to a major conference.

   *Relationship to this dissertation:* In this work, we present the first density-based outlier (LOF) detection solution using MapReduce called DLOF. First, DLOF features two distributed strategies for K nearest neighbor (kNN) search which is shown to be the most resource intensive step of LOF processing. These are a one-pass strategy and a multiple-pass strategy. Furthermore, in DLOF we propose a pruning strategy that quickly identifies and excludes inliers without even conducting the expensive KNN search. Chapters 18 to 22 in Part IV of this dissertation are based on this work.

## Topic III: Interactive Outlier Exploration

Topic III of this dissertation satisfies the need of quickly identifying true outliers that are of interest to the analysts and facilitating the interpretation of these mined outliers.

7. Lei Cao, Mingrui Wei, and Elke A. Rundensteiner, *Online Outlier Exploration Over Large Datasets,* **SIGKDD** 2015, pages 89-98.

   *Relationship to dissertation:* In this work, we present the first online outlier exploration platform, called ONION, that enables analysts to effectively explore anomalies even in large datasets. ONION features an innovative interactive anomaly exploration model that offers an "outlier-centric panorama" into big datasets along with rich classes of exploration operations. Chapters 23 to 26 in Part V of this dissertation are based on this work.

8. Mingrui Wei, Lei Cao, and Elke A. Rundensteiner, *Managing Outliers Over Big Data,* in submission to a major conference (software demonstration).

   *Relationship to dissertation:* This software demonstration presents the ONION interactive outlier exploration system. ONION system is composed of an offline preprocessing phase followed by an online analytic and exploration phase. It enables analysts to interactively explore outliers at near real-time speed even over large datasets. ONION features several novel visual interaction modes including the data, model, and parameter views.

## Other Publications

The below listed publications correspond to other research projects I have undertaken during my PhD at WPI mostly on the topic of stream query optimization.

9. Lei Cao and Elke A. Rundensteiner, *High performance Stream Query Processing with Correlation-Aware Partitioning,* **PVLDB 7(4)** 2013, pages 265-276.

10. Yingmei Qi, Lei Cao, Medhabi Ray, and Elke A. Rundensteiner, *Complex Event Analytics: Online Aggregation of Stream Sequence Patterns,* **SIGMOD** 2014, pages 229-240.

11. Yanwei Yu, Lei Cao, and and Elke A. Rundensteiner, *Outlier Detection over Massive-Scale Trajectory Streams,* under revision to a major journal.

12. Lei Cao and Elke A. Rundensteiner, *High Performance Multi-route Stream Query Processing,* in submission to a major journal.

# Contents

# List of Figures

# List of Tables

# 1

# Introduction

## 1.1 Motivation

As the advances in hardware, software and networks have enabled applications from business, scientific and engineering disciplines, social networks, to government, to generate data at unprecedented volume, variety, velocity, and varsity not seen before, discovering precious knowledge hidden in such data has become more critical than ever before.

Important insights extractable from such big data sources include *abnormal phenomena* or outliers. In general outliers are patterns in the data that do not conform to the expected behavior. Many modern applications ranging from credit fraud prevention, network intrusion prevention, climate change analysis to financial strategy planning rely on effective methods for detecting outliers to discover suspicious card usage and potential identity theft, to prevent cyber attack, to forecast disastrous weather phenomena, and to predict market changes and trade opportunities, respectively [1]. For such mission critical applications, a timely response often is of paramount importance.

**Outlier Detection Techniques.** Driven by the importance of outlier detection, a lot of work has focused on developing effective outlier detection techniques, including statistics based [2, 3], classification based [4, 5, 6, 7] and unsupervised detection techniques [8, 9, 10, 11].

*Statistical techniques* [2, 3] fit a statistical model to the given data and then apply a statistical inference test to determine if an unseen instance belongs to this model or not. Instances that have a low probability of being generated from the learned model, based on the applied test statistic, are declared to be outliers. Statistical techniques rely on the assumption that the data is generated from a particular distribution. This assumption often does not hold true for big real data sets.

The typical *classification-based approach* [4, 5, 6, 7] is to first learn a classifier using the available labeled training data. Then at the testing phase any unseen data instance is classified as normal or anomalous using the classifier. The testing phase in such case is fast, since each test instance only needs to be compared against the pre-computed model. However there are several issues that arise in supervised outlier detection. First, the anomalous instances are far fewer compared to the normal instances in the training data. Second, obtaining accurate and representative label, especially for the anomaly class, can be challenging. Furthermore, in many domains normal behavior keeps evolving and a current notion of normal behavior might not be sufficiently representative in the future.

The *unsupervised techniques* [8, 9, 10, 11] apply the concept of nearest neighbor analysis. They are based on the following key observation: normal data instances occur in dense neighborhoods, while outliers occur far from their closest neighbors. These unsupervised techniques are *widely applicable* [1] for the following reasons: (1) they do not make any assumption regarding the generative distribution for the data. Instead they are purely data driven; (2) they do not require training data which usually is difficult to acquire in the real world; (3) adapting nearest neighbor-based techniques to different data types is straightforward, and primarily requires defining an appropriate distance measure for the given data. Therefore by nature they are friendly to the *large variety* of big data types.

However, the computational complexity of unsupervised techniques is a significant challenge when handing *big data* since unsupervised techniques involve computing the distance between each pair of data instances to compute the nearest neighbors for each point. Although approximation based solutions such as sampling in big static data [12, 13, 14] or load shedding [15, 16, 17] in high velocity streaming data have been adopted to relieve the problem of high computation complexity of other analytics tasks, applying such an approximation-based solution in outlier detection might miss critical outliers that may indicate a credit card fraud or a severe security breach. Furthermore, similar to many other data mining techniques, the performance of unsupervised outlier detection greatly relies on the user-specified input parameter setting. For example a density threshold may define how sparse the neighborhood should be for a given data instance to be considered an outlier. Specifying an appropriate input parameter setting is challenging, since the number of possible parameter options (the cardinality of the *input parameter space*) tends to be huge or infinite.

Therefore although extensive research effort has been made on unsupervised techniques in the literature [18, 19], the current approaches are still not sufficient to *effectively capture true outliers hidden in big datasets in a timely manner*. As shown in [18] the largest dataset that had been tested in the literature is smaller than 1G, while no technique [19] can efficiently handle streaming data when the velocity is higher than 1M per second. Significant research remains to be undertaken to scale these widely applied unsupervised outlier detection techniques to big data. This is thus the focus of this dissertation.

Overall in this dissertation we thoroughly investigate and address the challenges caused by *high velocity streaming data*, *big volume static data* and *large cardinality of the input parameter space* and thus fill the void in the literature of effectively detecting outliers over big data. Other interesting problems such as how to select an appropriate distance metric to scale unsupervised outlier detection to various types of data (big variety), proposing methods that can quickly approximate the outliers yet without missing critical outliers, detecting outliers from uncertain data and supervised outlier detection under imbalanced class distributions of labeled data, are out of the scope of this dissertation.

**High Velocity Streaming Data.** The number of mobile devices, such as smart phones, pads, and RFID equipment, and their capabilities of generating and transmitting live data have both grown rapidly in recent years. As the volume and speed of data streams advance to new levels, discovering outliers hidden in this data has become more challenging than ever before. For example when monitoring the stock trading streams on the stock market, investors may *continuously* look for the *recent* outlier stocks whose behavior significantly differs from that of the majority of their peer stocks. Such abnormal stocks typically are either the *hot spots* or some *forgotten treasure* in the market. Both of them may correspond to potentially excellent investment opportunities. 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 the similar income levels. In such applications real time responsiveness is extremely important. Even a one-second delay may lead to a loss of huge funds and investment opportunities. Satisfying this stringent response time requirement is challenging when the stock or bank transactions are reported at several thousand transactions per second rate. Therefore *efficient continuous processing strategies* that are able to mine outliers at real-time from *extremely fast streams* must be developed.

Furthermore, a stream processing system may face a difficult challenge when it needs to handle not only one but potentially a large number of outlier mining queries simultaneously. Since outlier detection queries are usually parameterized, different analysts may submit queries of the same type but with different parameters settings to a single system, based on their own domain knowledge and their specific analysis tasks.

In the stock market application, with a large number of analysts monitoring the same stock transactions stream from NYSE every day [20] each may customize their outlier search requests by tuning their parameter settings using their personalized interpretation of abnormity. For example when utilizing an outlier detection technique to capture the stocks that dropped or rose significantly in the most recent transactions, each analyst has to define their own notions of “significance” in price fluctuation (e.g., 10, 30 or 50 percent of the original price) and the meaning of “most recent” transactions (e.g., transactions that happened in last 5, 10 or 30 minutes) based on their application semantics. Given the high algorithmic complexity of most outlier mining algorithms, serving a large number of such outlier mining queries in a single system is extremely resource intensive. The naive method of executing each query independently has prohibitively high demands on both computational and memory resources. Therefore *efficient shared execution strategies* for multiple outlier mining queries over data streams must be designed.

**Big Volume Static Date.** Nowadays various applications are generating data at unprecedented scale. For example based on the statistics from IBM [21], in social network applications 30 billion pieces of content are shared every month on Facebook, while 400 millions tweets are sent per day by about 200 million monthly active users. In the stock market the New York Stock Exchange captures 1 TB of trade information during each trading session. While in computer network it is projected that by 2016 there will be 18.9 billion network connections. Clearly one single machine even when configured with most advanced hardware does not have the storage resources to accommodate such huge amount of data nor the CPU power to quickly detect outliers.

Therefore in this big data era the development of a distributed solution for outlier detection that effectively leverages distributed computing is not an option, but a necessity. The MapReduce-based [22] platforms such as Hadoop [23] and Spark [24] are among the most popular distributed infrastructures due to their many desirable features including scalability to thousands of machines, flexibility in the data model, efficient fault tolerant execution, and cost effectiveness. Nevertheless, despite the importance of outlier detection and the popularity of the MapReduce distributed computing paradigm, to the best of our knowledge no work has been proposed to date to support outlier detection on the MapReduce-based infrastructures. Therefore, *efficient distributed outlier detection approaches* must be designed to handle terabyte or even petabyte level big datasets.

**Large Cardinality of Input Parameter Space.** Last but not least, traditional outlier detection systems require the analyst to select a fixed set of parameter values, and then to submit this instantiated request to attempt to detect outliers of interest. This request is then executed from scratch as a one time query to compute the outliers from the target dataset that match that specification. This one-at-a-time query approach suffers from severe limitations.

First, using the current systems, to acquire a good input parameter setting the analyst has to continuously re-submit individual requests with different parameter settings in a trial-and-error fashion and then manually analyze the respective results. This is extremely ineffective and corresponds to a taxing process for the analysts because of the sheer infinite number of possible parameter settings.

Second, mining outliers according to a particular parameter setting from scratch on large data tends to take hours or more as confirmed in our experiments [25]. This is clearly not within the tolerable response time range for the analysts during the process of *parameter tuning* and *outlier examination*.

Furthermore, important insights, such as how the detected outlier set changes when varying parameter settings, or what the relationship among different outlier points is (for example, whether some points are "stronger" outliers than others), might be missed during this tedious yet expensive exploration process. This information is critical for the analysts to interpret the characteristics of the outliers hidden in the dataset.

In short, the traditional one-at-a-time approach is neither effective nor efficient for modern outlier analytics applications. Therefore methodologies that allow users to *interactively analyze the outliers in the big dataset and easily locate appropriate parameter setting* are needed to address these shortcomings.

## 1.2 State-Of-the-Art

### 1.2.1 Continuous Outlier Detection over Streams

In recent years researchers started to look at the problem of detecting outliers in streaming environments [26, 27, 28]. Specifically [26, 28] proposed solutions for detecting outliers in count-based sliding windows. [27] improves upon this solution [26] by now supporting outlier detection in time-based sliding windows. All solutions leverage the overlap of sliding windows and thus avoid huge overhead wasted on recomputing-from-scratch at each window.

However, these existing techniques [26, 27, 28] didn't explore the optimization opportunities enabled by the *critical observation* below. That is, they didn't exploit the fact that outliers by nature only constitute a small portion of the general stream data population (otherwise they wouldn't be called outliers after all). Thus, the outlier detection algorithms should ideally concentrate their resource utilization on strictly serving these minority outlier candidates, rather than the general and much larger stream population. Without this important optimization opportunity, the existing techniques [26, 27, 28] cannot handle high-speed streams in real-time, say 1M tuples per second, as confirmed by the experiments in [19]. Yet such huge volume streams are increasingly common in modern streaming applications. As an example the US stocks market continuously receives around 1M transaction requests per second [20].

Furthermore, all of the above approaches focus on handling *one single outlier request* with a fixed parameter setting [26, 27, 28]. The simultaneous execution of multiple outlier mining queries with varying parameter settings remains unexplored.

In the broader area of shared execution strategies for multiple queries in streaming environments, the main focus of previous work has been on Select-Project-Join [29, 30, 31] and aggregation queries [32, 33]. These 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 aim to address in this dissertation is different from the more general-purpose optimization effort required by the traditional SQL query sharing. The semantics of outlier detection request does not contain any subexpression. Furthermore, usually no partial result will be generated and possibly shared in the mining process.

The only work we are aware of that supports simultaneous execution of multiple mining requests is [34]. It presents shared execution strategies for processing a workload of density-based clustering requests instead of outliers [34]. This work organizes cluster structures identified by multiple queries into one compact hierarchical data structure. The incremental maintenance of this structure for progressive clustering is then proposed. However outliers are defined as individual data points unlike clusters that can be modeled as connected structures with set-based inter-dependencies among the data points. Therefore the solution in [34] cannot be applied in our context.

### 1.2.2 Distributed Outlier Detection

A few recent efforts have been proposed to support distributed outlier detection [35, 36, 37, 38]. Among these works some focus on outlier definitions [37, 38] specific to particular domains (such as health care or network intrusion). These definitions leverage the domain knowledge having already been discovered by the experts in the particular fields to classify outliers. Therefore it is difficult to generalize such approaches to other fields. Others utilize different distributed computing paradigms that, unlike MapReduce, either suffer from a bottleneck by requiring a central master node to split and broadcast data to each slave node [36], or they allow all nodes unrestricted exchange of data with each other [35]. In contrast, the MapReduce infrastructure neither assumes a central node nor does it allow data exchange among the mappers (nor among the reducers) to enable easier distribution of tasks and higher scalability. Therefore none of these techniques is applicable on the MapReduce infrastructure. To support distributed outlier detection on MapReduce, partitioning approaches that make sure each reducer can detect outliers independently from other reducers have to be designed.

In the literature Map-Reduced based approaches have been proposed for other advanced analytics techniques such as similarity join, KNN-Join, and clustering [14, 39, 40]. Although they investigate the key concepts in distributed systems such as *load balancing* and *efficient partitioning* that determine the efficiency of distributed mining algorithms, their methods cannot be applied in our outlier detection area as shown below.

For example in [39] the authors propose an efficient similarity join algorithm called *MR-MAPSS* that studies the load balancing problem. *MR-MAPSS* first partitions the input data records into work sets. It then achieves load-balancing across the reducers by splitting and repartitioning the densely clustered large work sets. However their *load-balancing* method relies on the traditional load balancing assumption, namely an equal number of data points indicates equal work load. This assumption is proven to be not true in outlier detection by our experimental investigation and theoretical analysis [25].

In [40] an approximation KNN-Join algorithm is proposed on MapReduce. Their *partitioning* method ensures that each reducer produces results in total isolation with other reducers. It first maps the multi-dimensional data sets into a single dimension data using space-filling curves (z-values), and then transforms kNN join into a sequence of one-dimensional range searches. Thus the *partitioning* of a multi-dimensional dataset is reduced to an equal size one-dimensional partitioning problem. Although this problem can be efficiently solved for uniform datasets, the approximation accuracy is highly compromised when handling skewed data. Furthermore, its equal size partitioning method might lead to a highly unbalanced workload across different reducers.

In conclusion new approaches with innovative *partitioning* strategies and *load balancing* methodologies have to be developed to efficiently detect outliers by leveraging the Map-Reduce paradigm.

### 1.2.3 Interactive Outlier Exploration

To the best of our knowledge, the problem of interactively exploring outliers in big datasets proposed to be tackled in this dissertation has not been considered before in the literature.

In the context of other data mining techniques, [41] proposes the *PARAS* model to support the exploration of association rules in the interactive manner. By leveraging the redundancy relationships among rules and the common rules shared by the similar parameter settings, PARAS maintains the final rule sets with respect to all parameter settings (all possible combinations of support and confident values). Using the proposed index, association rule mining requests can thus be answered by PARAS with near real-time responsiveness as confirmed by their experiments [41]. However, in contrast to PARAS we focus on outlier exploration that is distinct from association rule mining. Association rule mining is supported by first discovering all frequent item sets. Clearly these set based techniques cannot be applied to detect individual point based outliers. Furthermore, PARAS has to maintain the final results with respect to all parameter settings, while the number of association rules generated by large datasets is prohibitively large. Therefore this solution is not scalable to large datasets. In order to interactively analyze outliers over large datasets other approaches have to be explored instead of extensively maintaining all possible final results.

In density-based clustering, the *OPTICS* algorithm [42] produces the clusters in a time complexity linear in the size of the dataset with respect to any parameter settings picked in a given parameter setting range. *OPTICS* achieves this by creating an augmented ordering of the dataset to represent the clustering structure corresponding to a set of parameter settings. However, producing outliers as by-products of clustering has already been shown to be not effective in capturing abnormal phenomena [18]. Furthermore, the ordering information is only effective in representing the clusters with respect to a small range of parameter settings. Our work instead aims to support any outlier detection request with any possible parameter setting in a near real-time fashion.

## 1.3 Research Challenges Addressed in This Dissertation

**Continuous Outlier Detection Over Data Streams.** First, designing scalable stream outlier detection strategies that satisfy the stringent response time requirements of online monitoring applications is extremely difficult, because the processing of an outlier detection request is resource-consuming due to the algorithmic complexity of the mining process. As shown in [1], the algorithmic complexity of most outlier detection techniques is known to be quadratic with respect to the number of points. Continuously mining outliers from high volume, high velocity stream data is like mining needles in a haystack. There is so much hay to mine and so little time to utilize.

Second to handle a large workload composed of hundreds or even thousands of outlier requests over data streams in real time, effective sharing of system resources utilized for the processing of each of these queries must be achieved. However outlier mining requests with different parameter settings may cause totally different outliers to be identified. Furthermore, 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. Therefore a sharing-aware execution strategy that completely avoids the redundant computation across the process of different outlier detection requests on data is hard to develop.

**Distributed Outlier Detection.** The design of an efficient distributed outlier detection algorithm is challenging.

First, designing an effective partitioning strategy for the MapReduce-based outlier detection approach is challenging. Intuitively the default partitioning solution in MapReduce would randomly spread the information that is necessary to prove the status of one point into possibly numerous nodes. Therefore a point $p$ would not be able to prove its outlier status on the local reducer node on which $p$ resides. This inevitably would lead to a multi-pass solution, thus introducing heavy communication costs due to requiring a repeated re-distribution of the whole dataset. On the other hand partitioning the data points with similar characteristics to the same node might be able to preserve the norm on the local node for each data point to evaluate its abnormity. However real world datasets tend to be skewed [43] instead of being uniformly distributed over their domain spaces. For this reason, data characteristics-based partitioning suffers from the problem that the number of points allocated to each node may vary extremely − leading to an unbalanced workload.

Second, a common limitation in distributed analytics work [35, 36, 37, 38] is that they apply one single detection algorithm to all compute nodes. This "monolithic" detection approach is based on the implicit assumption that there is one outlier algorithm that is superior to all others for all types of datasets. However, we observe that although numerious centralized algorithms have been proposed to speed up the outlier detection process, e.g., [8, 44], none of them has shown consistent superiority in all circumstances. Instead, the performance strongly varies depending on the characteristics of the dataset being processed. Since the data partitions in a distributed environment may each have different characteristics, this “monolithic” detection approach misses important optimization opportunities to minimize the overall costs of the distributed outlier detection process. To solve this problem we must assign an appropriate detection algorithm to each partition based on its characteristics. This requires a thorough understanding of the correlations between the characteristics of the data and the performance of the algorithms. However, to date no such work appears in the literature.

Third, the partition generation problem (Challenge 1) and the algorithm-selection problem (Challenge 2) are strongly interdependent, i.e., a change in one may cause a modification in the other. For example, to minimize the overall detection costs, the effectiveness of a partitioning plan should be evaluated based on the costs estimated from the detection algorithms assigned to each partition. On the other hand, the algorithm assignments must be determined based on the characteristics of the data subsets produced by the partitioning plan. This raises the proverbial chicken and egg question.

**Interactive Outlier Exploration.** Designing an interactive outlier detection system that effectively derives the outliers of interest to the analysts with real time responsiveness, thereby meeting the requirements of online analytics applications, is challenging. It is challenging to design an interactive system that can recommend appropriate parameter settings and allow users to online analyze outliers over big data.

First, due to the algorithmic complexity of mining techniques [45], processing each outlier request from scratch over big datasets each time when it is submitted clearly cannot satisfy the response time requirement of interactive systems. On the other hand precomputing and storing the results for all potential detection requests beforehand on first sight appears infeasible because of the infinite number of possible parameter settings.

Furthermore, even if this were achievable, it would remain hard if not impossible for an analyst to figure out the most appropriate parameter setting in a trial and error fashion. Worst yet together the cardinality of the big dataset and the sheer number of possible parameter settings make the problem of explicitly modeling the relationships among the outliers and the trends of how the detected outlier set migrates across distinct parameter settings almost intractable. However this is inevitable for the analysts to be able to analyze the characteristics of the outliers.

## 1.4 Proposed Solutions

In this dissertation, we focus on the unsupervised outlier detection approaches [1, 46]. Different unsupervised outlier detection approaches can be categorized as global versus local approaches [46], i.e., the decision on the outlierness of a data point $p$ can be based on the complete (global) dataset or only on a (local) selection of adjacent data points. We investigate the typical detection techniques of both categories, namely distance-based outliers [8, 9] of the global approach and density-based outliers [10] of the local approach.

The seminal distance-based technique proposed in [8] computes the anomaly score of a data instance by counting the number of neighbors ($k$) that are not more than $r$ distance apart from the given data instance, while in another major variation of distance-based technique proposed in [9] the anomaly score of a data instance is defined as its distance to its $k$th nearest neighbor in a given data set.

The density-based technique in [10] assigns an anomaly score to a given data instance, known as Local Outlier Factor (*LOF*). For any given data instance, the *LOF* score is equal to the ratio of average local density of the $k$ nearest neighbors of the instance and the local density of the data instance itself. The local density is represented by the radius of the smallest hyper-sphere centered at the data instance that contains its $k$ nearest neighbors.

In this dissertation we thoroughly investigate the problem of making unsupervised outlier detection techniques effective yet efficient in big static and streaming data. Fundamental observations and optimization principles that are not only general across a rich variety of unsupervised outlier detection techniques, but also applicable to other big data mining problems are proposed to attack the challenges of *Continuous Outlier Detection Over Data Streams*, *Distributed Outlier Detection* and *Interactive Outlier Exploration*.

### 1.4.1 Continuous Outlier Detection Over Data Streams

***Single Outlier Detection Request.*** We design an efficient continuous processing strategy called **LEAP** to mine outliers from extremely fast streams [19]. To satisfy the stringent response time requirement of the online monitoring applications, the design of this strategy explores the fundamental optimization opportunities enabled by the general properties of the unsupervised outlier definitions in streaming data.

First, this strategy takes advantage of the rarity property of outliers. Given a dataset D, the majority of points in D are guaranteed to be inliers. Furthermore, given a data point *p* in *D* potentially examining a small subset of points in *D* will be sufficient to prove that $p$ is an inlier. Therefore an efficient outlier detection algorithm should be able to quickly eliminate inliers by collecting the *least amount of evidence* necessary to prove the inlier status of the data points (inlier evidence).

Second, this strategy fully utilizes the temporal relationships among stream data points. The data points that arrived later in the window are guaranteed to have a more decisive impact on the outlier detection process compared to earlier ones. This is so because the younger a data point $p$ is, the longer its contribution of proving the outlier status of other points will persist into the future. Since the key task for the outlier detection process is to eliminate any guaranteed inliers, identifying enough longer lasting inlier evidence is likely to eliminate the need for further examination for those shorter lasting ones.

In particular our contributions include:

1. We present the first result on efficiently supporting the major distance-based outlier classes. In particular neither the $O^{(k,n)}_{kmax}$ [9] nor the $O^{(k,n)}_{kavg}$ [47] outliers had been handled in the streaming outlier detection literature to date.

2. We propose the *minimal probing* optimization principle, which frees detection algorithms from the burden experienced by the state-of-the-art methodologies of having to conduct range query searches [26, 27, 48].

3. We introduce the *lifespan-aware prioritization* principle, which guides the outlier detection algorithms to probe neighbors for stream data points in a time-aware manner to minimize the frequency of probing operation.

4. We integrate these two principles into a general framework called LEAP, which is proven to be optimal in terms of the CPU costs for determining the outlier status of each point.

5. Our experimental studies based on real and synthetic data show that our proposed algorithms achieve three orders of magnitude performance gain compared to the state-of-the-art techniques in a rich variety of scenarios.

***Multiple Outlier Detection Requests.*** We propose the **SOP** strategy to efficiently handle an outlier analytics workload composed of a large number of outlier detection requests with arbitrary parameter settings. This includes optimization to guarantee the full sharing of both CPU computations and memory utilization for the processing of the outlier analytics workload. In particular computation-wise, in each active window it only requires a single pass through the batch of the data points to answer all requests. Memory-wise, it assures that only one single copy of the neighbor information shared across all requests is maintained. The key observation here is that certain relationships exist among the outliers generated by different mining requests. For example some mining requests might generate the same set of outliers although they are configured with different parameter settings. Some parameter settings are “more restricted” than others in terms of recognizing outliers, therefore guaranteed to generate more outliers than the others.

The contributions in this area 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 [19, 27].

### 1.4.2 Distributed Outlier Detection

We design scalable distributed approaches to detect both distance-based and density-based outliers from high volume static data. In general our approaches feature novel partitioning strategies to be deployed on mappers and outlier detection strategies to be deployed on reducers. These strategies together minimize the overall execution costs that in a distributed system correspond to both communication and processing costs.

First at the mapper side our supporting area partitioning strategy makes each partition self-sufficient yet only introducing a minimal amount of data replication. This minimizes the costs that are required when having to re-distribute data repeatedly between mappers and reducers. Furthermore, an imbalanced workload may not only result in a significant slowdown in processing time, but also risk job failure in some cases. Therefore the partitioning strategy has to ensure that each reducer is assigned a balanced workload. The key observation here is that to achieve load balancing, assigning an equal number of data points to each reducer is not sufficient. Instead the partitioning strategy should also take into consideration the data distribution of the mined dataset and the costs estimated by the cost model for the detection algorithm to be applied.

At the reducer side the traditional MapReduce based mining algorithms assume that one single mining algorithm is applied to all reducers. However the data partitions in different nodes might have different characteristics. Therefore the best algorithm selected based on the overall characteristics of the whole dataset might not serve any of the data partitions well. Therefore we propose a novel *multi-tactic* paradigm. This model makes use of the fact that each reducer in a computer cluster executes independently of each other. Hence they may run different outlier detection algorithms as needed, namely *different algorithms for different reducers*.

The key contributions in this area include:

1. We propose the first distributed approach called *DLOF* that effectively solves the problem of detecting density-based outliers from large volume data.

2. We design a distributed framework called *DOD* that, using the *supporting area*

technique, detects all distance-based outliers in a single MapReduce job. *DOD* is proven to involve minimal communication overhead.

3. For the first time, we theoretically analyze and contrast the costs of distinct classes of outlier detection algorithms under various data distributions. Based on this theoretical foundation we prove that the traditional frequency-based load balancing assumption does not hold in the outlier detection context, and propose a novel *cost-driven* strategy that effectively generates partitions of balanced workloads.

4. We propose a *multi-tactic* strategy that automatically selects the best outlier detection algorithm for a given data partition. Our proposed density-aware technique successfully separates the two interdependent problems of partition-generation and algorithm-selection.

5. We experimentally evaluate our *DLOF* and *DOD* techniques using TBs of data. The results demonstrate that our techniques outperform the baseline solutions by a factor of 15x.

### 1.4.3 Interactive Outlier Exploration

In this area we propose to design an interactive outlier exploration paradigm called **ONION** to meet the requirements of online outlier analytics applications. First, ONION is able to answer any outlier detection request with real time responsiveness. Second, it assists the analysts to quickly pinpoint a good parameter setting fitting the datasets to be mined in a systematic way. Furthermore, it facilitates the understanding and interpretation the mined outliers.

In general **ONION** is composed of two phases, namely offline phase and online phase. At the offline phase we extract and abstract the key components of outlier analytics, namely the input data, the input parameter settings along with the possible outliers generated from the data together into a comprehensive yet compact knowledge base by utilizing the power of computer clusters. The key observation here is that most of the data points are guaranteed to be inliers no matter how the parameter setting changes. The offline phase only needs to be conducted once. At the online phase we provide the users a rich set of analytics tools. It not only supports the traditional outlier mining operation, that is, given a particular input parameter, asking for the generated outliers. It also supports other novel analytics operations such as given a set of outliers, returning the parameters generating this outlier set, or given a set of sampling outliers, output other outliers similar to these samples. Furthermore, these analytics operations will not be conducted on the raw big dataset and the large parameter space any more. They will be supported by directly looking at the aggregated knowledge base built in the offline phase. Therefore they can be answered in real time. These analytics operations in combination provide a powerful yet flexible tool for users to explore the data and interpret the generated outliers as well as recommend appropriate parameter settings to the users.

In particular the key contributions in this area include:

1. We propose the first interactive outlier analytics platform that enables analysts to pinpoint appropriate parameter settings and explore outliers in a systematic way.

2. We establish for the analysts an “outlier-centric panorama” into big datasets by integrating the input data and parameter space into a comprehensive multi-space ONION knowledge base.

3. We design logarithmic-complexity algorithms for the processing of each outlier exploration operations with real-time responsiveness by leveraging the compact ONION knowledge base.

4. We confirm the superiority of ONION compared to the traditional mining platform

in effectiveness of recognizing true outliers by conducting a user study with real GroundMoving Target Indicator (GMTI) dataset.

5. Our experimental performance study demonstrates that ONION is at least five orders of magnitude faster than its state-of-the-art competitors for traditional outlier detection queries.

## 1.5 Dissertation Organization

The rest of this proposal is organized as follows. Chapter 2 first provides the background and preliminary materials needed for this dissertation. We then discuss in detail the three research topics of this dissertation, namely *Continuous Outlier Detection Over Data Streams* in Part I (Chapters 3-7) and Part II (Chapters 8-12), *Distributed Outlier Detection* in Part III (Chapters 13-17 and Part IV (Chapters 18 to 22), and *Interactive Outlier Exploration* in Part V (Chapters 23-26), respectively. The discussion of each of the three research topics includes the problem formulation and analysis, description of the proposed solution, experimental evaluation, and lastly a discussion of related work. Chapter 27 concludes this dissertation and Chapter 28 discusses promising future work.

# 2

# Preliminaries

In this dissertation, we study the unsupervised outlier detection that is categorized as global outliers and local outliers. we focus on typical detection techniques of both categories, namely distance-based outliers in global outlier category [8, 9] and density-based outliers [10, 11] in local outlier category. In this chapter we give the definitions of distance-based outlier notion, density-based outlier notion, and the corresponding streaming outlier detection concepts. The Hadoop distributed platform that we utilize to detect outliers in large volume dataset is also briefly introduced.

## 2.1 Distance-Based Outlier

The work of Knorr and Ng on the distance-based notion of outliers (DB-outlier) [8] unifies statistical distribution-based approaches and triggered the data mining community to develop different approaches that have a less statistically oriented but more spatially oriented notion to model outliers. The idea is based on statistical reasoning but simplifies the approach to outlier detection considerably motivated by the need for scalable methods handling huge datasets.

*DB-Outlier* uses a range threshold $r \geq 0$ to define the *neighborship* between any two data points. For two data points $p_i$ and $p_j$, if the distance between them is no larger than $r$, $p_i$ and $p_j$ are said to be neighbors. Any distance function can be plugged to calculate the distance. We use the function $NumNei(p_i, r)$ to denote the number of neighbors a data point $p_i$ has, given the $r$ threshold.

**Definition 2.1** ***Distance-Based Outlier (DB-Outlier):*** *Given a distance range threshold $r$ and a neighbor count threshold $k$, a distance-based outlier is a data point $p_i$, where $NumNei(p_i, r) < k$.*

Inspired by [8] two variations of distance-based outlier notion were proposed in [9] and [47] that are both defined based on the well-known notion of "k-nearest neighbors ($k$NN)". Given a data point $p_i$ and its *kth-nearest neighbor* $p_j$, $d(p_i, p_j)$ is called the $k$NN maximum distance of $p_i$ denoted as $D^{kmax}(p_i)$, while the average distance to all its *k-nearest neighbors* is called the $k$NN average distance of $p_i$ denoted as $D^{kavg}(p_i)$. $D^{kmax}(p_i)$ is also called k-distance.

**Definition 2.2** *Given input parameters $k$ ($k \geq 1$) and $n$ ($n \geq 1$), a point $p_i$ is a **kNN maximum distance outlier** denoted by $O^{(k,n)}_{kmax}$ in D if at most n-1 other points $p_j$ exist with $1 \leq j \leq n-1$ in D such that $D^{kmax}(p_j) > D^{kmax}(p_i)$.*

**Definition 2.3** *Given input parameters $k$ ($k \geq 1$) and $n$ ($n \geq 1$), a point $p_i$ is a $k$**NN average distance outlier** denoted by $O^{(k,n)}_{kavg}s$ in $D$ if at most n-1 other points $p_j$ exist with $1 \leq j \leq n-1$ in D such that $D^{kavg}(p_j) > D^{kavg}(p_i)$.*

## 2.2 Density-Based Outlier

Density-based approaches consider ratios between the local density around an object and the local density around its neighboring objects. These approaches introduce the notion of *local outliers*. The concept of a local outlier is important since in many applications, different portions of a dataset can exhibit very different characteristics, and it is more meaningful to decide on the outlying possibility of a point based on other points in its neighborhood. However, unlike distance-based outlier local outlier model is not effective in detecting outlying clusters, namely the clusters in the sparse region. The original density-based outlier approach is introduced by Brenunig et. al. in [10]. The basic idea is to assign a density-based local outler factor (LOF) to each object of the dataset denoting a degree of outlierness.

We begin with the notion of the *k-distance neighborhood* of point $p$.

**Definition 2.4** *(k-distance neighborhood of an object p)*

*The k-distance neighborhood of p contains every object in dataset D whose distance from p is not greater than the k-distance, is denoted as:*

$N_k(p) = \{q \in D \setminus \{p\} \mid d(p,q) \leq k - distance(p)\}.$

Note that since there may be more than k objects within k-distance(p), the number of objects in $N_k(p)$ may be more than k. Later on, the definition of LOF is introduced, and its value is strongly influenced by the k-distance of the objects in its k-distance neighborhood.

**Definition 2.5** *(Reachability distance of p w.r.t point o)*

*The reachability distance of point p with respect to point o is defined as:*

$reach - dist_k(p,o) = max\{k - distance(o), dist(p,o)\}$

**Definition 2.6** *(Local reachability density of p)*

*The local reachability density of a point p is the inverse of the average reachability distance from the k-nearest-neighbors of p defined by:*

$lrd_k(p) = 1/[\frac{\sum_{o\in N_k(p)} reach-dist_k(p,o)}{|N_k(p)(p)|}]$

Essentially, the local reachability density of an object p is an estimation of the density at point p by analyzing the k-distance of the points in $N_k(p)$. The local reachability density of p is just the reciprocal of the average distance between p and the points in its k-neighborhood. Based on local reachability density, the local outlier factor can be defined as follows.

**Definition 2.7** $LOF_k(p) = \frac{\sum_{o \in N_k(p)} \frac{lrd_k(o)}{lrd_k(p)}}{|N_k(p)(p)|}$

LOF is the average of the ratio of the local reachability density of p and those of p's k-nearest-neighbors. Intuitively, p's local outlier factor will be very high if it is local reachability density is much lower than those of its neighbors.

## 2.3 Global Outlier Versus Local Outlier

In a rather general sense, the nature of outlier detection requires the comparison of an object with a set of other objects w.r.t some property (e.g., the kNN distance or a density model). When comparing different outlier detection methods, we find different levels of restriction of the set to compare with. Furthermore, the property to be compared is usually also derived from the dataset taking into account again a set of other objects. Both sets, namely set A from which to derive the property for an object and set B to compare with, need not be identical. We can name set A the context set for model building, and set B the reference set for model comparison.

This decomposition has been implemented gradually (and probably only to a certain extent intentionally) during the development of outlier detection methods as surveyed in the previous section. Consider the fundamental statistical methods. They are modeling the complete dataset by a single distribution and judging an object basically by the probability of whether it could have been generated by the corresponding model. In this case, both the model building set and the reference set are the complete dataset.

The first approach to DB-outlier detection already considers the local neighborhood by means of a range-query but compares the property thus derived with the complete dataset. The same is true for kNN-related outlier models: the model building set are the kNNs while the derived property is compared with the properties of the complete dataset as a reference. Therefore they are called global outliers.

The meaning of "locality" introduced in LOF relates to the locality of the reference set as well as the model building set. LOF uses the same neighborhood for both situations, but it could easily be abstracted to use different neighborhoods.

## 2.4 Outlier Detection in Sliding Window Streams

Periodic sliding window semantics as proposed by CQL [49] are widely utilized for defining the substream of interest from the otherwise infinite data stream. Such semantics can be either time or count-based. Each query $Q$ has a fixed window size $Q.win$ and slide $Q.slide$. For time-based windows each window $W_c$ of $Q$ 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 $Q.slide$. For count-based windows, a fixed number (count) of data points corresponds to the window size $Q.win$. The window slides after the arrival of $Q.slide$ new data points.

Outliers will be generated based on the points that fall into the current window $W_c$, namely the population of $W_c$. A point $p_i$ in $W_c$ might have 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. Now we define the stream outlier detection problem we tackle.

**Definition 2.8** ***Outlier Detection In Sliding Window Stream:*** *Given a stream $S$, a streaming distance-based outlier detection query $Q$ with $O^{(k,R)}_{thres}$, $O^{(k,n)}_{kmax}$, $O^{(k,n)}_{kavg}$, LOF defined in Def. 2.1, 2.2, 2.3, 2.7 with window size as $Q.win$ and slide size as $Q.slide$, $Q$ continu-*

*ously detects and outputs the outliers in the current window $W_c$ when the window slides.*

## 2.5 MapReduce Basics

MapReduce is a framework for parallel processing of massive data sets popular for its scalability to thousands machines, flexibility in the data model, efficient fault tolerance execution, and cost effectiveness. A job to be performed using the MapReduce framework has to be specified as two phases: the map phase as specified by a Map function (also called mapper) takes key/value pairs as input, possibly performs some computation on this input, and produces intermediate results in the form of key/value pairs; and the reduce phase which processes these results as specified by a Reduce function (also called reducer). The data from the map phase are shuffled, i.e., exchanged and merge-sorted, to the machines performing the reduce phase. It should be noted that the shuffle phase can itself be more time-consuming than the two others depending on network bandwidth availability and other resources.

In more detail, the data are processed through the following 6 steps as illustrated in Figure 2.1:

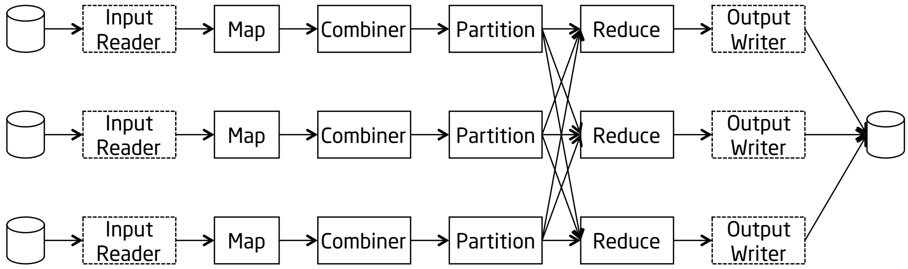


**Figure 2.1:** MapReduce Dataflow

1. Input reader: The input reader in the basic form takes input from files (large blocks) and converts them to key/value pairs. It is possible to add support for other input

types, so that input data can be retrieved from a database or even from main memory. The data are divided into splits, which are the unit of data processed by a map task. A typical split size is the size of a block, which for example in HDFS is 64 MB by default, but this is configurable.

2. Map function: A map task takes as input a key/value pair from the input reader, performs the logic of the Map function on it, and outputs the result as a new key/value pair. The results from a map task are initially output to a main memory buffer, and when almost full spill to disk. The spill files are in the end merged into one sorted file.

3. Combiner function: This optional function is provided for the common case when there is (a) significant repetition in the intermediate keys produced by each map task, and (b) the user-specified Reduce function is commutative and associative. In this case, a Combiner function will perform partial reduction so that pairs with same key will be processed as one group by a reduce task.

4. Partition function: As default, a hashing function is used to partition the intermediate keys output from the map tasks to reduce tasks. While this in general provides good balancing, in some cases it is still useful to employ other partitioning functions, and this can be done by providing a user-defined Partition function.

5. Reduce function: The Reduce function is invoked once for each distinct key and is applied on the set of associated values for that key, i.e., the pairs with same key will be processed as one group. The input to each reduce task is guaranteed to be processed in increasing key order. It is possible to provide a user-specified comparison function to be used during the sort process.

6. Output writer: The output writer is responsible for writing the output to stable

storage. In the basic case, this is to a file, however, the function can be modified so that data can be stored in, e.g., a database.

As can be noted, for a particular job, only a Map function is strictly needed, although for most jobs a Reduce function is also used. The need for providing an Input reader and Output writer depends on data source and destination, while the need for Combiner and Partition functions depends on data distribution.

Hadoop [50] is an open-source implementation of MapReduce, and without doubt, the most popular MapReduce variant currently in use in an increasing number of prominent companies with large user bases, including companies such as Yahoo! and Facebook.

Hadoop consists of two main parts: the Hadoop distributed file system (HDFS) and MapReduce for distributed processing. As illustrated in Figure 2.2, Hadoop consists of a number of different daemons/servers: NameNode, DataNode, and Secondary NameNode for managing HDFS, and JobTracker and TaskTracker for performing MapReduce.

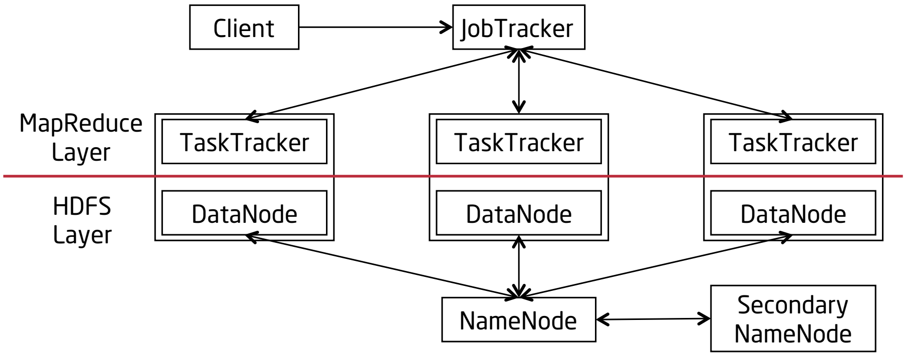


**Figure 2.2:** Hadoop Architecture

# Part I

# Outlier Detection Over Data Streams

# 3

# A Generic Outlier Detection Framework

We now introduce our scalable framework called *LEAP*, capable of continuously processing distance-based outliers with low CPU and memory resource utilization. LEAP is built on two fundamental optimization principles namely *minimal probing* and *lifespan-aware prioritization* as described below.

## 3.1 Theoretical Foundation

In all distance-based outlier definitions, 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 eliminating inliers from it. In fact, initially, each point $p_i$ in the dataset is a *potential outlier candidate*, until one has acquired enough evidence to show that $p_i$ is an inlier. For example, in the process of identifying $O_{thres}^{(k,R)}$ outliers, until finding that $p_i$ has at least k neighbors and thus qualifies as inlier, $p_i$ cannot be safely removed from the *outlier candidate set*.

This fact leads us to an important observation. That is, to identify whether a point $p_i$ is a distance-based outlier in a dataset $D$, one may not need the distance between $p_i$ to *every* other point in $D$. Instead, a potentially small subset of points will be sufficient to prove that $p_i$ is an inlier. Also due to the rarity of outliers, the majority of points in the dataset could be labeled as inliers in this way by collecting only a small amount of information. To describe the least amount of information needed to prove $p_i$'s inlier status we define the concept of **Minimal Evidence Set for Inlier** (*MESI*).

**Definition 3.1** *Given an outlier query and a dataset $D$, the* ***MESI*** *set for a data point $p_i \in D$ is a dataset $M$ such that $M \subseteq D$, if the distance set $DistSet(M, p_i)$ = $\{d(p_1, p_i)$, $d(p_2, p_i), ..., d(p_n, p_i)|\ p_{j(1 \leq j \leq n)} \in M\}$ is sufficient to label $p_i$ as an inlier, and there does not exist any $M' \subseteq D$ such that $|M'| < |M|$ and $DistSet(M', p_i)$ = $\{d(p_1, p_i)$, $d(p_2, p_i)$, ..., $d(p_m, p_i)|p_{j(1 \leq j \leq m)} \in M'\}$ is sufficient to label $p_i$ as an inlier.*

The size of *MESI* for a point $p_i$ is usually much smaller than the size of $p_i$'s complete neighborhood. For example, for $O^{(k,R)}_{thres}$ outlier, the *MESI* for any point $p_i$ is composed of *any* $k$ points that are within $R$ distance from $p_i$. Thus its size is $k$. In general, this input parameter $k$ is much smaller than the average number of neighbors each point may have in $R$ distance range. Otherwise the outliers detected with fewer than $k$ neighbors would not considered to be *abnormal phenomena* in the dataset. The cardinality of *MESI* for a point $p_i$ in the $k$NN outlier definitions is also bounded by a constant value $k$ as we will show in Chapter 5. This observation guides us to propose the **Minimal Probing** optimization principle (Sec. 3.2).

Although *MESI* is sufficient to prove a point's inlier status in the current window, unlike in static environments, locating more neighbors beyond *MESI* for a given point may be beneficial in streaming environments. These additional neighbors may help us to determine the status of this point in future windows. Thus, we now extend the concept of *MESI* in a static dataset to *MESI* in a sequence of stream windows. In particular, we define the concept of **Minimal Evidence Set for Inlier in a Window Sequence** as below.

**Definition 3.2** *Given a streaming outlier detection query $Q$ and all points in the current window $W_c$, denoted by $D_{W_c}$, $MESI_{(W_{c,c+x})}$ for $p_i$ in a window sequence from $W_c$ to $W_{c+x}$, is a dataset $M$ with $M \subseteq D_{W_c}$, if the distance set $DistSet(M, p_i)$=$\{d(p_1, p_i)$, $d(p_2, p_i)$, ..., $d(p_n, p_i)|p_{j(1 \leq j \leq n)} \in M\}$ is sufficient to label $p_i$ as an inlier in windows $W_c$ to $W_{c+x}$, and there does not exist any $M' \subseteq D_{W_c}$ with $|M'| < |M|$ and $DistSet(M', p_i)$ = $\{d(p_1, p_i)$, $d(p_2, p_i), ..., d(p_m, p_i)|p_{j(1 \leq j \leq m)} \in M'\}$ is sufficient to label $p_i$ as an inlier in windows $W_c$ to $W_{c+x}$.*

In other words, the $MESI_{(W_{c,c+x})}$ for a point $p_i$ is a minimal subset of the current window population $D_{W_c}$ that provides sufficient evidence to prove that $p_i$ is an inlier in windows $W_c$ to $W_{c+x}$, regardless of the characteristics of the future incoming stream. This is possible because by analyzing the time stamp of a point $p_i$ and the query window (the slide and window sizes), we can determine the number of windows that $p_i$ will survive in. For example, for a point $p_i$ that just arrived with the latest slide in the current window $W_c$, if we found $k$ points within $R$ distance from $p_i$ that arrived when $p_i$ did, then these $k$ points form $MESI_{(W_{c,c+x})}$ for $p_i$, where $W_{c+x}$ is the last window in which $p_i$ will be alive. This is because these points will be accompanying $p_i$ as its neighbors until $p_i$ expires. We are now ready to define the concept of **Life Time Minimal Evidence Set for Inlier**.

**Definition 3.3** *$MESI_{(W_{c,c+x})}$ for $p_i$ is a **life time MESI** of $p_i$, denoted as $\boldsymbol{MESI_{lt}}$, if $W_{c+x}$ is the last window in which $p_i$ participates before its expiration.*

A $MESI_{lt}$ for $p_i$ is an ideal evidence set because it proves the inlier identity of $p_i$ during its entire remaining life, hence named *safe inlier*. It eliminates the need for any future maintenance effort on $p_i$ for the potential detection of its outlier status. Acquiring the $MESI_{lt}$ with *minimal CPU costs* is the key objective for outlier detection in streaming windows. This insight inspires us to propose the **Lifespan-Aware Prioritization** optimization principle in Sec. 3.3.

## 3.2 Minimal Probing Principle

As elaborated in Chapter 7, all state-of-the-art techniques [26, 27, 48] rely on complete neighborhood searches to identify outliers. In this work, we abandon this methodology and instead present an optimization principle referred to as *minimal probing*. The key idea is that we no longer conduct complete neighborhood searches, such as range query searches, but instead use a lightweight operation called *probing*.

**Definition 3.4** *Given a point $p_i$ in the current window $W_c$, **probing** is an operation that evaluates the distance between $p_i$ and other points in $W_c$ until either the MESI for $p_i$ in $W_c$ is acquired or $p_i$'s entire neighborhood has been evaluated.*

The goal of probing for a point $p_i$ is the discovery of a *MESI* for $p_i$ in the current window rather than its complete neighbor set. Therefore probing is fundamentally more efficient compared to a complete neighborhood search, as it significantly reduces the number of data points that need to be evaluated.

Furthermore, the minimal probing principle guides us to intelligently use this lightweight *probing operation* so to maximize the system resource savings. The idea is to carefully extract and then to organize the evidence gathered during each probing process, and furthermore to reuse it whenever possible to avoid repeated probing process.

For all three outlier definitions, with the probing only applied in two situations as explained below we can guarantee the correctness of the query. First, each *new* point $p_i$ that just arrived in the query window needs a probing to figure out its status in the current window. Second, an existing point $p_i$ *without a valid MESI* in the new window needs a probing to re-evaluate its status.

In the first situation, for a newly arriving point $p_i$ the probing operation has to be conducted *from scratch* to search for the needed evidence of $p_i$.

However this is not the case in the second situation. For a point $p_i$ two conditions can lead to the absence of its *MESI*. First, $p_i$ had been classified as an outlier in the previous window. Therefore no *MESI* has so far been acquired. Second, $p_i$ lost its prior *MESI* when the stream slides to the current window $W_c$ and expired points are removed from $W_c$. In both cases, the known MESI evidence about $p_i$ which survived the stream data expiration can still contribute to simplify this probing operation. Rather than searching for a new *MESI* from scratch, the probing operation instead only acquires *enough new evidence* to prepare the *MESI* for $p_i$ for the window $W_c$.

Therefore although the goal for probing is to acquire *MESI* for $p_i$ in the current window, the collected evidence provides us with much richer information than just proving $p_i$'s current status. The method of organizing the MESI to facilitate the fully reuse of the evidence gathered by probing is discussed in Sec. 3.4.

As conclusion, the minimal probing principle uses a lightweight probing process to replace the expensive complete neighbor search. It guides us to fully exploit all evidence gathered during the probing process and thus to minimize the costs of each probing process.

## 3.3 Lifespan-Aware Prioritization Principle

Next we propose our second optimization principle termed
*Lifespan-Aware Prioritization*. By utilizing the lifespan information of data points this principle further optimizes the probing operation to always discover the *best MESI*.

**Lifespan of MESI.** As mentioned in Sec. 3.1, the *MESI* of a point $p_i$ in the current window as a whole may serve as the *MESI* of $p_i$ in a sequence of future windows. The number of windows in which a *MESI* can survive, termed the **lifespan of MESI**, relies on how many windows each point $p_j$ in this *MESI* can survive, also termed the **lifespan** of point $p_j$. In the sliding window scenario, the lifespan of a point $p_i$ can be determined as follows.

**Lemma 3.1** *Given the slide size $Q.slide$ of a query $Q$ and the starting time of the current window $W_c.T_{start}$, the **lifespan** $p_i.life$ of a data point $p_i$ in $W_c$ with time stamp $p_i.ts$ is calculated by $p_i.life = \lceil \frac{p_i.ts - W_c.T_{start}}{Q.slide} \rceil$*[1]*, indicating that $p_i$ will participate in windows $W_c$ to $W_{c+p_i.life-1}$.*

Hence given Lemma 3.1 the lifespan of a *MESI* can be decided as below.

**Lemma 3.2** *Given a MESI of $p_i$ in the current window $W_c$ denoted as $MESI(p_i)$, the **lifespan of MESI**$(p_i)$ $MESI(p_i).life = min\{p_j.life \mid p_j \in MESI(p_i)\}$.*

By Def. 3.2 $MESI(p_i)$ is a $MESI_{(W_{c,c+MESI(p_i).life-1})}$ of $p_i$ covering the window sequence from $W_c$ to $W_{c,c+MESI(p_i).life-1}$. As introduced in Sec. 3.2, among the existing points in window $W_c$ only those without their *MESI* covering the new window $W_{c+1}$ must conduct probing to re-evaluate their status. Therefore, the longer a window sequence a *MESI* covers, the fewer probing processes are needed for this point. Naturally the *MESI* with largest lifespan will be the *best MESI*. Henceforth quickly deriving the best *MESI* of each point is critical for minimizing the probing frequency and in turn saving CPU resources.

Next we analyze how we can further optimize our probing process to always acquire the best *MESI*, but without sacrificing its efficiency. On the one hand, the probing process

[1] For count-based windows, $p_i.ts$ and $W_c.T_{start}$ are sequence numbers indicating the arrival positions of data points in a stream.

for $p_i$ should acquire the best *MESI* of $p_i$. On the other hand, we want the probing process must stay lightweight, so that it stops immediately once it has gotten the *MESI* of $p_i$ in the current window. Our solution is to leverage the lifespan theory of *MESI* in Lemma 3.2 to prioritize the order in which the probing operation processes the data points.

**Definition 3.5** ***Lifespan-aware Prioritization***: *During the probing process of $p_i$, if two data points $p_j$ and $p_k$ have the same probability to be in the MESI of $p_i$ for the current window, we always evaluate $p_j$ first, if $p_j.life > p_k.life$.*

Since the succeeding points $p_j$ that arrived after $p_i$ do not expire earlier than $p_i$, their influence will persist during the entire life of $p_i$. Therefore any such $p_j$ contributes equally to $p_i$ in terms of determining $p_i$'s outlier status, although they may have different lifespans. Therefore we can treat all succeeding points of $p_i$ as if they all had the same lifespan, namely a lifespan larger than $p_i$'s.

## 3.4 Lifespan-Aware Probing Operation

The above *lifespan-aware prioritization* principle together with the *minimal probing* notion implies an optimized probing operation termed *LifEspan-Aware Probing operation* or *LEAP*. LEAP represents the core operation of our framework.

**Definition 3.6** *Assume window $W_c$ is composed of k slides denoted as $S_i$, (1 $\leq$ i $\leq$ k). $S_i$ arrives earlier than $S_{i+1}$. Given a point $p_i$ in $W_c$, **LEAP** is a probing that evaluates the status of $p_i$ by testing other points in the $S_k$, $S_{k-1}$, ... order.*

Intuitively we can see that *LEAP* is guaranteed to produce the *best MESI*. In sliding window streams the data points are naturally ordered by their arrival time and expire in a predictable order. Hence the lifespan of any point can be precisely calculated. By Lemma

3.1, points in a particular slide share the same lifespan, while points in different slides have distinct lifespans. Later arriving slides have longer lasting lifespans. By conducting the search with a later arriving slide first order, the points with a larger lifespan will always be tested first. Therefore given a point $p_i$, *LEAP* will produce a *MESI* composed of the evidence with the largest lifespan, that is the *best MESI*. Furthermore LEAP stops immediately as soon as a *MESI* is acquired. Thus it is as lightweight as an ordinary probing operation.

The information collected in the probing process of $p_i$ needs to be carefully selected and kept to minimize the costs of the future probing for $p_i$ (Sec. 3.2). The information shown to be valuable and termed *potential evidence*, is organized as a general *lifespan-aware evidence* structure denoted as $p_i.evi[\,]$.

**Definition 3.7** *The* ***lifespan-aware evidence*** *for a data point $p_i$ ($p_i.evi[\,]$) represents an ordered list of potential evidence of $p_i$ in the current window $W_c$ with each entry of $p_i.evi[\,]$ corresponding to a set of data points with the same lifespan, where the ordering is determined by the lifespan.*

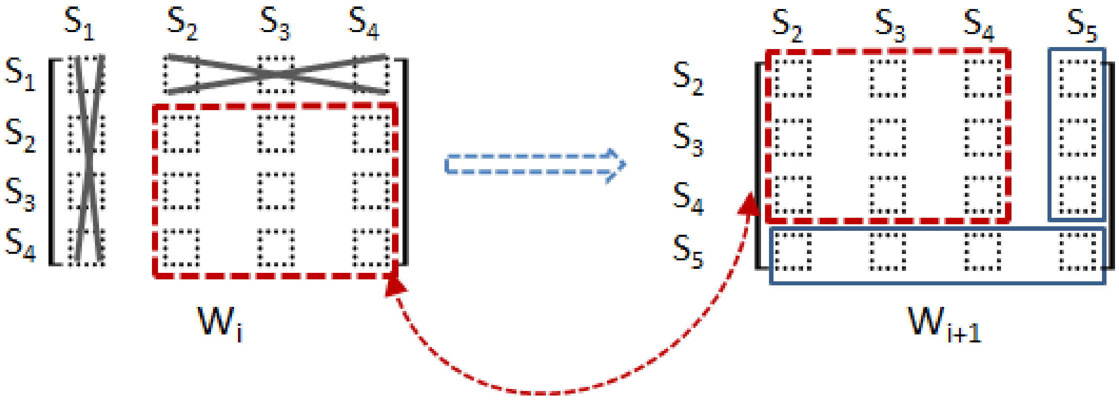


**Figure 3.1:** LEAP: Sharing of the lifetime proximity measure

As shown in Fig.3.1 the storage of the $evi[\,]$ structure of a particular window $W_i$ with 4 slides can be abstracted as a two dimensional matrix $M_i$. The element $M_i[S_x][S_y]$

represents a linear data structure which contains the $S_y$th entries of all points in slide $S_x$. This abstract structure explicitly illustrates that our lifespan-aware evidence infrastructure is extremely conclusive to handle the stream evolution. When the window slides from $W_i$ to $W_{i+1}$, by moving the elements bounded in the dash rectangles one unit up to the top left corner of $M_i$, it can be easily transformed into $M_{i+1}$ of $W_{i+1}$ only by having to conduct the computation for the elements within the new slide $S_5$.

**Space Complexity Analysis.** The storage of the $evi[\,]$ structure has a worst case space requirement $O(nr)$ with *r* as the ratio of the *Q.win* over *Q.slide* and *n* as the number of unsafe inliers and outliers. In fact this structure can be further compressed to its half size due to the observation that $p_i$'s succeeding neighbors contribute equally to $p_i$ in terms of determining $p_i$'s status, even if they have different lifespan (as stated in Section 3.3). Therefore the entries representing its succeeding neighbors can be merged with the final number of entries at most being equal to its lifespan.

The precise data structure specific to each outlier type will be introduced in Chapters 4 and 5.

## 3.5 Optimality of LEAP

The LEAP operation, when continuously applied to determine the outlier status of a data point $p_i$ until its expiration, is shown to be optimal in CPU resources consumed for all three outlier definitions.

**Theorem 3.1** *Given a point $p_i$ in current window $W_c$ and function f($p_i$,$W_c$, $Pr_s$) indicating the CPU costs required by a search strategy $Pr_s$ to evaluate the outlier status of $p_i$. Then*
$\sum_{j=c}^{c+life-1} f(p_i, W_j, LEAP) \leq \sum_{j=c}^{c+life-1} f(p_i, W_j, Pr_s)$ *with life denoting the lifespan of* $p_i$.

**Proof:** We first establish a prerequisite. Given a data point $p_i$ LEAP takes the same CPU cost to acquire a member of *MESI* for $p_i$ as any other search strategy $Pr_s$ takes. We denote this cost as $\boldsymbol{C^m}$. This prerequisite is justified as follows.

First, given a stream *S* with an unknown distribution, then each point in $W_c$ has the equal chance to be in the *MESI* of $p_i$. Thus in average any $Pr_s$ will test the same number of points, hence the same costs to acquire a member of *MESI* for $p_i$.

Second, LEAP is orthogonal to indexing. The both optimization principles of LEAP aim to minimize the frequency of neighbor searches, while indexing instead focuses on accelerating the search of each single neighbor by reducing the neighbor search space. Therefore LEAP is able to exploit whatever indexing methods ever invented or possibly coming up with in the future.

Then we prove Theorem 3.1 using Math Induction.

(1) First we prove $\sum_{j=c}^{c} f(p_i,\, W_j, LEAP) \leq \sum_{j=c}^{c} f(p_i,\, W_j, Pr_s)$.

LEAP immediately stops once it acquires the complete MESI for $p_i$. We use $\mid MESI(p_i) \mid$ to denote the cardinality of MESI. Hence f($p_i$,$W_j$, $LEAP$) = $\mid MESI(p_i) \mid$ * $C^m$. For any other probing strategy $Pr_s$ the cost $f(p_i,\, W_j, Pr_s)$ = $x$ * $C^m$. By Def. 3.1, MESI is the minimal information needed to prove $p_i$'s status. Hence $\mid MESI(p_i) \mid \leq x$. Therefore

$$\begin{aligned}\sum_{j=c}^{c} f(p_i, W_j, LEAP) = f(p_i, W_c, LEAP) =\mid MESI(p_i) \mid *C^m \\ \leq x * C^m = f(p_i, W_c, Pr_s) = \sum_{j=c}^{c} f(p_i, W_j, Pr_s).\end{aligned} \tag{3.1}$$

(2) Then our induction step from n to n+1 is:

$$\begin{aligned}if \sum_{j=c}^{n} f(p_i, W_j, LEAP) \leq \sum_{j=c}^{n} f(p_i, W_j, Pr_s),\, then \\ \sum_{j=c}^{n+1} f(p_i, W_j, LEAP) \leq \sum_{j=c}^{n+1} f(p_i, W_j, Pr_s)\, with\, n < c + life - 2\end{aligned} \tag{3.2}$$

Given the costs $\boldsymbol{C_s}$ LEAP reaps in savings to process $p_i$ from $W_c$ through $W_n$ compared to $Pr_s$, we can prove Eq. 3.2 as follows. When the stream slides from $W_n$ to $W_{n+1}$, the costs LEAP takes to ensure the status of $p_i$ are guaranteed to be not $C_s$ larger than the costs $Pr_s$ takes.

LEAP will be more expensive than $Pr_s$ only if more elements expire in $L^m(p_i)$ (the MESI for $p_i$ produced by LEAP) than in $Pr_s^m(p_i)$ (the evidence produced by $Pr_s$). Suppose *r* more elements expire in $L^m(p_i)$ than in $Pr_s^m(p_i)$. This means that in $Pr_s^m(p_i)$ of $W_n$, there are at least *r* members younger than the oldest member of $L^m(p_i)$. However in the first window $W_c$, the oldest member of $Pr_s^m(p_i)$ is at least as old as the oldest member of $L^m(p_i)$. To achieve this, $Pr_s$ must have acquired *at least r more* MESI members than *LEAP*, because LEAP always tests the points with larger lifespans first. However to re-establish the MESI of $p_i$ in $W_{n+1}$, LEAP only has to acquire *exactly r* more MESI members than $Pr_s$.

(3) By steps (1) and (2), Theorem 3.1 is proven. ■

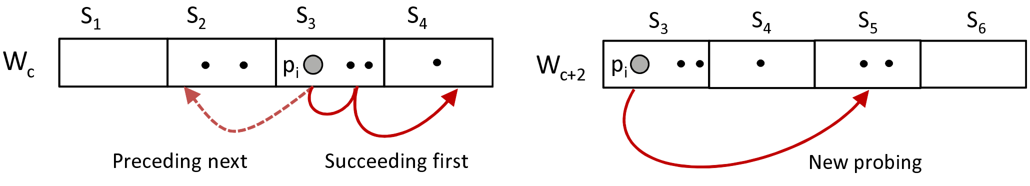
# 4

# Strategies for Distance-Threshold Outliers

We now apply our framework to distance-threshold outliers.

**MESI and Lifespan-Aware Evidence.** By Def. 2.1 once acquiring *k* neighbors, a point $p_i$ can be safely declared as an inlier. Therefore the *MESI* for $p_i$ is a data set that contains exactly k neighbors. As the window slides, all other data points examined so far besides its unexpired *MESI* members have no chance to ever be in the *MESIs* of $p_i$. Therefore only keeping the k neighbors in the MESI is sufficient to avoid any distance recomputation for $p_i$. Furthermore to determine the status of $p_i$, we only need the number of its neighbors rather than who its *exact* neighbors are. Therefore the *Lifespan-Aware Evidence* structure of $p_i$ ($p_i.evi[\,]$) for distance-threshold outliers is simply a list of counts, each list entry corresponding to the number of MESI members (neighbors) of $p_i$ in a particular slide.

**Thresh_LEAP.** Based on the above MESI and *Lifespan-Aware Evidence* structure we present a customized algorithm *Thresh_LEAP* (Alg. 1) for $O_{thres}^{(k,R)}$ outlier detection. When a new window $W_c$ arrives, Thresh_LEAP starts by evaluating each new arrival $p_i$ that had not been in $W_{c-1}$ by simply calling the LEAP operation (Alg. 2). Here we explain step by step using an example how *LEAP* works.



(a) LEAP: $W_c$ (b) LEAP: $W_{c+2}$

**Algorithm 1** Thresh_LEAP($W_c$)

```
1: for each $p_i \in W_c.S^{new}$ do
2:     LEAP($p_i$,$W_c$);
3: for each $p_i \in W_c.S^{exp}$.triggered do
4:     expireEvidence($p_i$);
5:     LEAP($p_i$, $p_i$.skippedPoints($W_c$));
```

**Algorithm 2** LEAP($p_i$,$W_c$)

```
Input: Data point $p_i$, Dataset $W_c$ //Data points in the current window
Output: Bool isOutlier //Outlier status of $p_i$
1: Bool IsOutlier = false;
2: if (NULL == $p_i$.evi[ ]) then
3:     buildSuccEvidence($p_i$);
4: for each q $\in$ $p_i$.succPoint($W_c$) do
5:     if (true == $p_i$.isInNeighborhood(q)) then
6:         $p_i$.updateSuccEvidence();
7:         if (true == $p_i$.isMESIAcquired()) then
8:             $p_i$.isSafe = true;
9:             return isOutlier;
10: while $p_i$.precSlides $\neq$ NULL do
11:     slide = getSlideWithLargestLifespan($p_i$.precSlides($W_c$));
12:     $p_i$.buildPrecEvidence(slide);
13:     for each q $\in$ slide do
14:         if (true == $p_i$.isInNeighborhood(q)) then
15:             $p_i$.updatePrecEvidence(slide);
16:             if (true == $p_i$.isMESIAcquired()) then
17:                 slide.updateTriggeredList($p_i$);
18:                 return isOutlier;
19: isOutlier = true;
20: return isOutlier;
```

**Example 4.1** *We use an example query $Q$ with $k$ = 5 and a fixed $R$ with the ratio of $Q.win$ over $Q.slide$ as 4 to explain how LEAP handles the new data points. As shown in Fig.4.1(a), window $W_c$ is divided into four slides. Given a new data point $p_i$ LEAP first tests its succeeding data points (Line 5). At the same time the first entry of $p_i.evi[\,]$ is* *established as ($S_{succ}$:0) which represents the number of $p_i$'s succeeding neighbors (Line 3). Once a neighbor is acquired, we update the succeeding entry of $p_i.evi[\,]$ (Line 7), and check whether its MESI has been achieved (Line 8). By testing all its succeeding data points in this window, $p_i$ finds three neighbors. However, it still did not acquire its MESI. Then it has to turn back and proceed to probe its preceding slides (Line 14). The slide with the largest lifespan is tested first (Line 15). In this case it is $S_2$. Correspondingly a new entry ($S_2$:0) is created and appended to $p_i.evi[\,]$ (Line 16). The search is terminated after $p_i$ gets its fifth neighbor which completes the MESI for $p_i$ (Line 20). $p_i$ is labeled as **unsafe inlier**. $S_2$ is being remembered as the **triggering slide** of $p_i$, meaning that the expiration of $S_2$ might lead to a status transformation of $p_i$. To indicate this check $p_i$ is inserted into the triggered outlier candidate list of $S_2$, namely $S_2.triggered$ (Line 21). The $p_i.evi[\,]$ at this point is $< (S_2 : 2), (S_{succ} : 3) >$.*

After the new arrivals have been all processed, Thresh_LEAP proceeds to process the unexpired points from $W_{c-1}$ that remain in $W_c$. Clearly the $evi[\,]$ has already been previously established for them. However not all unexpired points need to be re-evaluated. As shown in Alg.1 (Line 4), only the points in $S^{exp}.triggered$ list are re-examined by the LEAP operation with $S^{exp}$ denoting the most recently expired slide. For example, when the stream evolves from $W_c$ to $W_{c+1}$ the expiration of $S_1$ would not trigger the examination of $p_i$, because $p_i$ is not in $S_1.triggered$. Only the departure of $S_2$ will trigger the process of checking the status migration of $p_i$ (Fig. 4.1(b)).

**Example 4.2** *We still use $p_i$ of Example 4.1 to explain the above re-evaluation procedure. For $W_{c+2}$, Thresh_LEAP first updates the $p_i.evi[\,]$ to ($S_{succ}$:3) by removing entry ($S_2$:2) (Line 5, Alg. 1). Then the LEAP operation is activated again on the new slide $S_5$ which was skipped while $S_1$ expired (Line 6, Alg. 1). The MESI is filled up again after finding the two newly arriving neighbors in $S_5$. Its $p_i.evi[\,]$ is updated to ($S_{succ}$:5). Now $p_i$ has* *five MESI members which did not arrive earlier than $p_i$. Therefore $p_i$ achieves its life time MESI $MESI_{lt}$. Now $p_i$ is guaranteed to never become an outlier again and thus is marked as safe inlier (Line 9, Alg. 2). Thus at this point the $p_i.evi[\,]$ can be safely purged altogether.*

As shown in this example it is extremely efficient to determine the status of $p_i$ with the assistance of $p_i.evi[\,]$ structure. When the window slides from $W_{c+1}$ to $W_{c+2}$, its leftmost most side $S_2$ entry will be pruned from $p_i.evi[\,]$. Then by summing up the alive entries (in this case this would be only one entry $S_{succ}$), the LEAP operation continues to be aware of the current status of $p_i$. To acquire the new status of $p_i$, it proceeds to test the new data points from $S_5$ until $p_i$'s *MESI* is again established.

# 5

# Strategies for $k$NN Outliers

**MESI for $k$NN Outliers.** We now demonstrate how we apply our framework to detect $k$NN outliers. We use Alg.3 to introduce the *MESI* for $k$NN outliers. By Def. 2.2 of $O^{(k,n)}_{kmax}$ outliers, Alg. 3 outputs the top-n outliers in a window $W_c$. Such a set called *outliersSet* in Line 1 is maintained during the search process. Let $D^{kmax}_{min}$ be the shortest distance between any data point in *outliersSet* seen so far and its $k$th nearest neighbor (Line 2). Assume that for a given point $p_i$ we are processing its distance to its kth-nearest neighbor ($D^{kmax}(p_i)$) (Lines 4 to 6). Since $D^{kmax}(p_i)$ monotonically decreases as we process more points, the current value is an upper-bound on its eventual value. If the current value becomes smaller than $D^{kmax}_{min}$, then $p_i$ cannot be an outlier (Lines 7 to 9). Therefore the *MESI* for $p_i$ is acquired, which is its $k$NN in the data points seen so far (neighbors($p_i$)). These points are so-called the temporary $k$NN of $p_i$.

If $D^{kmax}(p_i)$ is larger than the cutoff threshold $D^{kmax}_{min}$, $p_i$ will be an outlier candidate. Both the *outliersSet* and $D^{kmax}_{min}$ will be updated (Lines 12-16). As more points are processed, more extreme outliers will be found. The top-n outliers will be finalized after all data points have been processed.

By replacing the $D^{kmax}(p_i)$ with $D^{kavg}(p_i)$ and $D^{kmax}_{min}$ with $D^{kavg}_{min}$ the same rule can

**Algorithm 3** $k$NN_MESI($W_c$)

```
outliersSet = ∅; //the top-n outliers set
$D^{kmax}_{min}$ = 0;
for each $p_i \in W_c$ do
    for each $p_j \in W_c$ - $p_i$ do
        neighbors($p_i$) = nearest($p_i$,neighbors($p_i$) + $p_j$, k);
        $D^{kmax}(p_i)$ = maxDist($p_i$, neighbors($p_i$));
        if ((| $neighbors(p_i)$ | == k) $\wedge$ ($D^{kmax}(p_i) \leq D^{kmax}_{min}$)) then
            $p_i$.outlierCandidate = false;
            break;
    if (false $\neq$ $p_i$.outlierCandidate) then
        outliersSet = topOutliers(outliers + $p_i$, n);
        if (| $outliersSet$ | == n) then
            $D^{kmax}_{min}$ = min($D^{kmax}(p_i)$ — $\forall$ $p_i$ in neighbors);
```

be applied to $O^{(k,n)}_{kavg}$ outlier.

**Lifespan-Aware Evidence.** Unlike the $O^{(k,R)}_{thres}$ outlier for $k$NN outliers, only recording *MESI* of $p_i$ is not sufficient for avoiding the distance re-computation whenever the status evaluation is triggered. The *non-MESI* points could also contribute to the *MESIs* of future windows. For example, given a point $p_i$ whose *MESI* members all expire, if no arriving points are close enough to $p_i$, the *MESI* of $p_i$ in the next window must be formed based on the points which have been evaluated before but were not yet part of the *MESI* of $p_i$. In this case to avoid re-computation we would have to keep more information besides the *MESI* of $p_i$ in the current window. However keeping all pre-computed distances is not practical.

Fortunately this is where our insight comes to the rescue. Namely keeping the $k$NN corresponding to each unexpired slide (or the temporary $k$NN for a slide not completely evaluated) is sufficient to avoid re-computation. The global $k$NN with respect to the unexpired data points seen so far is guaranteed to be in the union of these local $k$NN sets. That is, this global $k$NN can be easily derived by merging and sorting the local $k$NN sets. We need to evaluate the distance between $p_i$ and new arrivals only if the $k$NN distance of this global $k$NN is still larger than the cutoff threshold. Otherwise this global $k$NN will remain to be the *MESI* of $p_i$ in the new window. In short, with this structure the distance re-computation is completely eliminated. Therefore for $k$NN outliers $p_i.evi[\,]$ is a list of data points (along with their distances to $p_i$) sets, each corresponding to its $k$NN in each unexpired slide. Since $p_i.evi[\,]$ is compact, keeping it for each point does not introduce prohibitive memory overload.

**Algorithm 4** $k$NN_LEAP($W_c$)

```
1:  $D_{min}^{kmax}$ = 0; //Reset cutoff threshold
2:  outliersSet = ∅; //Reset the top-n outliers set
3:  for each $p_i \in W_c$.unExpired do
4:      if (false == $p_i$.isSafe) then
5:          expireEvidence($p_i$);
6:      if (true == isStillInlier($p_i$)) then
7:          continue;
8:      else
9:          LEAP($p_i$,$p_i$.skippedPoints($W_c$));
10: for each $p_i \in W_c$.newArrival do
11:     LEAP($p_i$,$W_c$);
```

**LEAP Operation for $k$NN Outliers.** Given a point $p_i$, LEAP first probes the points with larger lifespan. An entry of $p_i.evi[\,]$ is established to represent the $k$NN in its succeeding points. If its *MESI* is not acquired by considering its succeeding points, then the search will need to proceed by processing the preceding points in decreasing order with respect to their lifespans. During this process an entry is created for each preceding slide. LEAP continues to evaluate the distance between $p_i$ and other data points in $W_c$ until either its temporary $k$NN distance is smaller than the cutoff threshold (*MESI* is acquired) or all points of $W_c$ have been tested. Only in the latter case, the probing operation will return the traditional full $k$NN of $p_i$. In this case both the top-n outliers set *outliersSet* and the cutoff threshold will be updated. Due to space restriction, the pseudo code is omitted here.

**$k$NN Outliers Detection With LEAP ($k$NN_LEAP).** Alg. 4 shows how LEAP is utilized to detect $k$NN outliers in a window $W_c$. $k$NN_LEAP first resets the top-n outlier candidates set and the cutoff threshold (Lines 1 to 2). Then it starts processing the unexpired data points, namely the points that were already in window $W_{c-1}$ (Line 3). Given a point $p_i$, $k$NN_LEAP first purges the expired entry of $p_i.evi[\,]$. Then it re-calculates its temporary $k$NN (Lines 4, 5) if its MESI consists of expired data points (unsafe status).

If its current $k$NN distance (either the previous distance for a safe point or the newly established one for an unsafe point) is larger than the cutoff threshold, the LEAP operation for $p_i$ will be triggered again on the points skipped last time (Lines 7 to 11). Then $k$NN_LEAP proceeds to process the new arriving data points with the LEAP operation (Lines 13 to 15) until all new arrivals are evaluated.

# 6

# Experimental Evaluation

## 6.1 Experimental Setup & Methodologies

All algorithms are implemented on the HP CHAOS Stream Engine [51]. Experiments are performed on a PC with 3.0G Hz CPU and 4GB memory, which runs Windows 7 OS.

**Real Datasets.** We use two real streaming datasets. The Stock Trading Traces dataset (STT) [52] has one million transaction records throughout the trading hours of a day. The high dimensional Forest Cover (FC) dataset available at the UCI KDD Archive (url:kdd.ics.uci.edu) also used by [27], contains 581,012 records with 54 quantitative attributes.

**Synthetic Datasets.** We deploy a data generator to produce streams with a controlled number of outliers and data distribution types. Those datasets contain Gaussian distributed data points as inlier candidates with uniform distributed noise. Both the Gaussian distributed points and noise are randomly distributed in each segment of the stream.

**Metrics.** We measure two metrics common for stream systems, namely CPU time and peak memory consumption. Each experiment evaluates 10,000 windows. Both metrics are averaged over all windows. Although the experiments are reported using count-based windows, time-based windows provide similar results.

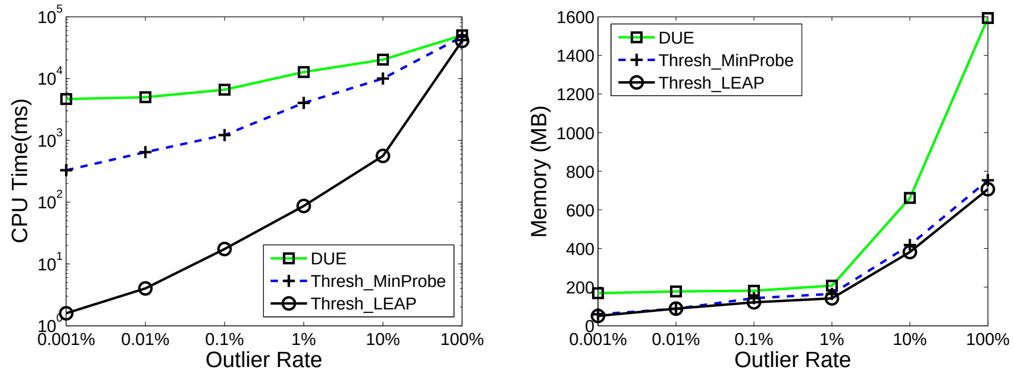
**Alternative Algorithms.** Our experiments focus on evaluating the effectiveness of our both optimization principles, namely *minimal probing* and *lifespan-aware prioritization*, in detecting distance based outliers. For distance-threshold outliers, we compare our algorithms Thresh_MinProbe and Thresh_LEAP (Chapter 4) against the state-of-the-art method DUE [27] as introduced in Chapter 7. [1] The Thresh_MinProbe applies only the first principle. It essentially equals to DUE enhanced with *minimal probing*. Thresh_LEAP instead utilizes the LEAP framework which applies both principles. For $k$NN outliers, no existing algorithms in the literature tackle this type of outlier in the streaming context. Hence, we compare our $k$NN_MinProbe and $k$NN_LEAP algorithms against $k$NN_BASIC which applies the static Orca algorithm [53] to compute the top-$n$ outliers from scratch for each window. Similar to distance-threshold outliers, $k$NN_MinProbe applies only Minimal Probing principle, while $k$NN_LEAP applies both. Since these three methods all experience only a slight difference for $O^{(k,n)}_{kmax}$ and $O^{(k,n)}_{kavg}$ outlier types, for space reasons, we present the results for $O^{(k,n)}_{kmax}$ outliers only. To evaluate the effect of indexing, similar to [54] we implement a hash-based grid index augmented with a time-aware mechanism for efficiently evicting expiring data. We carefully tune the granularity of the cells and equally apply the same best setting to all compared algorithms. We denote each algorithm xx augmented by this index by "xx-Index".

**Methodology.** We evaluate the performance of the proposed methods by varying the most important parameters. Specifically, our experiments cover the three major cost factors, namely stream velocity, volume, and outlier rate. We vary the velocity of a data stream by varying the slide size from 0.5k to 50k while leaving all other settings constant.[2]

[1] In this work we chose to compare against DUE rather than the more sophisticated MCOD algorithm of [27], because in the experiments of [27], MCOD does not show clear advantage over DUE in most of the cases.

[2] Here we only present the results for distance-threshold outliers, since $k$NN outliers are confirmed to be not sensitive to slide size.

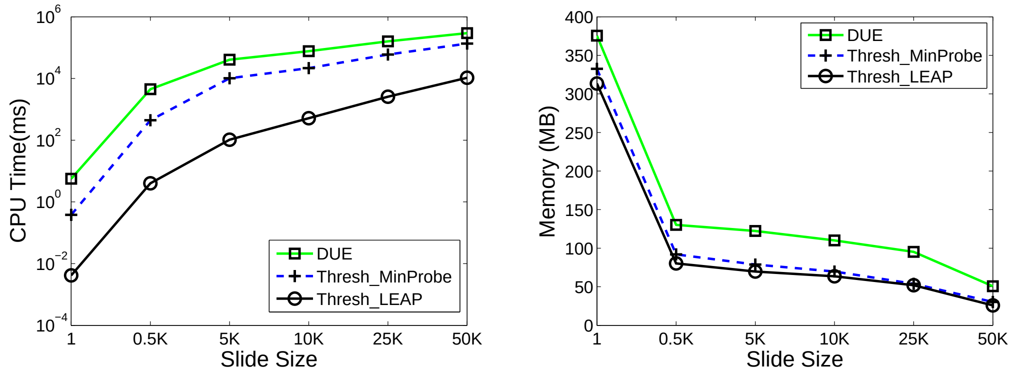
We also measure scalability on high volume streams by varying the window size $\omega$ from 1k to 200k. Similarly, we measure how well these methods work for different outlier rates. For the distance-threshold type, this means varying $R$, while for $k$NN types varying $n$ as defined in Sec. 2.1. Both control outliers from being rare (0.001%) to being common in the dataset (100%). Since this change also affects the density of neighboring area for each data point, this experiment also reflects data distribution variation in essence. We also measure the scalability of our approach over data dimensionality by varying dimensions from 2 to 40.



(c) CPU (log scale)

(d) Memory (linear scale)

**Figure 6.1:** LEAP: Varying Outlier Rates on Synthetic Dataset



(a) CPU (log scale)

(b) Memory (linear scale)

**Figure 6.2:** LEAP: Varying Slide Sizes on Synthetic Dataset

## 6.2 Evaluating Distance-Threshold Outliers

### 6.2.1 Varying Outlier Rates

We first analyze the effect of the outlier rate $\beta$ by varying $\beta$ from 0.001% to 100% with a fixed slide size of 500 and window size of 100K on synthetic data. As shown in Fig. 11.6(b), our Thresh_MinProbe and Thresh_LEAP are superior to DUE with respect to both CPU time and memory usage. In particular, as outlier rate is smaller than 0.01% which is common in real life application [47], Thresh_MinProbe shows a 10 times improvement over DUE while Thresh_LEAP gains another 100 times improvement on this basis in terms of CPU time. Thresh_MinProbe wins over DUE by applying the probing operation, which stops immediately after acquiring MESI rather than evaluates the complete neighborhood as range query search does. Thresh_LEAP further outperforms Thresh_MinProbe by applying the lifespan-aware prioritization principle which enables probing operation to always produce MESI with largest lifespan without introducing any additional cost. This minimizes the frequency of conducting probing in continuously evolving streams. The CPU time of all three methods increases as $\beta$ increases, because more computation time is spent on verifying the larger number of outliers. Our methods win for all outlier rates from 0.001% to 100%. That is, even in the extreme case when all data points are outliers, the overhead introduced by our methods is still smaller than DUE.

Thresh_MinProbe and Thresh_LEAP use on average 35% and 40% less memory than DUE. This is because Thresh_MinProbe and Thresh_LEAP only store the neighbor count of each slide for outlier candidates, while DUE maintains the actual neighbor relationships. Comparing against Thresh_MinProbe, Thresh_LEAP maximally accelerates the speed of discovering safe inliers. Since safe inlier introduces zero memory overhead, Thresh_LEAP consumes less memory.

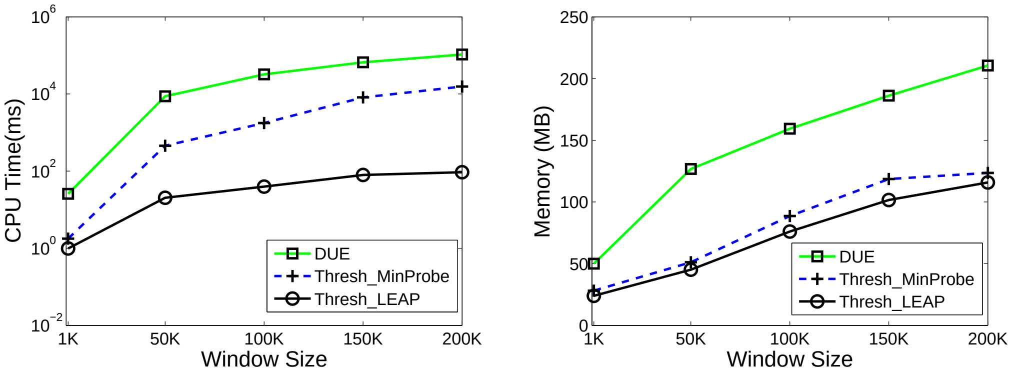
### 6.2.2 Varying Slide Sizes

Fig. 11.1 depicts the performance of the three algorithms for varying slide sizes on synthetic data when the outlier rate is fixed to 0.01% and the window size to 100k. Again Thresh_LEAP and Thresh_MinProbe clearly outperform DUE in CPU time, reaching up to 15 times and 1350 times improvement than DUE for small slide sizes. This is again due to the effectiveness of the LEAP operation as explained in the previous section. As the slide size increases, the processing time on each window increases accordingly. The reason is obvious. The larger slide size introduces more new data points, which in turn cost more CPU time to process. The CPU time of DUE increases by 290 seconds when varying the slide size from 0.5k to 50k, while Thresh_MinProbe and Thresh_LEAP increase only by 130 and 10 seconds.

Again, our method is not only superior in CPU but also in memory consumption. As the slide size increases, the percentage of safe inliers over the whole window increases, leading to less memory consumption for all three algorithms to store information for unsafe inliers.

### 6.2.3 Varying Window Sizes

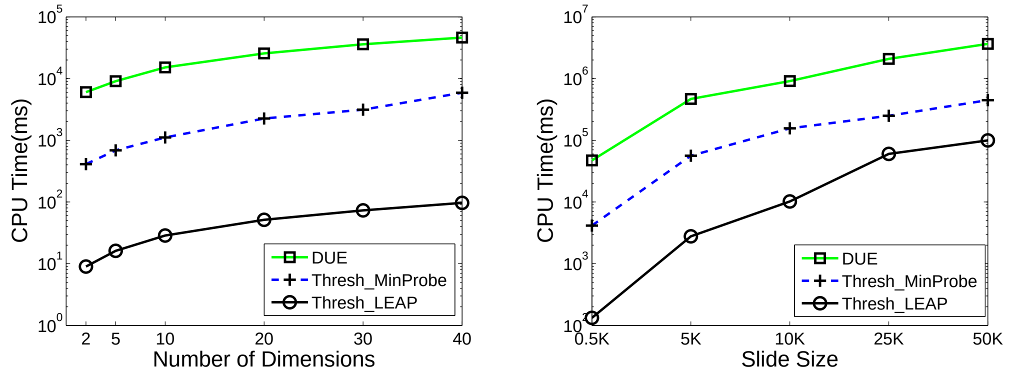
Next, we evaluate the effect of varying window sizes $\omega$ from 1k to 200k. We show the results on real dataset STT with fixed $k$ as 30, the outlier rate 0.1%, and slide size 500. In Fig. 6.3(a)-(b), Thresh_LEAP and Thresh_MinProbe outperform DUE in terms of both CPU and memory. In all cases, the CPU time consumed by Thresh_MinProbe is up to 1 order of magnitude smaller than DUE. Thresh_LEAP further outperforms Thresh_MinProbe by 2 orders of magnitude. As the window size increases, all algorithms consume more CPU time. Thresh_LEAP and Thresh_MinProbe take more CPU resources to process the triggered outlier candidates, while for DUE, larger window takes the range query



(a) CPU: STT (log scale)

(b) Memory: STT (linear scale)

**Figure 6.3:** LEAP: Varying Window Sizes on STT Real Dataset



(a) Synthetic dataset

(b) FC cover real dataset

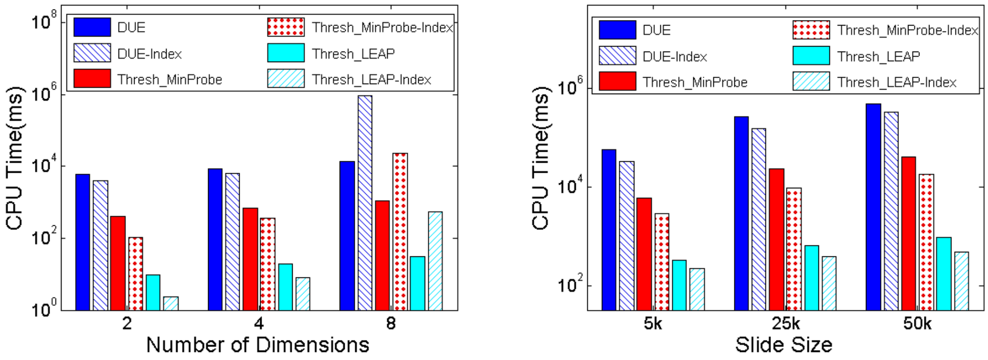
**Figure 6.4:** LEAP: Dimension Experiments

more time to search for neighbors for new arrivals. Thresh_LEAP wins more against Thresh_MinProbe in larger window. The reason is lifespan-aware prioritization enables probing operation to always acquire MESI with largest lifespan. When window size increases, the potential value of this lifespan increases, making this optimization even more effective. As the window size increases, Thresh_LEAP and Thresh_MinProbe still incur less memory consumption compared to DUE since only DUE has to store the actual neighbors.

### 6.2.4 Varying Dimensionality of Data

We evaluate the scalability of our algorithms on high dimensional data by varying the number of dimensions from 2 to 40. We fix the window size to 100K, slide size to 5K, and outlier rate to 0.1%. As shown in Fig. 6.4(a), Thresh_MinProbe algorithm consistently outperforms DUE around 15 fold in terms of CPU time, while Thresh_LEAP further outperforms Thresh_MinProbe around 35 fold. This is expected, since both our optimization principles are orthogonal to the number of data dimensions. The CPU costs of all three algorithms are near linear in the data dimensionality, because the cost of the distance calculation between two points is linear in the number of dimensions, while distance calculation costs are the most significant fraction of the overall outlier detection costs. This is the base price any method has to pay.

We also evaluate the performance of our algorithm on **real life** FC cover dataset (54 dimensions) by varying the slide size. The results shown in Fig. 6.4(b) again confirm the effectiveness of our approach to high dimensional datasets.



(a) Synthetic dataset

(b) STT real dataset

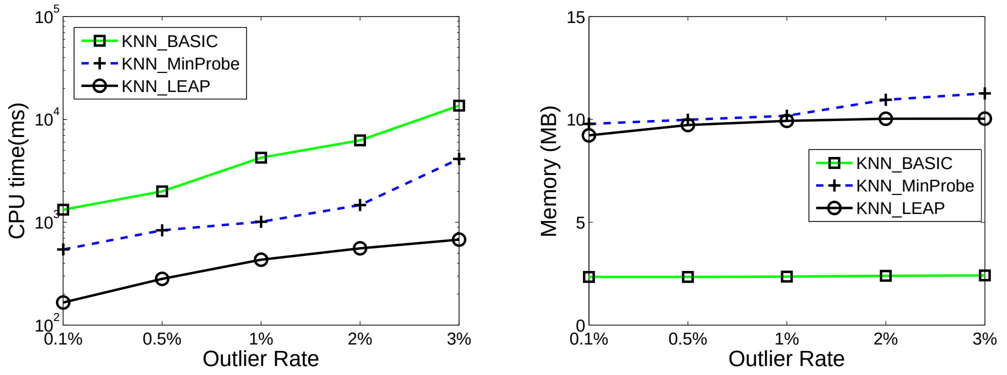
**Figure 6.5:** LEAP: Indexing Experiments

### 6.2.5 Effectiveness of Indexing

We compare all three algorithms against their corresponding indexed versions on synthetic dataset with the number of dimensions varying from 2 to 8. Other settings remain the same as in Sec. 6.2.4. As shown in Fig. 6.5(a), for the 2D and 4D cases, indexing improves the performance of all algorithms. This is as expected, because our both optimization principles are orthogonal to indexing as shown in Sec 3.5. In the 2D case with the help of the index, DUE reduces around 37 percent of their CPU costs, while Thresh_MinProbe and Thresh_LEAP reduce 64 and 76 percent respectively. This is because Thresh_LEAP and Thresh_MinProbe have different stopping criteria for the neighbor search than DUE. Given a data point $p_i$, Thresh_LEAP and Thresh_MinProbe first locate and probe for neighbors in the cell that $p_i$ falls in. This cell can be located in constant time using the grid index. Potentially Thresh_LEAP and Thresh_MinProbe will acquire enough neighbors of $p_i$, and hence terminate after searching through this single cell. On the other hand, DUE would not stop its search until all neighbors of $p_i$ are acquired. Therefore it locates all cells which could contain the neighbors of $p_i$, leading to a larger cell lookup costs compared to Thresh_LEAP and Thresh_MinProbe. Second, Thresh_MinProbe performs worse than Thresh_LEAP because it triggers probing operation more frequently and consequently introduce more expensive cell lookup operations.

However, as the number of dimensions increases, the number of the cells in the index to be examined also increases exponentially, leading to a significant increase of index maintenance overhead. This overwhelms the performance gain achieved by utilizing the grid index when the dimensions rise up to 8. In the 8D case, DUE introduces 900ms on average index maintenance costs per each slide which is much larger than the 60ms saved for distance calculation. This condition holds for all the algorithms. Thus, indexing performs well only on low dimensional datasets as had previously been observed for static data in the literature [8, 9].

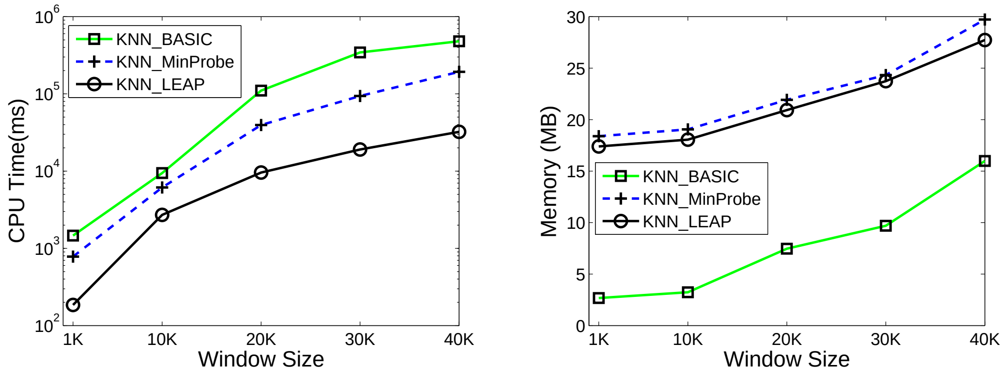
As shown in Fig. 6.5(b), our experimental results on **real life** STT dataset with varying slide size also confirms the orthogonality of our approach to the indexing.



(a) CPU (log scale)

(b) Memory (linear scale)

**Figure 6.6:** LEAP: Varying Outlier Rates on Synthetic Dataset



(a) CPU: STT (log scale)

(b) Memory: STT (linear scale)

**Figure 6.7:** LEAP: Varying Window Sizes on STT Real Dataset

## 6.3 Evaluating $K$NN Outliers

### 6.3.1 Varying Outlier Rates

This experiment evaluates the impact of varying outlier rates, namely varying $n$, on performance. We fix the window size at 10k and slide size at 1k, while varying $n$ from 10 to 300. Most practical applications have a low outlier rate (below 1%). Here we adopt outlier rates ranging from 0.1% to 3% as done in [27].

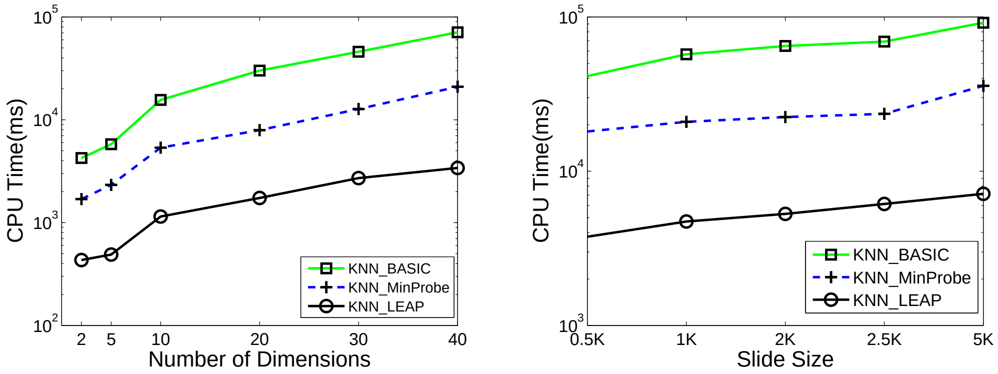
The CPU costs of all three algorithms increase as the outlier rate increases because a major part of the computation time is spent on processing the potential outliers. As shown in Fig. 6.6, KNN_MinProbe and KNN_LEAP both significantly outperform the baseline method KNN_BASIC. In particular, KNN_MinProbe outperforms KNN_BASIC 2.5 fold. KNN_LEAP further outperforms KNN_MinProbe 6 fold. The reason that KNN_MinProbe wins over KNN_BASIC is that it exploits the minimal probing principle to reuse the unexpired MESI members. Similar to distance-threshold outlier, KNN_LEAP wins over KNN_MinProbe because it searches for the MESI in an intelligent time-aware order. This minimizes the probing frequency needed.

The memory consumption of KNN_MinProbe is a little more than KNN_LEAP, while KNN_BASIC consumes less. This is as expected, because the first two need to maintain a similar $k$NN metadata structure per slide to reuse it in the next window. KNN_LEAP consumes less memory than KNN_MinProbe since it reduces the demand for acquiring new MESI members. The memory consumption is stable even with increasing outlier rates, making this a practical compromise for the tremendous gain achieved in CPU resources.

### 6.3.2 Varying Window Sizes

Here, we use the real dataset to evaluate the impact of varying window sizes. We fix the slide size at 200 and $n$ at 100, while varying the window size from 1k to 40k. As depicted in Fig. 6.7, the CPU costs of all algorithms rise as the window size increases. Yet our best solution KNN_LEAP consistently utilizes the least CPU time and exhibits the slowest increase in CPU consumption. KNN_LEAP and KNN_MinProbe are about 8 and 2 times faster than KNN_BASIC at $\omega$ = 1k case and up to 15 and 3 times faster when $\omega$ reaches 40k. For a fixed outlier rate, a larger window size results in a larger number of inliers and a wider lifespan range. Both factors are key for our framework to outperform the full $k$NN query search.

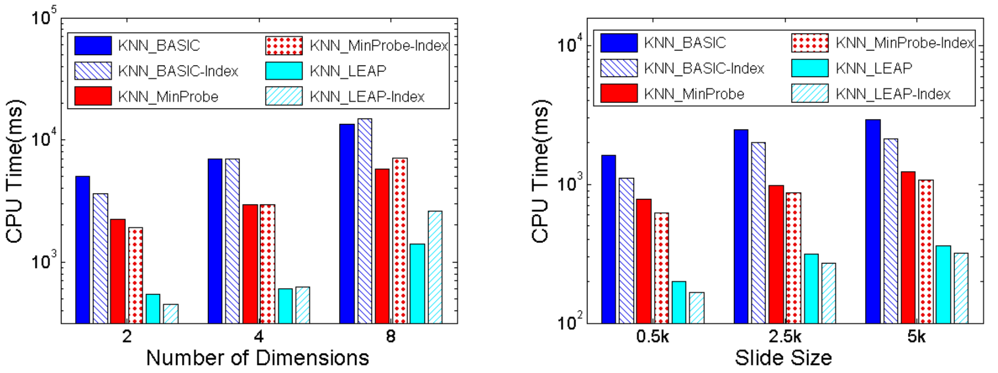
The memory consumption also scales with the window size. For KNN_LEAP and KNN_MinProbe, when the window size increases 40 times, the overhead only increases by about 2 fold. The reason is that the lifespan-aware evidence structure shares more lifetime proximity as the window size increases. This helps our approaches to achieve more compact storage.



(a) Synthetic dataset

(b) FC cover real dataset

**Figure 6.8:** LEAP: Dimension Experiments (kNN Based)



(a) Synthetic dataset

(b) STT real dataset

**Figure 6.9:** LEAP: Indexing Experiments (kNN Based)

### 6.3.3 Varying Dimensionality of Data

Fig. 6.8(a) demonstrates the CPU costs of all three algorithms as the number of dimensions increases from 2 up to 40. We fix window size at 10k, outlier rate at 1%, and slide size at 500. KNN_MinProbe and KNN_LEAP outperform KNN_BASIC even more as the dimension number increases. In 2D case, the KNN_MinProbe and KNN_LEAP outperform KNN_BASIC by 2.5 and 12 times respectively, while in 40D case they outperform KNN_BASIC by 4 and 20 times. This is because minimal probing and lifespan-aware principle both minimize the frequency of when the distance calculation has to be deployed. Therefore, when the distance calculation itself constitutes an even large percentage of overall computation cost with the increasing dimensions, they perform even better. In conclusion, KNN_LEAP performs consistently well as the number of data dimensions increases.

We also run experiment on **real life** dataset FC Cover by varying slide size. The results shown in Fig. 6.8(b) again confirm the effectiveness of LEAP to high dimensional datasets.

### 6.3.4 Effectiveness of Indexing

Fig. 6.9(a) shows that indexing improves the CPU resource consumption of all three algorithms for low dimensional data ($<$ 4D), while it starts to negatively impact the detection efficiency in higher dimensional cases. In the 8D case, indexing for the KNN_BASIC method reduces the distance calculation cost by 3000ms, yet costs 4500ms for maintaining the grid. A similar situation of maintenance costs superseding any achievable gain holds for our proposed algorithms. Therefore, the grid index benefits $k$NN outlier detection only when the data dimensions is rather low (in our case, $< 4$).

As shown in Fig. 6.9(b) the other experiment by varying slide size on **real life** STT data also confirms that our approach could benefit from the indexing as the data dimension is low.

# 7

# Related Work

**Distance-based Outliers on Static Data.** The $O_{thre}^{(k,R)}$ definition of distance-based outliers was first introduced by Knorr and Ng [8] for static datasets. They describe two detection algorithms. The cell-based algorithm, exponential in the number of data dimensions, is not scalable for high dimensional datasets. The index-based algorithm (using an R-tree or k-d tree) is shown to be non-competitive for three dimensional datasets and up if index building costs are considered. This implies that such relatively expensive indexing would not fit well in our streaming data scenario, because worst yet the index would have to be continuously re-built.

The $k$NN-based outlier definition was first introduced for static data in [9]. As they show for three dimensional datasets, their index-based (R*-tree) algorithm already performs worse than their partition-based algorithm even after excluding the index building costs. [53] proposes the Orca algorithm which outperforms the predecessor partition-based algorithm [9] with randomization and a simple pruning strategy. Orca scales well to high dimensional dataset. For this reason in this work we now adapt Orca to the streaming context and then use it as baseline to compare our framework against.

**Density-Based Outliers on Static Data.** Like distance-based outliers density-based outlier detection is a particular category of neighbor-based outlier detection techniques. They assign an outlier score to any given point by measuring the density relative to its local neighborhood restricted by a pre-defined threshold [10, 11]. Therefore density-based outliers are regarded as “local outliers”. However distance-based outlier detection instead takes a global view of dataset and marks each point as either outlier or inlier with respect to some user defined global parameters. Furthermore, both [10] and [11] only handle static datasets without taking the potential data update into account. Therefore the techniques proposed in [10] and [11] cannot be applied to solve our problem, namely detecting distance-based outliers on streaming data.

**Distance-based Outliers on Streaming Data.** With the emergence of digital devices generating data streams, outliers on streaming data have recently been studied [26, 27, 48]. However existing work [26, 27, 48] only considers the simpler distance-threshold variation of distance-based outliers. The processing of the more popular $k$NN-based variants [9] remains unsolved in the streaming context. Next we further elaborate on the existing results on this first outlier type.

In [48], given a data point $p_i$, it pre-computes the number of neighbors of $p_i$ for *each future window* that $p_i$ will participate in. It improves CPU performance at the expense of a huge memory overhead by pre-discounting the effect of expired data points for each and every future window in advance. Our work not only improves the CPU efficiency by three orders of magnitude, but also reduces the memory consumption.

[26] analyzes the expiration time of all neighbors of a point gathered by a range query. Then they use the expiration time of the neighbors to locate *safe inliers*, namely any point $p_i$ with more than k neighbors which have arrived after $p_i$.

[27] further outperforms [26] and [48] by integrating the *safe inlier* concept of [26] into an event queue, so that it can efficiently schedule the necessary checks that have to be made when points expire. However it still relies on full range query searches to

process newly arriving points. Therefore it fails to respond in real time when applied to high velocity streaming data targeted by our effort. In our work by exploiting the *minimal probing* and *lifespan-aware prioritization* principles, we succeed to avoid the full range query searches, thereby satisfying the performance requirements of modern streaming applications. Furthermore the above algorithms *ignore* indexing, while in our work we also investigate whether streaming outlier detection can benefit from indexing.

**Outliers on Sensor Data.** In [55] an interesting online technique is proposed to detect outliers in streaming sensor data. First, it utilizes a kernel density estimator to model the distribution of the sensor data. Then given a point $p_i$, the number of its neighbors is estimated by the density distribution function $f(p_i)$. Therefore [55] is able to quickly approximate whether $p_i$ is a $O_{thres}^{(k,R)}$ outlier. However the *approximation* nature determines that it cannot be directly applied to our context of computing *exact* distance-based outliers. Furthermore [55] only considers the $O_{thres}^{(k,R)}$ definition of distance-based outlier. The more popular $k$NN based definitions are not discussed.

**Stream Clustering.** The clustering definition most closely related to distance-based outliers is density-based clustering [48]. It puts adjacent points that have *enough* neighbors into the same cluster. This problem has been shown to be more expensive than distance-based outlier detection [48], because due to the inter-dependence among the data points the cluster structure is more complex to detect and update than the individual outlier points.

Most other clustering or summarization methods [56] instead focus on discovering accumulative statistical features of the stream. They do not specifically identify neighbor relationships among individual points, which is the key for distance-based outlier detection. Thus they are not directly applicable to our problem of distance-based outliers.

Yet in principle the general idea of micro-clusters or summaries [56] could potentially be exploited to eliminate points from dense areas that cannot be outliers. Clearly one could only eliminate points in dense areas as outlier candidates if the cell (micro-cluster) is small enough such that all points in the cell are neighbors with each other. However having such small cells tends to be not practical in streaming data with high dimensions, potentially requiring us to dynamically maintain too many cells (exponentially increasing with dimensions) and thus causing overwhelming costs.

# Part II

# Multi-query Outlier Detection Over Data Streams

# 8

# Varying 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*.

## 8.1 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 $\mathbb{Q}$.

### 8.1.1 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 [57] 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. 2.1), points in a dataset *D* are classified either as outliers or inliers. Thus, the process of identifying outliers in *D*

[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.

is equivalent to the process of finding and eliminating inliers from it. By Def. 2.1, *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 8.1** *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* $\infty$*.*

By Def. 8.1, $dist(p_i, p)$ = $dist(p_j, p)$ if $r_m < dist_o(p_i, p) < dist_o(p_j, p)$ ¡ $r_{m+1}$. This new *normalized distance* calculated using Def. 8.1 now accurately represents the importance of each data point to *p*.

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 8.2** ***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. 8.1.*

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. 23.3, 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* [1].

**Lemma 8.1** *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* $\mathbb{Q}$*.*

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

Here we sketch the key ideas of the proof for this lemma.

**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. 23.3, 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 $\{q_1, q_2, ..., q_m\}$.

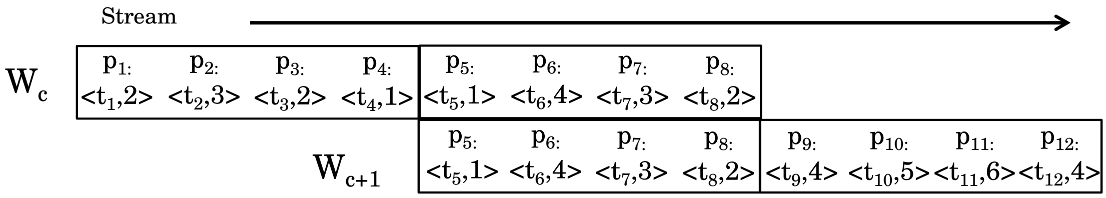


**Figure 8.1:** SOP: Sliding window stream

Next we illustrate these two observations with an example.

**Example 8.1** *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. 8.1. 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 $k$NN(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 $k$NN(p). Potentially it might still benefit the evaluation of *p* in the future windows.

As shown in Fig. 8.1 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 $k$NN(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 $q_3$.

### 8.1.2 The K-SKY Algorithm

Although the traditional *K-skyband* algorithms could be applied to support our $Q^s$ query [57, 58], we now design a customized algorithm called K-SKY that more efficiently supports the multiple outlier detection queries compared to existing algorithms [57, 58]. 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. 23.3. 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 evaluate 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. 23.3, 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 5** K-SKY($p, W_c.plist$,$p.skyband$,$\mathbb{Q}$)

```
Output: skybandPoints //the k-1-skyband point set
if $p.skyband$ == NULL then
    $W_c.input$ = $W_c.plist$; // New point;search from scratch
else
    expireSkyband($p.skyband$)
    $W_c.input$ = $p.skyband$ + $W_c.pList.new$; //Old point; search in new arrivals and unexpired skyband points
for each $p_i \in W_c.input$ from $W_c.input.tail$ to $W_c.plist.head$ do
    d = $dist(p, p_i)$;
    if (TRUE == $p_i$.skyEvaluate(d,skybandPoints, $\mathbb{Q}$)) then
        p.updateOutlierStatus($\mathbb{Q}$);
    else
        if d $\leq \mathbb{Q}.r_{min}$ then
            break;
```

**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. 5, 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 *r₁* 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 8.2** ***Optimality.*** *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 13.1 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. 23.3, $p_j$ 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 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. ■

**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 6** skyEvaluate(d,skybandPoints, $\mathbb{Q}$)

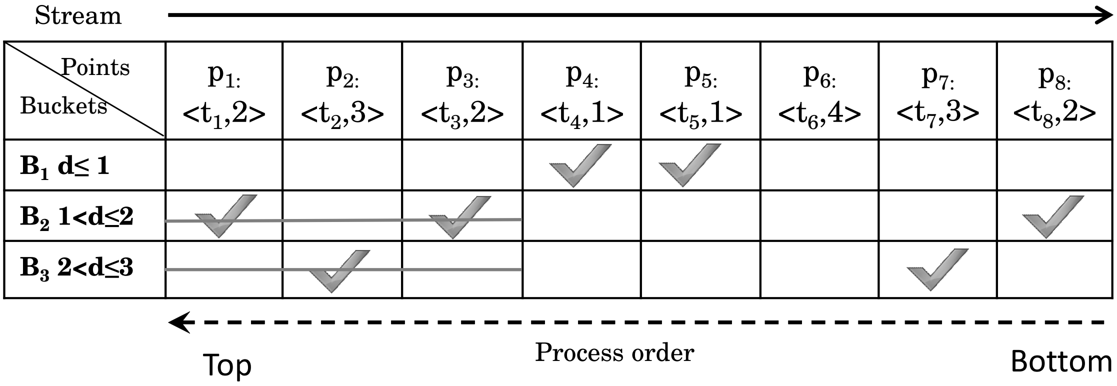
```
Output: isSkyband; //Boolean: skyband point or not
1: $layer$ = skybandPoints.getLayer(d);
2: count = 0;
3: for i =1; i++; i $\leq$ $layer$ do
4:     count += skybandPoints.layerCount(i);
5: if count $\leq$ k - 1 then
6:     skybandPoints.map($p_i$);
7:     return true;
8: else
9:     return false;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

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. 8.2, the points in each layer have the same distance to point *p* based on the normalized distance function in Def. 8.1. 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. 6, 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).



| Points / Buckets | $p_1$: <$t_1$,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> |
|---|---|---|---|---|---|---|---|---|
| **$B_1$ d≤ 1** | | | | ✓ | ✓ | | | |
| **$B_2$ 1<d≤2** | ✓ | | ✓ | | | | | ✓ |
| **$B_3$ 2<d≤3** | | ✓ | | | | | ✓ | |

**Figure 8.2:** SOP: 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 8.2** *Given the stream and the queries in Example 8.1, point $p_8$ is processed first by K-SKY as shown in Fig. 8.2. Since $1 < dist(p, p_8) = 2 \leq 2$, by Def. 8.1, $p_8$ 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. 8.1, $p_3$, $p_2$, 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 $O(|W|L\frac{|r|}{2})$.

## 8.2 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.

### 8.2.1 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 subgroup $\mathbb{Q}_j$ share the same k parameter value $(k_j)$. Therefore each $\mathbb{Q}_j$ corresponds to one skyband query $Q^s_j$.

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 $Q^s_j$ .

**Definition 8.3** ***Skyband Point Rule.*** *$p_i$ is a **skyband point** if:*

*(1) $p_i$ is hashed into some bucket $B_m$;*

*(2) $k' = \sum_{j=1}^{m} \mid B_j \mid < k_{max}$; 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. 8.3 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 $Q^s_j$ unless $k'$ is smaller than the *k* parameter $k_j$ of queries in $\mathbb{Q}_\gimel$ (the query sub-group corresponding to $Q^s_j$). Furthermore, any point $p_i$ will be discarded by query $Q^s_j$ if $dist(p, p_i)$ is larger than $r_n$ of $\mathbb{Q}_\gimel$ by the *domination relationship* defined in Def. 23.3. The above two conditions are captured by Condition 3 in Def. 8.3.

**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 $O(|W|LB)$.

## 8.2.2 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. 8.1.1). 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. 2.1, *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. 5) 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. 23.3, 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. 5), 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 $\mathbb{Q}$.

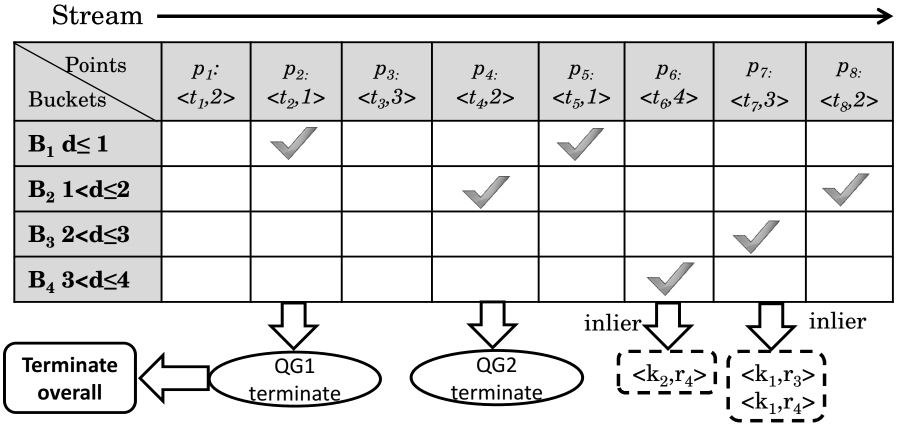
**Example 8.3** *Given two query groups $QG_1$ and $QG_2$ in Fig. 8.3, on stream in Fig. 8.4 we demonstrate how K-SKY supports outlier detection queries with varying k and r parameters. As shown in Fig. 8.4, $p_8$ is processed first and hashed to bucket $B_2$ of LSky. $p_7$ is processed next and inserted 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.*

**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. 7.

Given a point *p* in the current window $W_c$, Alg. 7 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

| | k \ r | $r_1 = 1$ | $r_2 = 2$ | $r_3 = 3$ | $r_4 = 4$ |
|---|---|---|---|---|---|
| **QG1** | $k_1 = 2$ | X | | X | X |
| **QG2** | $k_2 = 3$ | | X | X | X |

**Figure 8.3:** SOP: Queries With Varying *k* and *r* Parameters



| Buckets \ Points | $p_1$: $<t_1,2>$ | $p_2$: $<t_2,1>$ | $p_3$: $<t_3,3>$ | $p_4$: $<t_4,2>$ | $p_5$: $<t_5,1>$ | $p_6$: $<t_6,4>$ | $p_7$: $<t_7,3>$ | $p_8$: $<t_8,2>$ |
|---|---|---|---|---|---|---|---|---|
| $B_1$ d≤ 1 | | ✓ | | | ✓ | | | |
| $B_2$ 1<d≤2 | | | | ✓ | | | | ✓ |
| $B_3$ 2<d≤3 | | | | | | | ✓ | |
| $B_4$ 3<d≤4 | | | | | | ✓ | | |

**Figure 8.4:** SOP: K-SKY For Multiple Queries

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.

**Algorithm 7** detectOutlier( $W_c.plist$,$\mathbb{Q}$)

```
Output: outliers; //the outlier sets
1: for each $p \in W_c.plist$ do
2:     if (IsSafeInlier(p) == TRUE) then
3:         break;
4:     else
5:         K-SKY(p, $W_c.plist$,$p.skyband$,$\mathbb{Q}$);
6:         $\mathbb{Q}_{in}$ = markInlierStatus(p, $\mathbb{Q}$);
7:         markSafeInlier(p, $\mathbb{Q}$);
8:         if $\mathbb{Q}$ - $\mathbb{Q}_{in} \neq \emptyset$ then
9:             insertOutlier(outliers,p,$\mathbb{Q}$ - $\mathbb{Q}_in$);
```

# 9

# 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.

## 9.1 Varying the Window Parameter

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. 9.1. 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. 9.1. 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*.

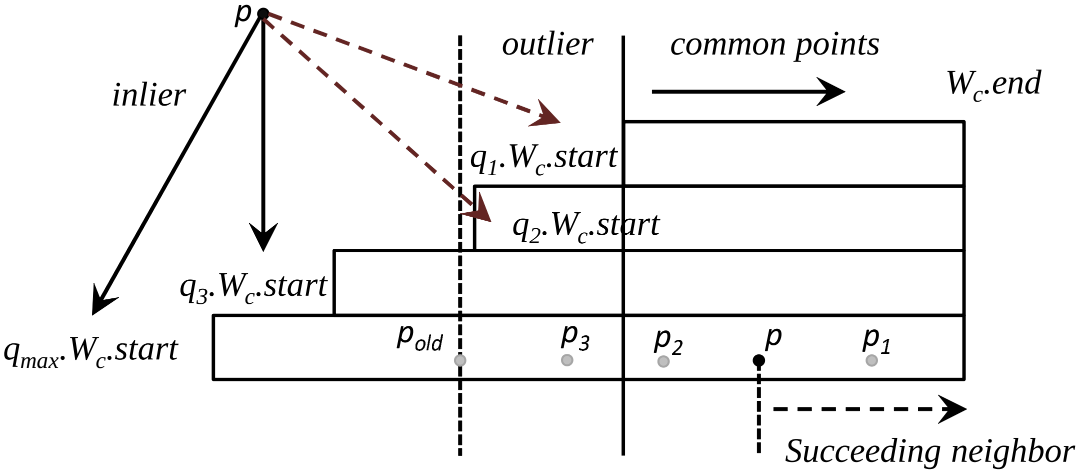


**Figure 9.1:** SOP: Queries with varying window sides

As shown in Sec. 8.1.2, 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. 9.1). 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. 8.1.2, 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 9.1** *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 $q^s_{max}$. ■

By Lemma 9.1 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. 9.1, 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. 9.1. 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.

## 9.2 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 $q_i.slide$ mod $q_{sft}$ = 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 output 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.

## 9.3 Varying Both Window 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. 9.2). Then Lemma 9.1 introduced in Sec. 9.1 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.

# 10

# 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. 9.3, 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. 10.1, 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.

After the query parser transforms the outlier detection queries into the skyband queries,

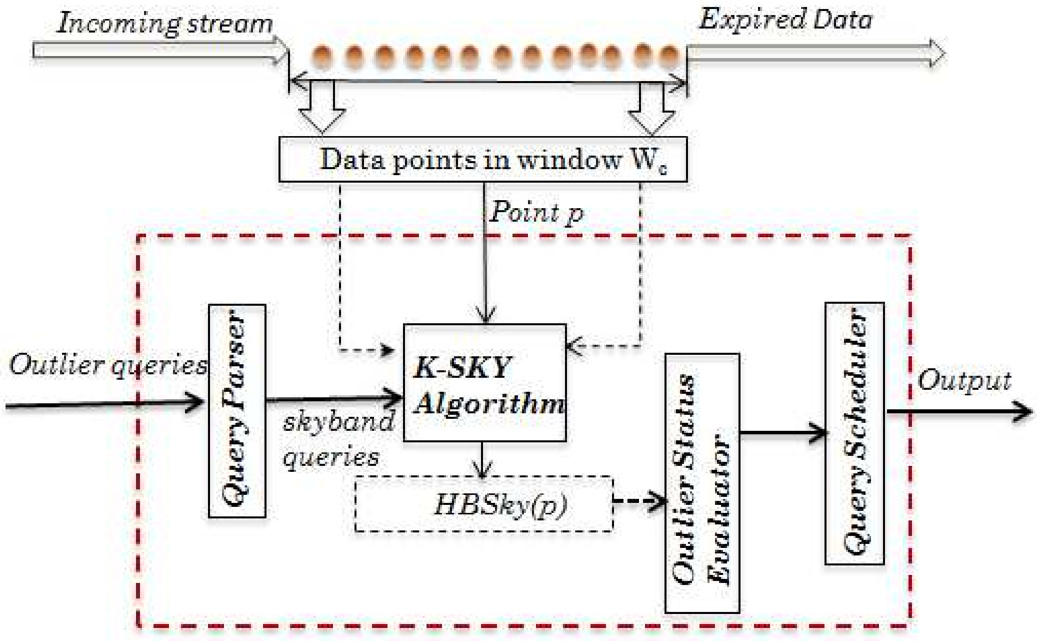


**Figure 10.1:** SOP Framework

the K-SKY algorithm for multiple skyband queries introduced in Sec. 8.2 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. 8.2.2.

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 9.1 has to be applied to evaluate whether *p* is indeed an inlier for these queries by checking their window sizes as shown in Sec. 9.1.

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. 9.2).

**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.

# 11

# Performance Evaluation

## 11.1 Experimental Setup & Methodologies

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 [51].

**Real Data Sources.** We use the Stock Trading Traces Data (STT) [52]. 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 [19, 27]. Since MCOD [27] does not support variations in window-specific parameters, we have extended MCOD by inserting our 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 [19]. 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.

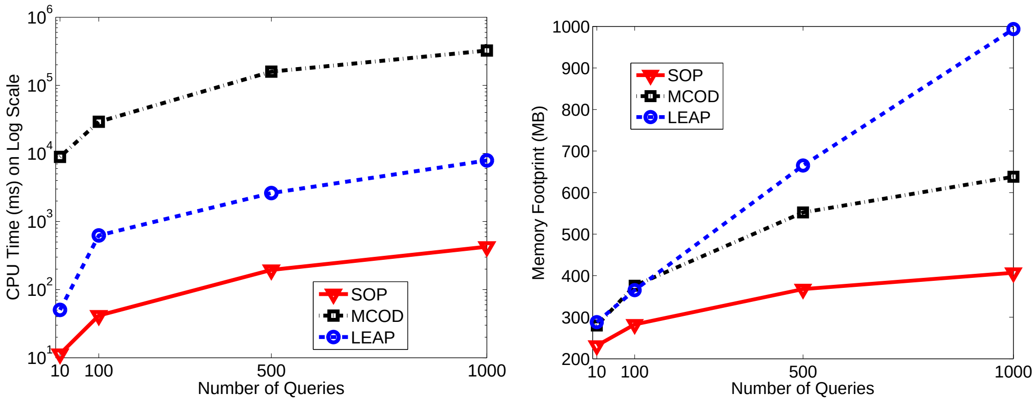
| Workload | Pattern | | Window | |
|---|---|---|---|---|
| | R | K | W | S |
| (A) | arbitrary | fixed | fixed | fixed |
| (B) | fixed | arbitrary | fixed | fixed |
| (C) | arbitrary | arbitrary | fixed | fixed |
| (D) | fixed | fixed | arbitrary | fixed |
| (E) | fixed | fixed | fixed | arbitrary |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

**Table 11.1:** SOP: Combinations of Different Workloads

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

**Table 11.2:** SOP: The Ranges of the Parameters

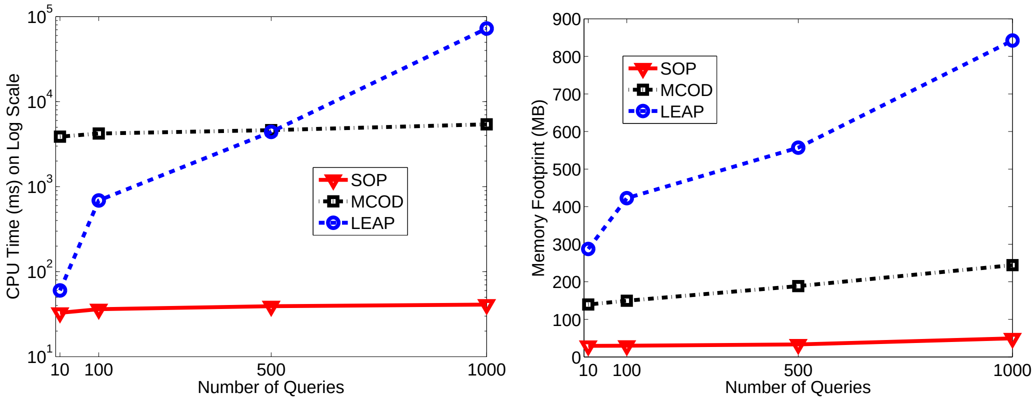
All in all our study covers important combinations of the four query parameters. They range from varying one specific parameter only at a time to the more general cases of varying all 4 of them among the queries populating the workload as shown in Table 11.1. The varying ranges of the parameters are listed in Table 11.2.



(a) CPU (log scale)

(b) Memory (log scale)

**Figure 11.1:** SOP: Varying R Values For Queries On Synthetic Dataset



(a) CPU (log scale)

(b) Memory (log scale)

**Figure 11.2:** SOP: Varying K Values For Queries On Synthetic Dataset

## 11.2 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 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 [27] and LEAP [19] 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. 11.1(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. 11.1(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.

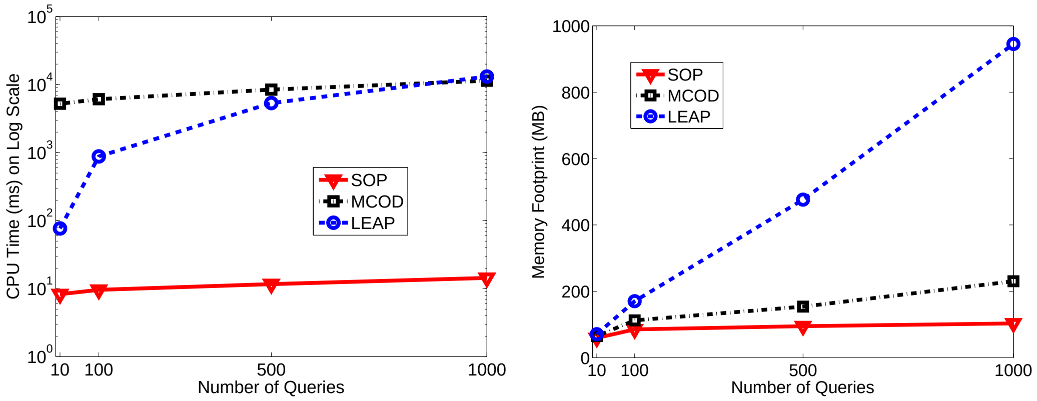
**Arbitrary K Case.** In this experiment, we analyze the performance of SOP by varying k parameter values of the queries. We use a fixed window size of 10K and a slide size of 0.5K. Parameter r is fixed at 700. A value for k is randomly selected in the range from 30 to 1500 for each query.

As shown in Fig. 11.2(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.

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. 11.2(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.

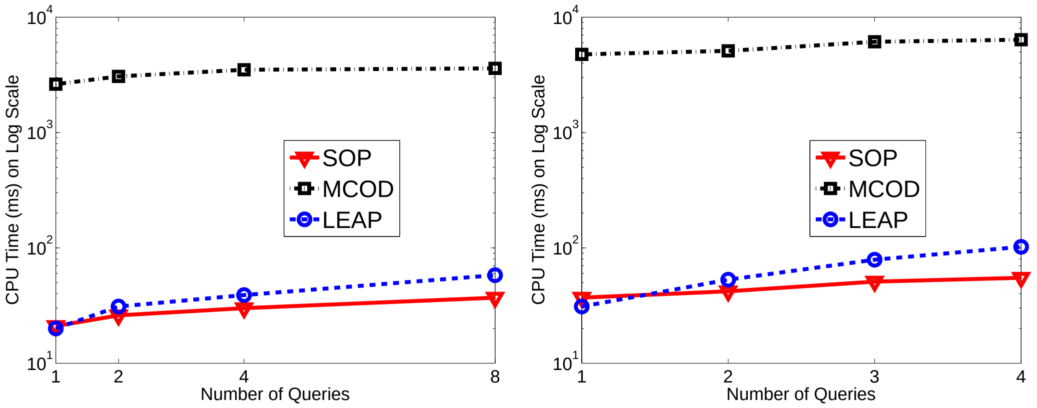
Fig. 11.3 depicts the performance of the three algorithms in terms of CPU costs and memory consumption. We observe that SOP consistently outperforms MCOD and LEAP up to 3 orders of magnitude. This confirms that K-SKY not only effectively shares the computation among the queries with an identical k parameter, but it also achieves



(a) CPU (log scale)

(b) Memory (log scale)

**Figure 11.3:** SOP: Varying K and R Values ON Synthetic Dataset



(a) Same attribute

(b) Different attributes

**Figure 11.4:** SOP: CPU (Log Scale): Small Workload

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. 11.4(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.

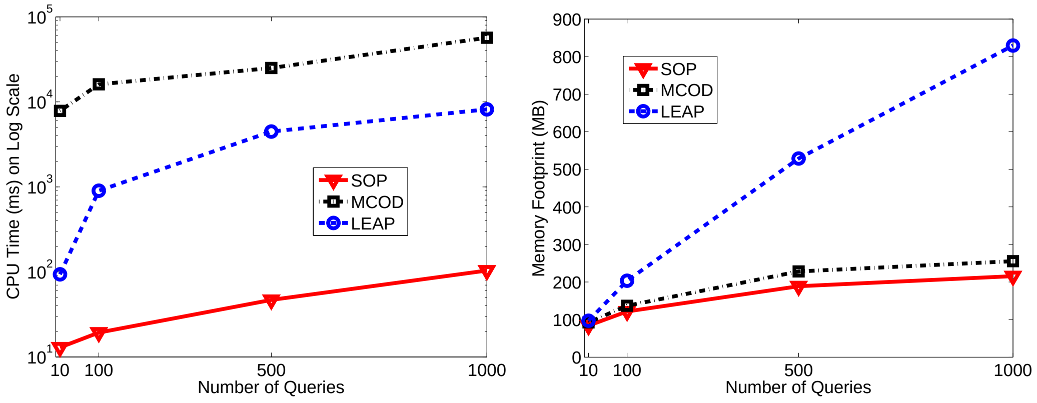
Furthermore, we also evaluate the performance of SOP when handling queries utilizing different set of attributes. In this experiment, the queries are divided into 3 groups. The queries in the same group utilize the same set of attributes. We vary the number of queries in each group from 1 to 4. To support such workload SOP is slightly extended using a simple divide and conquer approach. As depicted in Fig. 11.4(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.

## 11.3 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 [52] 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.

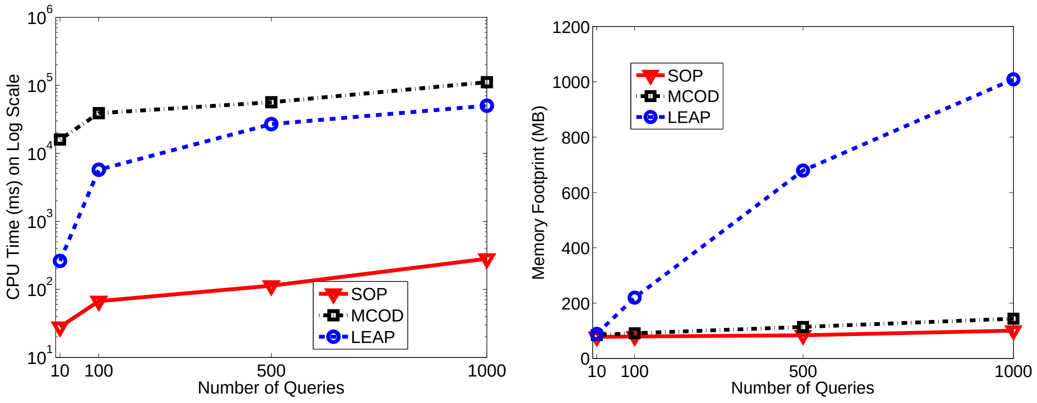
As shown in Fig. 11.5, SOP features significantly better performance on both CPU and memory consumption compared to MCOD and LEAP. Since MCOD leverages the



(a) CPU (log scale)

(b) Memory (linear scale)

**Figure 11.5:** SOP: Varying W For Queries On STT Dataset



(a) CPU (log scale)

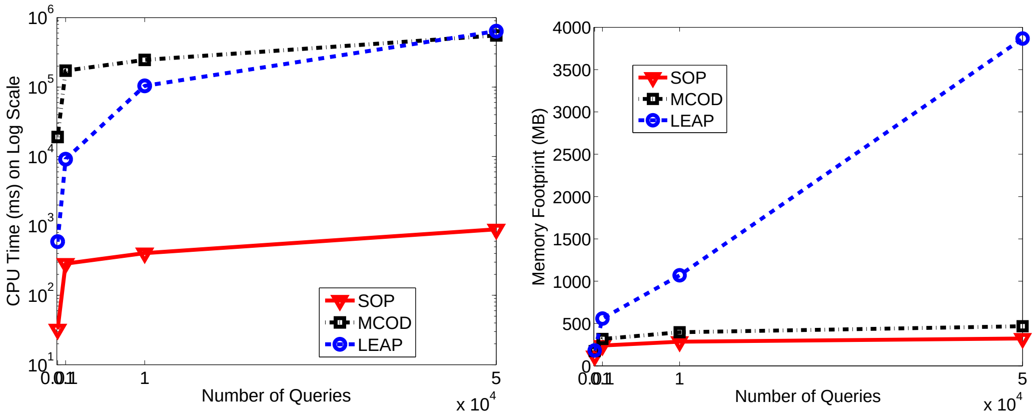
(b) Memory (log scale)

**Figure 11.6:** SOP: Varying W and S Values On STT dataset

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. 11.5(a). This is because based on our *safe-for-all* observation in Sec. 9.1, 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. 11.6, 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. 11.6 demonstrate the effectiveness of the swift query strategy for handling arbitrary win and arbitrary slide case.



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

**Figure 11.7:** SOP: Varying K, R, W and S values on synthetic dataset

## 11.4 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. 11.7(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. 11.7(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.

# 12

# 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 [19, 26, 27]. However as described in Chapter Chapter 7 existing work [19, 26, 27] focuses primarily on processing a single outlier detection request.

$k$**NN Queries on Streams.** Continuous $k$NN queries over sliding windows have indeed been studied in [54, 59]. 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 $k$NN set [59] or eagerly pre-compute the possible $k$NN set for each future window as new data arrives [54].

However to determine the outlier status of *p* it is not necessary to always discover the full $k$NN 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 $k$NN 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 [29, 31, 32, 33]. 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.

# Part III

# Distributed Outlier Detection: Distance-Based Outlier

# 13

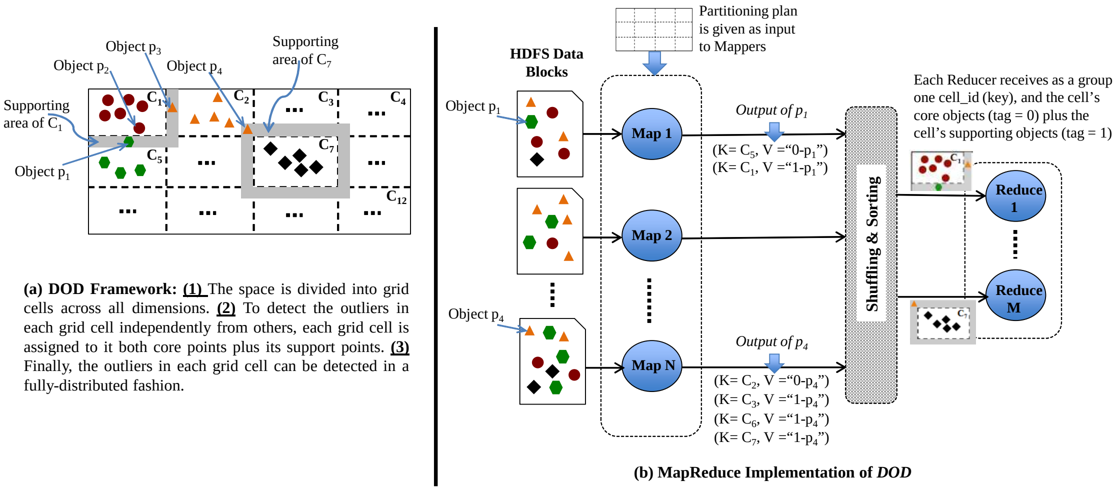
# Distributed Outlier Detection Framework

In this section, we introduce the basic *DOD* distributed framework that correctly discovers distance-based outliers using a single MapReduce job. This framework represents the solid foundation upon which key optimization strategies are then introduced. Below we will prove that under a uniform distribution of the data values, this proposed partitioning scheme is guaranteed to generate minimal communication overhead between the map and reduce phases in the MapReduce job.

## 13.1 The DOD Framework

*DOD* involves three key steps, namely *grid cell partitioning*, *enhancement with supporting area*, and *parallelized outlier detection* (refer to Figure 13.1).

**Step 1: Grid Cell Partitioning.** As a first step, *DOD* partitions the entire domain space of a dataset *D* $Domain(D)$ into *n* disjoint grid cells $C_i$ such that $C_1 \cup C_2 \cup ... \cup C_n$ = $Domain(D)$. A grid cell is formally defined below:



**(a) DOD Framework:** (1) The space is divided into grid cells across all dimensions. (2) To detect the outliers in each grid cell independently from others, each grid cell is assigned to it both core points plus its support points. (3) Finally, the outliers in each grid cell can be detected in a fully-distributed fashion.

**(b) MapReduce Implementation of *DOD***

**Figure 13.1:** DOD: Distributed Outlier Detection Using the DOD Framework.

**Definition 13.1** ***Grid Cell.*** *A grid cell $C_i$ in a d-dimensional domain space is a hyper rectangle $C_i = \langle(low_{1i}, high_{1i}),\ (low_{2i}, high_{2i}), ..., (low_{di}, high_{di})\rangle$, where $(low_{xi}, high_{xi})$ are the boundaries of $C_i$ in the $x^{th}$ dimension, where $1 \leq x \leq d$.*

The points inside cell $C_i$, referred to as the $C_i$'s **core points**, are denoted as $C_i.core = \{p_j \mid p_j \in C_i\}$. The areas of the domain space covered by each grid cell may or may not be of equal size. In general, any partitioning strategy could be utilized to produce such grid cells. Figure 13.1(a) depicts a two-dimensional space partitioned into grid cells using an *equi-width partitioning* method or in short *uniSpace*. That is, it divides each dimension of the domain space into equal width segments. Then points are grouped based on their membership in a particular grid cell. Fig. 13.1(a) shows such a grouping by representing the points in each grid cell with the same shape.

**Step 2: Enhancement with Supporting Areas.** The data points inside each grid cell are not sufficient to detect the outliers in each cell independent from the other cells. For example, data point $p_2$ in grid cell $C_1$ appears to be an outlier when considering only grid cell $C_1$. However, $p_2$ may have neighbor points in grid cell $C_5$, e.g., $p_1$, which may make $p_2$ an inlier. To break such dependency between the grid cells and thereby enable true parallelization for outlier detection among different grid cells, we introduce the notion of *supporting area*. As formally defined in Def. 18.1 the data points within the *supporting area* of cell $C_i$ may affect the outlier decision of at least one core point of $C_i$.

**Definition 13.2** ***Supporting Area.*** *The* ***supporting area*** *of a grid cell $C_i$, denoted as $C_i.suppArea$, is an extension of the boundaries of $C_i$ in each dimension of $D$. All data points $p_j$ (also called support point )$\in C_i.suppArea$ satisfy the following two conditions: (1) $p_j \notin C_i.core$, and (2) there exists at least one point $p_k \in C_i.core$ such that dist($p_j$, $p_k$) $\leq$ r, where r is the distance threshold parameter in Def. 2.1.*

Figure 13.1(a) highlights in grey the supporting areas of grid cells $C_1$ and $C_7$ respectively. Each grid cell $C_i$ will now be augmented with its support points in addition to its core points. For example, $C_1$ will be extended to contain *support points* $\{p_1,\ p_3\}$ in $C_1$'s supporting area, along with its circle-shaped *core points*.

**Step 3: Parallelized Outlier Detection.** The final step is to directly apply any centralized outlier detection algorithm, e.g., the *Nested-Loop* algorithm [8], to each of the grid cells $C_i$ to identify the outliers contained within that cell. This step can now be applied to each grid cell in total isolation from the others. Hence each can be distributed to different machines.

## 13.2 Optimality in Duplication Rate

The cost of a MapReduce algorithm is usually determined by two factors, namely communication and computation costs. The communication costs correspond to the costs of transmitting data from mappers to reducers. Often, if not always, the communication costs are the dominant costs of a MapReduce job [60]. Similar to the communication costs, the computation costs (especially those of the reducers) are also directly related to the number of the data points transmitted from mappers to reducers, i.e., the more data points that are received by the reducers, the more computational work is performed by them.

In the *DOD* framework presented in Figure 13.1, each data point has to be transmitted at least once from mappers to reducers since the latter are performing detection of the outliers. Therefore, the efficiency of the framework can be modeled using the notion of *"Duplication Rate"*, which refers to the average number of duplicates that mappers need to create for each input data point. The larger the duplication rate, the more data points must be transmitted from mappers to reducers, and thus the higher the communication and computation costs. The duplication rate is defined next.

**Definition 13.3** ***Duplication Rate (dr).*** *For a dataset $D$ and a MapReduce algorithm $A$ for detecting distance-based outliers in $D$, the "duplication rate" dr(D, A) $\in [1, \infty]$ represents the average number of duplicates that the mapper phase of $A$ generates per data point $p_i \in D$.*

Intuitively, to minimize the overall costs, an algorithm should produce all outliers for input dataset *D* in the fewest possible rounds of MapReduce jobs. In addition, mappers should transmit the smallest number of data points to the reducers to minimize the duplication rate. In the following, we show that the *DOD* framework using the *uniSpace* partitioning strategy is optimal w.r.t the duplication rate in the case of a uniformly distributed dataset. Without loss of generality, we will use our working example of a two-dimensional space in Figure 13.1(a).

**Lemma 13.1** ***Correctness and Minimal Duplication Rate.*** *Assume a two-dimensional uniformly-distributed dataset $D$, where the values in each dimension $d_1, d_2$ are normalized to $[0, 1]$ ($0 \leq d_1 \leq 1$ and $0 \leq d_2 \leq 1$). Consider n the number of the equi-width grid cells generated from the uniSpace partitioning strategy, and r the distance threshold* *parameter in the outlier detection problem. Then the DOD framework* ***correctly detects the distance-based outliers*** *in D in* ***a single MapReduce job*** *with the* ***minimal duplication rate*** $dr(D, DOD) = 1 + \pi r^2 n + 4r\sqrt{n}$.

**Proof.** Since the two-dimensional domain space of *D* is partitioned into equi-width squared grid cells of equal area sizes, the area that one grid cell $C_i$ covers is: $A(C_i)$ = $\frac{|d_1|\times|d_2|}{n}$, where $\mid d_1 \mid \times \mid d_2 \mid$ represents the area of the entire domain. Since $0 \leq d_i \leq 1, i \in \{0, 1\}$, then $\mid d_1 \mid \times \mid d_2 \mid= 1$. Therefore $A(C_i) = \frac{1}{n}$, and the side length of $C_i$, denoted as $l$, is computed as: $l = \sqrt{\frac{1}{n}}$.

Since *D* is uniformly distributed, each grid cell $C_i$ will hold the same number of core points $|core(C_i)| = \frac{|D|}{n}$. Moreover, the duplication rate over the entire dataset *D* denoted as $dr(D, DOD)$ will be equivalent to the duplication rate of a single grid cell $C_i$ denoted as $dr(C_i, DOD)$, where

$$dr(D, DOD) = dr(C_i, DOD) = \frac{|core(C_i)|+|C_i.suppArea|}{|core(C_i)|} \qquad (1)$$

Since the data values are uniformly distributed over the space, the cardinalities can be directly mapped to the underlying areas as follows:

$$dr(D, DOD) = dr(C_i, DOD) = \frac{A(C_i.suppArea)+A(C_i)}{A(C_i)} \qquad (2)$$

where $A(C_i.suppArea)$ represents the size of the supporting area of $C_i$.

As illustrated in Figure 13.1(a), $C_i.suppArea$ is composed of four (*r* x *l*) rectangles plus four quarter circles each of radius *r*. Therefore, $A(C_i.suppArea) = \pi r^2 + 4rl$ = $\pi r^2 + 4r\sqrt{\frac{1}{n}}$. By replacement in Eq. (2), we get:

$$dr(D, DOD) = dr(C_i, DOD) = \frac{\pi r^2+4rl+l^2}{l^2} = \frac{\pi r^2+4r\sqrt{\frac{1}{n}}+\frac{1}{n}}{\frac{1}{n}}$$

$$= 1 + \pi r^2 n + 4r\sqrt{n} \qquad (3)$$

To prove that $dr(D, DOD)$ is the minimal possible duplication rate, we first prove that for a given grid cell $C_i$, the support points in $C_i.suppArea$ are the necessary and sufficient set of points to determine the outlier status of the core points in $C_i$.

*"Necessity" Proof.* By Def. 18.1, any point $p_j \in C_i.suppArea$ is the neighbor of at least one point $p_i \in C_i$. If $p_j$ is excluded from $C_i.suppArea$, then possibly $p_i$ in $C_i$ would have been falsely reported as an outlier if $p_i$ happens to only acquire $k - 1$ neighbors.

*"Sufficiency" Proof.* Any data point $p_j \notin C_i.suppArea$ is not the neighbor of any point $p_i \in C_i$. Therefore $p_j$ has no influence on the decision of whether or not $p_i$ is an outlier by the distance-based outlier definition in Def. 2.1.

Next we show that the square-shaped grid cells lead to the lowest duplication rate. In other words, any other rectangle-shaped grid cells would lead to larger duplication rate. Suppose $C_j$ is a $y$ by $z$ rectangle cell. $C_j$ covers the same size of domain space as the square cell $C_i$ with side length $x$, and hence $x^2 = y \times z$. By applying Eq. (2) over $C_i$ and $C_j$, we get:

$$dr(C_i, DOD) = 1 + \frac{\pi r^2 + 4rx}{x^2} \tag{4}$$

$$dr(C_j, DOD) = 1 + \frac{\pi r^2 + 2ry + 2rz}{yz} \tag{5}$$

Given that $x^2 = y \times z$, to prove that $dr(C_i, DOD) \leq dr(C_j, DOD)$ we only need to show that $2x \leq y + z$. This is equivalent to proving that $(2x)^2 \leq (y + z)^2$, or equivalently $(y + z)^2 - 4x^2 \geq 0$. Since $x^2 = y \times z$, then by replacement of $x$, we need to prove that $((y + z)^2 - 4yz) \geq 0$. The L.H.S is equivalent to $(y - z)^2$, which is guaranteed to be always larger than or equal to zero. That is:

$$dr(C_i, DOD) \leq dr(C_j, DOD) \tag{6}$$

This proves the optimality claimed in Lemma 13.1. ■

```
Map (key k, value p) Function
Auxiliary Inputs
    - $S$ → Space partitioning strategy
    - $r$ → distance threshold of the outlier algorithm

Begin
    $C_i$ ← Compute the grid cell in which $p$ is a core point;
    output ($C_i$, "0-$p$");
    For (each grid cell $C_j$ in which $p$ is a supporting point) Loop
        output ($C_j$, "1-$p$");
    End Loop
End
```

```
Reduce (key cell-Id, v-list [ $p_1$, $p_2$, ..., $p_m$ ]) Function
Auxiliary Inputs
    - $k$ →count threshold of the outlier algorithm
    - $r$ → distance threshold of the outlier algorithm

Begin
    core-list ← the set of points in v-list with prefix tag = "0"
    support-list ←the set of points in v-list with prefix tag = "1"
    outlier-list ← Execute OutlierDetection(v-list, $r$, $k$);
    For (each outlier $o$ in outlier-list) Loop
        If ($o$ exists in core-list) Then
            output (null, $o$)          // Report Outliers
    End Loop
End
```

**Figure 13.2:** DOD: MapReduce Pseudocode of the DOD Framework.

## 13.3 MapReduce Implementation of DOD

We sketch one MapReduce implementation of the *supporting area* partitioning strategy in Figure 13.1(b). For the ease of implementation, instead of directly applying the supporting area definition in Def. 18.1, we utilize the simplified definition in Def. 13.4.

**Definition 13.4** *Given a d-dimensional grid cell $C_i$, the supporting area of $C_i$ is an $r$-extension to the boundaries of $C_i$ in each dimension. That is, $C_i.suppArea$ = $\langle(low_{1i} - r, high_{1i} + r),\ (low_{2i} - r, high_{2i} + r), ..., (low_{di} - r, high_{di} + r)\rangle$ - $C_i$, where*

$(low_{xi}, high_{xi})$ *are the boundaries in the* $x^{th}$ *dimension of* $C_i$ *(* $1 \leq x \leq d$*).*

Since the supporting area defined in Def. 13.4 is a superset of the supporting area in Def. 18.1, it is guarenteed to be sufficient to support each grid cell to be processed independently without relying on the points in any other cell.

The pseudocode of the map and reduce functions is presented in Figure 13.2. The input dataset, which resides in HDFS, has no prior partitioning properties, i.e., the data points are randomly distributed over the HDFS blocks. Each map function retrieves one data block as well as the space partitioning strategy (Figure 13.1(b)). Then for each data point $p_i$, the map function produces two types of output records, i.e., core- and supporting-related records.

The core-related record is one key-value pair record in the form of (K = $C_i$, V ="0-$p_i$"), where the key is the ID of the grid cell for which $p_i$ is a core point, i.e., $p_i \in C_i$. The prefixed flag "0" in the value component indicates that $p_i$ is a core point for $C_i$ (Lines 2-3 in the map function in Figure 13.2). For example, referring to Figure 13.1(b), the mapper *Map 1* generates output record (K = $C_5$, V ="0-$p_1$") for data point $p_1$.

Mappers also create zero or more supporting-related records for an input data point $p_i$ in the form of (K = $C_j$, V ="1-$p_i$"), where the key $p_i \in C_j$ is the ID of the grid cell for which $p_i$ is a support point, i.e., $p_i \in C_j.suppArea$. The prefixed flag "1" in the value component indicates that $p_i$ is a support point for $C_j$ (Lines 4-6 in the map function in Figure 13.2). For example, in Fig. 13.1(b), the mapper *Map N* generates three additional output records for point $p_4$ since it is a support point for $C_3$, $C_6$, and $C_7$.

After the internal shuffling and sorting phase based on the cell ID, each group received by a reducer will correspond to a specific grid cell, say $C_i$, and will consist of the union of the core and support points belonging to $C_i$ (See Figure 13.1(b)). The reducer function categorizes the data points according to their attached flag encoded in the value (lines 2-3 in the reduce function in Figure 13.2). Lastly, it executes an outlier detection algorithm to detect outliers within the $C_i.core$ set (lines 4-8).

# 14

# DOD with Load Balancing

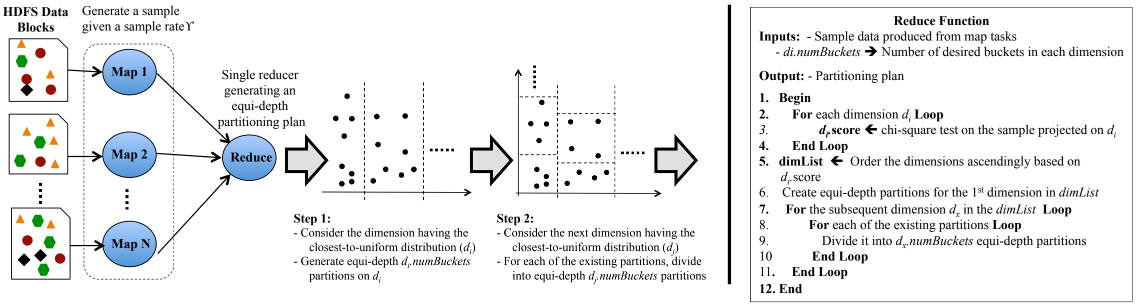
As presented in Section 13.2, the *DOD* framework adopting the uniform space partitioning method *uniSpace* leads to optimal duplication rate when handling uniformly distributed datasets. However, the datasets in most real-world applications are not uniformly distributed over the domain space. Therefore, the *uniSpace* partitioning method may, at times, cause a severe load imbalance. For example, *Reducer 1* in Figure 13.1(b) may process grid cell $C_1$ which contains an order-of-magnitude more points than $C_7$ processed by *Reducer M*. Load imbalance has been shown to be one of the most challenging problems for distributed data processing, e.g., [61]. It may not only result in significant slowdown, but also cause job failure in some cases. In this section, we investigate more sophisticated partitioning methods to overcome this challenge.

## 14.1 Data-Driven Partitioning (DDriven)

In this section, we propose a data-driven partitioning method called *DDriven* that now takes the data's distribution into account. Hence, *DDriven* generates grid cells that, in spite of having different grid sizes, contain a similar number of data points (*equi-*

*cardinality* cells). *DDriven* relies on a lightweight pre-processing strategy to determine a plan for partitioning the domain space into grid cells, henceforth called the *partitioning plan*. This new pre-processing phase is composed of two steps, namely *distribution estimation* and *partitioning-plan generation*. Both steps can be performed using one MapReduce job.

In the first step, *DDriven* estimates the distribution of the data by drawing a sample from the input dataset. We opt for random sampling since it preserves the distribution of the underlying dataset [62]. Since we only need to roughly estimate the distribution, the sampling rate $\Upsilon$ as an input parameter by default is set to a small value, e.g., 0.5 %. Considering the size of big datasets, the sample is generated in a distributed fashion, for example by drawing samples within the map phase of a MapReduce job. Then, the mappers' output is passed to a centralized node, i.e., a single reducer, for the plan generation step. Since drawing the random sample at the map phase is intuitive [62], we ignore the details here.


HDFS Data Blocks
Generate a sample given a sample rate $\Upsilon$
Map 1
Map 2
Map N
Single reducer generating an equi-depth partitioning plan
Reduce
Step 1:
- Consider the dimension having the closest-to-uniform distribution ($d_i$)
- Generate equi-depth $d_i$.*numBuckets* partitions on $d_i$
Step 2:
- Consider the next dimension having the closest-to-uniform distribution ($d_j$)
- For each of the existing partitions, divide into equi-depth $d_j$.*numBuckets* partitions

**(a) The data driven partitioning for generating approximate equi-depth partitions**

```
Reduce Function
Inputs:  - Sample data produced from map tasks
   - di.numBuckets ➔ Number of desired buckets in each dimension

Output: - Partitioning plan

Begin
   For each dimension $d_i$ Loop
         $d_i$.score ← chi-square test on the sample projected on $d_i$
   End Loop
dimList ← Order the dimensions ascendingly based on
   $d_i$.score
Create equi-depth partitions for the 1st dimension in dimList
  For the subsequent dimension $d_x$ in the dimList Loop
        For each of the existing partitions Loop
           Divide it into $d_x$.numBuckets equi-depth partitions
        End Loop
   End Loop
End
```

**(b) Pseudocode of the centralized *Reduce()* Function**

**Figure 14.1:** DOD: Equipped with the Data-Driven Partitioning Strategy (*DDriven*).

In the next step, the partitioning plan is generated by the single reducer (Figure 14.1(b)). In addition to the sample data, the reducer receives a list that specifies the number of desired partitions in each dimension. For instance, $d_i.numBuckets$ is the number of desired partitions for dimension $d_i$. This list is calculated based on the number of reducers *n* assigned to the outlier detection task. By default, each dimension has the same number of partitions, with $d_i.numBuckets$ set as $\sqrt[d]{n}$ for a d-dimensional dataset.

The reducer computes for each dimension $d_i$ a *uniformity score*, i.e., a chi-square test score that measures whether or not the values on $d_i$ are close to a uniform distribution (lines 2-4). The smaller the score $d_i.score$, the more uniformly the points are distributed over $d_i$. Hence the better it is to start with this dimension in partitioning the dataset. The dimensions are sorted in an ascending order according to their chi-square test score (line 5). The $1^{st}$ dimension is selected to be partitioned first (line 6). Then the subsequent dimension having the next-smallest score, say $d_x$, is selected for further partitioning (lines 7-11). This is performed by considering each of the existing cells, and dividing that cell over $d_x$ into $d_x.numBuckets$. Using this strategy, although the distribution of the dataset as a whole may be skewed, the subsets falling into each grid cell tend to be relatively uniform. As illustrated in Chapter 13, a uniformly distributed dataset leads to a low duplication rate and lower overall costs.

Figure 14.1(a) illustrates an example of the partitioning process. We assume that the x-axis is the $1^{st}$ selected dimension. Thus, the points are divided vertically into $d_x.numBuckets$ equi-cardinality cells. The next selected dimension is the y-axis. Each of the existing vertical partitions will be further divided into $d_y.numBuckets$ equi-cardinality cells. This process proceeds until all dimensions are partitioned. The generated *partitioning plan* is then passed to the *DOD* framework presented in Section 13.1 for deployment.

## 14.2 Cost-Driven Partitioning (CDriven)

As presented in Section 14.1, the data-driven partitioning method *DDriven* tries to achieve a balanced workload across the computational nodes (reducers). However, this is based on the traditional assumption that *an equal number of data points leads to a balanced work-*

*load* [35, 36, 37, 38]. Although this assumption is widely adopted by most distributed analytical algorithms, we will show that this does not hold in the context of distance-based outlier detection. We will demonstrate this important observation using both an empirical study and a theoretical analysis.

For this we design an experiment to study the effect of the data's density on the execution of an outlier detection algorithm (Figure 14.2). We use two datasets, each consisting of the same number of data points (100KB). However their densities are very different, where *D-Dense* is much "denser" than *D-Sparse*. Here the density is defined as the *ratio of data cardinality to the domain area covered by the data*. The domain area covered by the *D-Dense* dataset is only $\frac{1}{4}$ of the domain area covered by the *D-Sparse* dataset. By the above measure, *D-Dense* is four times denser than *D-Sparse*.

We then apply the *Nested-Loop* algorithm [8] to both datasets with the *r* and *k* parameters set to 5 and 4 respectively. The *Nested-Loop* algorithm is among the most popular algorithms for distance-based outlier detection. Its logic is based on the following idea. Given a data point $p_i$, the algorithm evaluates the distance between $p_i$ and other points in the dataset *D* in random order until either *k* neighbors of $p_i$ are found ($p_i$ becomes inlier), or all data points in *D* are examined ($p_i$ becomes outlier). As depicted in Figure 14.2, although the input data size and the algorithm's input parameters are exactly identical, the execution performance is entirely different (4.5x slower in the case of *D-Sparse*).

The intuitive explanation is that the data points in *D-Dense* are closer to each other. Thus finding enough neighbors of a given point $p$ within a distance $r$ to declare $p$ as inlier is relatively faster in *D-Dense* than that in *D-Sparse*. That is, the likelihood that a randomly picked point in *D-Dense* is a neighbor of $p$ is higher than that in *D-Sparse*. Since outliers tend to be rare and thus the vast majority of the data points can be expected to be inliers, the algorithm applied to *D-Dense* will terminate early for most points. This explains the significantly lower overall costs compared to *D-Sparse*. This experiment reveals that the processing costs do not only depend on the dataset's cardinality, but also its densities and distribution over the domain space.

Next, we theoretically support this observation by establishing a formal model (Lemma 14.1) for the family of outlier detection algorithms that rely on random selection and comparisons among the data points, such as Nested Loop [8].

**Lemma 14.1** *Given a uniformly-distributed dataset $D$ of cardinality $|D|$ data points, and parameter settings $k$ and $r$ of the distance-based outlier algorithm,* ***the cost of detecting distance-based outliers based on random comparisons*** *is computed as $Cost(D) = \frac{|D| \times A(D) \times k}{A(p_i)}$, where $A(D)$ and $A(p_i)$ represent the areas of the domain space covered by the entire dataset $D$ and by the distance parameter $r$ around $p_i$, respectively.*

**Proof.** Since $D$ follows a uniform distribution, we have:

$$Cost(D) = Cost(p_i) \times |D| \tag{7}$$

where $Cost(p_i)$ denotes the cost of determining whether or not a data point $p_i$ is outlier. Since $p_i$ has, on average, $|D| \times \frac{A(p_i)}{A(D)}$ neighbors in $D$. Then, given any randomly picked point $p_j$, the probability that $p_j$ is a neighbor of $p_i$ denoted as $\mu$ equals to:

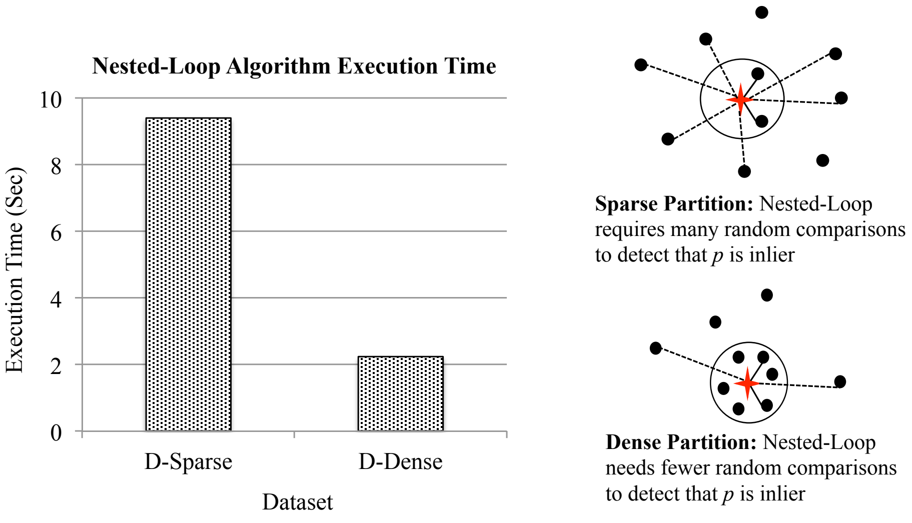
$$\mu = |D| \times \frac{A(p_i)}{A(D)} \; / \; |D| = \frac{A(p_i)}{A(D)} \tag{8}$$

The cost of processing $p_i$ denoted by $Cost(p_i)$ is determined by the number of trials $N$ to acquire $k$ neighbors. Considering the event that a randomly picked point is or is not a neighbor of $p_i$ as a binary variable, then the probability of observing $k$ occurrences of neighbors in a set of $N$ samples (random trials) follows the Binomial distribution $Bin(k \mid N, \mu) = \binom{N}{k} \mu^k (1 - \mu)^{N-k}$. And the expected value of $N$ is $E(N) = \frac{k}{\mu}$, which leads to:

$$Cost(p_i) = \frac{k}{\mu} = \frac{k}{\frac{A(p_i)}{A(D)}} = \frac{k \times A(D)}{A(p_i)} \tag{9}$$

By substitution in Eq. (7), we get $Cost(D) = \frac{|D| \times A(D) \times k}{A(p_i)}$, which proves the lemma. ■

Based on Lemma 14.1, the cost of detecting the distance-based outliers in $D$ relies on both the number of the data points, i.e., $|D|$, and the domain space area covered by the dataset $A(D)$. Since the domain space covered by a sparse dataset *D-Sparse* is larger than the domain space covered by a dense dataset *D-Dense*, we thus can conclude that *Cost(D-Sparse)* $>$ *Cost(D-Dense)*.



**(a) Experimental Evaluation** **(b) Sparse vs. Dense Partitions**

**Figure 14.2:** DOD: Sensitivity of Nested-Loop's Performance to Dataset Densities.

***CDriven* Partitioning Algorithm.** Inspired by Lemma 14.1, we propose a *cost-driven* partitioning method to replace the data-driven partitioning method (*DDriven*) utilized in the *DOD* framework. Instead of setting each grid cell to contain an equal number of data points, the cost-driven partitioning method called *CDriven* utilizes the cost model established in Lemma 14.1 to divides the domain space into grid cells each with similar *respective costs*. Therefore, the key improvement of *CDriven* over *DDriven* is the criteria used to determine how to split the large cells into smaller ones in an iterative partitioning process (line 9 in Figure 14.1(b)).

Recall that the cost model established in Lemma 14.1 is based on the assumption that the dataset follows a uniform distribution. We now observe that this is a reasonable model to use in real-world datasets even if the data is not fully uniform. This is so because, as presented in Figure 14.1(a), the partitioning method aims to make each grid cell uniform. Therefore, although the overall distribution of the input dataset may be skewed, each data subset within a single grid cell tends to be uniform. Hence, the processing costs of each grid cell calculated by Lemma 14.1 remain accurate estimates.

# 15

# DOD with Multi-Tactic Detection

So far the *DOD* framework follows the state-of-the-art approach of applying the same centralized detection algorithm to all reducers, e.g., *Nested-Loop*. However, it has already been observed in [18] that no conclusive winner emerged among alternative outlier detection algorithms that consistently outperformed all other algorithms on all datasets. In our distributed context we now leverage this shortcoming as an advantage. Since each reducer works independently of all other reducers, multiple distinct detection algorithms can be deployed concurrently in our *DOD* framework. That is, we propose the novel *multi-tactic* approach that selects a distinct outlier detection algorithm for each data partition driven by the particular characteristics of the data therein.

With the introduction of this *multi-tactic* outlier detection approach, the data partitioning problem (performed by the mappers), and the choice of the detection algorithms (employed by the reducers) become *strongly interdependent*. That is, to minimize the overall costs of the distributed outlier detection process, the new problem arises that the partitioning plan should be driven by the costs estimated from a specific detection algorithm. On the other hand, algorithms must be selected for each partition based on the characteristics of the data subsets produced by the chosen partitioning plan. Clearly, this is a proverb chicken and egg problem.

## 15.1 Density Matters

Our proposed solution rests on our solid theoretical analysis and the understanding of the performance of the typical classes of detection algorithms, such as the *Nested-Loop* and *Cell-Based* algorithms [8]. Our comprehensive analysis reveals that the *density* of each partition is the key factor that affects the performance of these algorithms. Again, we will support this observation both empirically and theoretically.

Similar to *Nested-Loop* introduced in Section 14.2, the *Cell-Based* algorithm [8] is another popular detection algorithm. As an index-based solution, it relies on pruning strategies to avoid checking un-necessary points. First, it uniformly partitions the domain space into a set of $d$-dimensional non-overlapped grid cells, where the length of the cell in each dimension is $r/2$ and $r$ is the distance threshold input parameter. Then it hashes each data point to exactly one grid cell. Each cell maintains the number of points it contains so that the algorithm can quickly identify all grid cells that have no outlier or no inlier. Both types of cells can be excluded from any further processing. The data points in the remaining grid cells have to be evaluated individually, in a fashion similar to *Nested-Loop*.

First, we evaluate the performance of the *Nested-Loop* algorithm versus *Cell-Based* under different data densities. Again *density* is defined as *the ratio of data cardinality to the domain area covered by the data*. In this experiment, we vary the density of the datasets by varying the size of the domain area while keeping the number of data points constant as 10,000. The $r$ and $k$ parameters are set as 5 and 4, respectively.

The results depicted in Figure 15.1 confirm our expectation that densities matter, and no algorithm is superior in all cases. Better yet, we observe a general trend in the results. Namely, the *Cell-Based* algorithm outperforms *Nested-Loop* in the cases where the data is either very sparse or very dense. In contrast, in the intermediate density cases the *Nested-Loop* algorithm is faster.

Intuitively the *Cell-Based* algorithm performs well when handling very sparse or very dense datasets, because in both cases, many of the *d*-dimensional grid cells can be directly marked as outliers (in the very sparse case) or as inliers (in the very dense case). This then saves computations. In other cases, deciding on the outlier status of the data points requires more computations on top of the pre-processing phase. Here, the *Cell-Based* algorithm suffers from the additional overhead of having to index the data points. It thus performs worse than *Nested-Loop*.

Next, we theoretically support this observation based on the cost model we establish for the *Cell-Based* algorithm (Lemma 15.1).

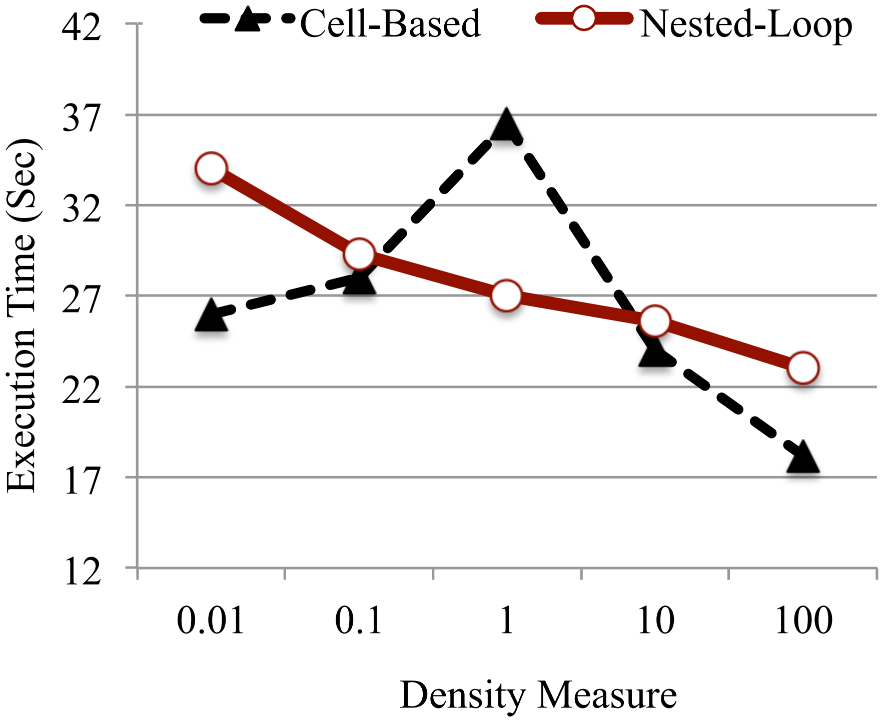


**Figure 15.1:** DOD: Performance of Different Detection Algorithms w.r.t Data's Densities.

**Lemma 15.1** *Given a uniformly distributed two-dimensional dataset $D$ of cardinality* $|D|$

*and data points that cover a domain space of area $A(D)$, the cost of detecting distance-based outliers using the Cell-Based algorithm with parameters $r$ and $k$ is defined as follows:*

*(1) $Cost(D) = |D|$ If $\frac{9}{8}r^2 \times \frac{|D|}{A(D)} \geq k$;*

*(2) $Cost(D) = |D|$ If $\frac{49}{8}r^2 \times \frac{|D|}{A(D)} < k$;*

*(3) $Cost(D) = |D| + \frac{|D| \times A(D) \times k}{\pi \times r^2}$ Otherwise*

**Proof.** In the *Cell-Based* algorithm, the area covered by each cell $C_i$ corresponds to $A(C_i) = \frac{1}{2}(\frac{r}{2})^2 = \frac{r^2}{8}$, where $\frac{r}{2}$ is the diameter of one rectangular cell. According to the algorithm, given a cell $C_i$, if there are more than $k$ points in $C_i$ and its direct adjacent cells (nine cells in total), then all data points in $C_i$ are marked as inliers. In other words, if $\frac{9}{8}r^2 \times \frac{|D|}{A(D)} \geq k$, then the cost of processing the entire dataset is equivalent to scanning and indexing the data points. That is $Cost(D) = |D|$. This proves Equation (1) of Lemma 15.1.

On the other hand, if there are fewer than $k$ points in the combined cells of $C_i$ and the cells within $2r$ distance from it (in total 49 cells), then all points in $C_i$ are guaranteed to be outliers without requiring any explicit comparisons. In other words, if $\frac{49}{8}r^2 \times \frac{|D|}{A(D)} < k$, then $Cost(D) = |D|$. This proves Equation (2) of Lemma 15.1.

If neither of the two aforementioned cases hold, then the unmarked cells need to execute a *Nested-Loop* algorithm, in addition to the indexing costs of the entire dataset. That is, the cost will be $Cost(D) = |D| + \frac{|D| \times A(D) \times k}{\pi \times r^2}$, where $\frac{|D| \times A(D) \times k}{\pi \times r^2}$ represents the cost of *Nested-Loop* as proven in Lemma 14.1. This proves Equation (3) of Lemma 15.1 (3). ■

According to Lemma 15.1, in the extreme cases of very sparse and very dense, the cost of *Cell-Based* is linear w.r.t $|D|$. Thus it outperforms *Nested-Loop*. Whereas in the other cases, it is more expensive than *Nested-Loop* due to overhead introduced by the indexing phase, without added benefit from this extra step.

```
Reduce Function
Inputs:
    - sample data produced from map tasks
    - numPartitions ➔ Number of desired partitions
    - numReducers ➔ Number of reducers in the system

Output: - Partitioning plan        // how the data space is partitioned
        - Execution strategy       // which algorithm will run on which partition

Step 1: Initialization of mini clusters
- For each mini bucket (mini cluster) b Loop    // Compute some statistics
    - b.count 🡐 count of points in b
    - b.densityMeasureArray[] 🡐 chi-square score for each dimension
    - b.densityMeasureGlobal 🡐 Aggregated score for the bucket
- End Loop

Step 2: Hierarchical clustering and generation of partitioning plan
- For each cluster c Loop
    - Generate candidate surrounding-neighbors extensions (rectangle shapes)
    - For each candidate cluster, measure the similarities in their densities
    - c.bestExt 🡐 The candidate with the highest similarity in densities
- End Loop
- nextExt 🡐 Select the extension with the highest similarity score
- Merge the clusters in nextExt to form one cluster, and update its statistics
- Repeat Step 2 until the number of clusters = numPartitions

Step 3: Cost estimation and generation of execution strategy
- For each partition p & a possible outlier detection algorithm g Loop
      - Estimate the cost of g on p
      - p.(algorithm, cost) 🡐 assign to p the cheapest algorithm at its cost.
- End Loop

Step 4: Assignments to reducers and generation of assignment plan
    // Divide the numPartitions partitions into numReducers  groups, where the
    // sum of costs in each group is roughly the same.
-MultiBinPacking(numPrtitions, numReducers);
```

**Figure 15.2:** DOD: The Pre-Processing Stage of DMT (Reduce-Side).

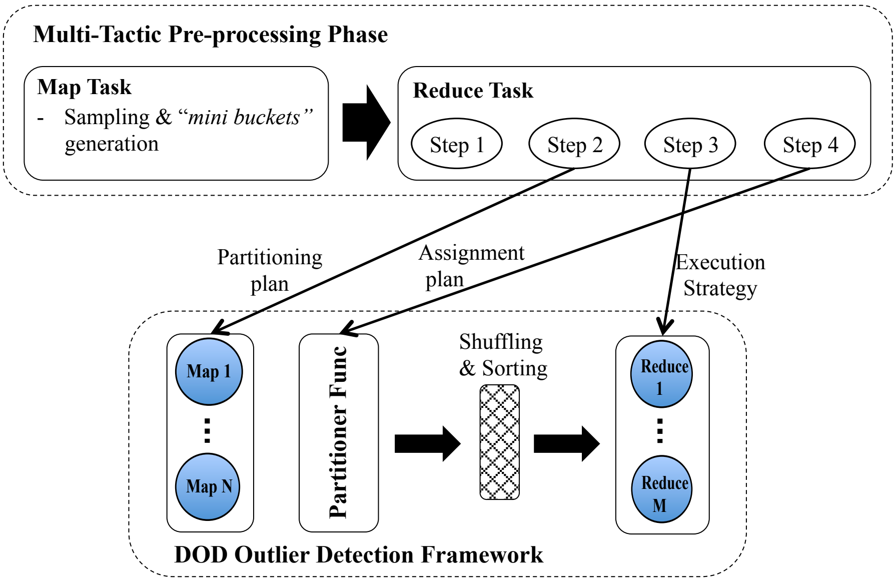


**Figure 15.3:** DOD: Execution Workflow of the Multi-Tactic DOD Framework.

## 15.2 Density-Aware Multi-Tactic Optimization

Inspired by Lemma 15.1, we propose a simplistic yet effective *density-aware multi-tactic* approach (a.k.a *DMT*) that successfully decouples the partition generation and the algorithm selection problems. In general, since the density of a partition *P* determines which algorithm performs better on *P*, partitions with similar densities should share the same selection of most appropriate detection algorithm. Therefore, *DMT* first divides the domain space into a large number of small regions (called *buckets*), and then clusters the buckets with similar densities together into larger clusters. The best detection algorithm is selected for each cluster based on its density. Next, a cost-aware partitioning algorithm is conducted by treating each cluster as one unit and estimating its detection costs using the selected algorithm. Eventually the full-fledged *DOD* framework enhanced with the *DMT* optimization effectively minimizes communication and computation costs along with a balanced workload.

**MapReduce Implementation of DMT.** Similar to the cost-aware partitioning technique introduced in Section 14.2, *DMT* executes a lightweight pre-processing job with a centralized reducer. Again, in the map phase, the map tasks output a sample from the input dataset according to the given sample rate $\Upsilon$. The only difference is that the map tasks will assume the entire data space is divided into equal size buckets, called *"mini buckets"*. The *mini buckets* then form the unit of processing. That is, the map tasks will not produce the individual sample points. Instead, they will aggregate them and produce the statistics at the *mini bucket* level.

The reduce function is slightly more complex. It involves four key steps (Figure 15.2). It receives as input the sample data in the form of *mini buckets*, the number of available reducers in the system (*numReducers*) and the number of desired partitions to be generated (*numPartitions*). The number of the desired partitions is computed from the number of available reducers multiplied by some integer number greater than 1, e.g., $numPartitions = \beta \times numReducers\ (\beta > 1)$. The intuition is that by having the number of partitions larger than the number of reducers, balancing the load among the reducers will be easier (refer to Step 4 in Figure 15.2).

In Step 1, each reducer computes the final aggregation of the *mini buckets* passed to it from the mappers. Several key statistics are computed for each *mini bucket*. These include the density measure w.r.t each dimension (Step 1, line 3) and an aggregated density measure over the entire *mini bucket* (Step 1, line 4). After Step 1, each *mini bucket* is treated as a micro-cluster. Step 2 applies a hierarchical clustering approach to group the mini buckets with similar densities into *numPartitions* final partitions such that each partition best conforms to one particular outlier detection algorithm. The clustering algorithm is similar to traditional hierarchical clustering algorithms, e.g., BIRCH [63], with the following customization: (1) our clustering algorithm takes the spatial properties of the clusters into account, i.e., only the spatially-adjacent clusters of similar densities are considered for possible expansion, (2) only rectangle-shaped clusters are allowed for a simple plan, and (3) the cardinality of each cluster is upper bounded by a threshold corresponding to the maximum number of data points that a single reducer can handle in main-memory. Mapping the memory limitation of a reducer to this cardinality constraint is straightforward, since a linear relationship exists between the memory consumption and the cardinality of the dataset.

In Figure 15.2, the reduce function tests the valid expansions for each cluster $c$ (Step 2, line 2). For each candidate expansion of $c$, a similarity measure, e.g., diameter metric [64], is estimated for the to-be-merged clusters based on their densities (Step 2, line 3). The candidate expansion with the highest score is selected as the best expansion of $c$. Finally, the best overall expansion in all existing clusters is selected and its sub-clusters are merged together (Step 2, lines 6-7). In the case of ties, the candidate expansion involving smaller clusters will be given a higher priority to balance the sizes of the clusters.

Step 2 repeats until the number of existing partitions reaches *numPartitions* (Step 2, line 8). It is worth highlighting that the computations involved in each iteration of Step 2 (after the $1^{st}$ one) are cheap. The reason is that most computations are re-usable between the different expansion operations. The only similarity measures that must be computed from scratch are those related to the newly-formed cluster. The outcome of Step 2 is the *final partitioning plan* to be used in the *DOD* framework by the mappers.

Step 3 then *decides on the execution strategy*, i.e., which outlier detection algorithm should be used for which partition. For each partition $P$ and a specific algorithm $A$, the cost of $A$ is estimated based on $P$'s properties including its size and densities (Step 3, line 3). The most efficient algorithm will be assigned to partition $P$.

The last step of this pre-processing phase (Step 4 in Figure 15.2) is then to assign the partitions to reducers. Recall that *numPartitions* is larger than *numReducers*. This gives us a degree of flexibility to make the assignments and balance the overall load. The load is measured as the sum of costs of the partitions assigned to a given reducer. This problem is equivalent to the problem of *multi-bin packing*, in which a set of $N$ numbers needs to be divided into $K$ subsets, such that the sums within each subset are as similar as possible. This problem is known to be NP-Complete. Several approximation algorithms have been proposed to solve it. In *DOD*, we adopt the polynomial-time algorithm proposed in [65].

**Overall Workflow of DOD.** In Figure 15.3, we summarize the execution workflow of the multi-tactic *DOD* framework. The workflow consists of two MapReduce jobs: the pre-processing job (top), and the outlier-detection job (bottom). The pre-processing job produces three types of outputs generated by the *density-aware multi-tactic* optimization phase (Figure 15.2). Figure 15.3 illustrates the flow of these outputs to the *DOD* framework. More specifically, the output of Step 2 (the partitioning plan) is passed to the map phase of the *DOD* framework. The output of Step 3 (the algorithm selection) is passed to the reduce phase. The output of Step 4 (the assignment plan) is passed to the partitioner function to stipulate which partitions are assigned to which reducers.

# 16

# Performance Evaluation

## 16.1 Experimental Setup & Methodologies

**Experimental Infrastructure.** All experiments are conducted on a shared-nothing cluster with one master node and 40 slave nodes. Each node consists of 16 core AMD 3.0GHz processors, 32GB RAM, 250GB disk, and nodes are interconnected with 1Gbps Ethernet. Each server runs CentOS Linux (kernel version 2.6.32), Java 1.6, Hadoop 1.0.1. Each node is configured to run up to 8 map and 8 reduce tasks concurrently. The sort buffer size is set to 512MB. Speculative execution is disabled to boost performance. The replication factor is set to 3.

**Datasets.** We utilize the *OpenStreetMap* dataset [40] to evaluate the performance of our strategy on real world application data. *OpenStreetMap* (500 GBs) is one of the largest real datasets publicly available and has been used in other similar research work [40]. *OpenStreetMap* contains the geolocation information of buildings all over the world. Each row in this dataset represents a building. Four attributes are utilized in this experiment, namely *ID, timestamp, longitude*, and *latitude*.

In order to evaluate the robustness of our proposed methods for diverse data distributions, we pick four segments from the whole dataset corresponding to buildings in Massachusetts, Ohio, California, and New York respectively. The four segments are equally sized ($\approx$30 million points). However, they vary significantly in their densities, i.e., New York and California are very dense, Ohio is relatively sparse, and Massachusetts is in the middle between them. In addition, we build hierarchical datasets with Massachusetts as the smallest unit, then New England, then the United States, up to the whole planet. The number of data points gradually grows from 30 million to 4 billion.

Lastly, to evaluate how *DOD* performs on terabyte level data we further generate a 2TB synthetic dataset based on the real *OpenStreetMap* dataset. More specifically, we developed a tool that randomly creates a distortion of the original dataset *D* by replicating each point $p$ in *D* three times to generate $p'$, $p''$, $p'''$, each with a random degree of alteration on each dimension of *D*.

**Metrics.** We measure the end-to-end execution time, which is common for the evaluation of distributed algorithms. Furthermore, we measure the breakdown of the execution time for the key stages of the MapReduce workflow including the preprocessing time, the partitioning (map) time, and the processing (reduce) time.

**Experimental Methodology.** We evaluate two key components of our MapReduce outlier detection algorithms, namely the *partitioning* method at the mapper side and the *outlier detection* method at the reducer side. In particular for the partitioning method, we evaluate four alternative strategies, namely (1) the default domain-based partitioning without supporting area *Domain*, (2) the uniform domain space partitioning *uniSpace* with supporting area (Sec. 13.1), (3) the data-driven partitioning *DDriven* (Sec. 14.1), and (4) the cost-driven partitioning *CDriven* (Sec. 14.2). Their performance is evaluated for varying sizes of datasets and diverse distributions. We use the domain-based partitioning *Domain* as the baseline approach to compare our proposed partitioning methods against. *Domain* needs an additional MapReduce job to confirm the outlier status of a point *p* if *p* is at the edge of a partition and is classified as an outlier in the first MapReduce job.

For outlier detection at the reducer side, we evaluate three alternative algorithms, namely *Nested-Loop*, *Cell-Based*, and the *multi-tactic* detection algorithm proposed in Sec. 15. *Nested-Loop* is the basic distance-based outlier algorithm widely adopted in the literature, while *Cell-Based* is the most commonly used index-driven detection algorithm. To avoid any influence of the partitioning method on the result, the same partitioning method is deployed for each, namely the data-driven partitioning *DDriven*.

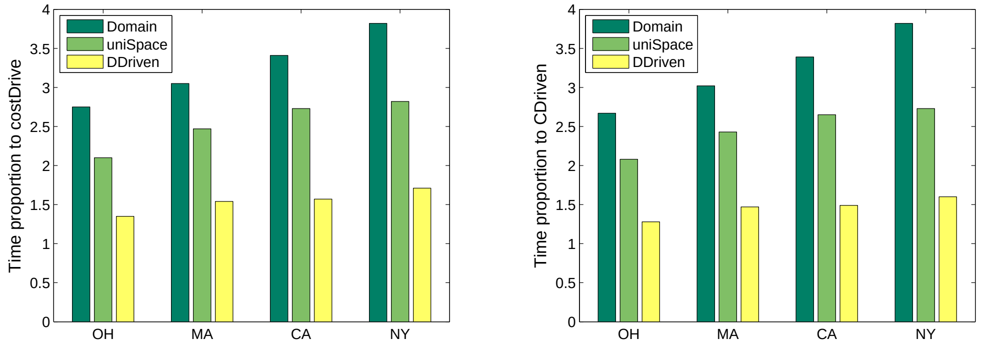
We also compare the full-fledged *DOD* solution against the state-of-the-art centralized distance-based outlier detection algorithm DOLPHIN [44] to confirm the scalability of *DOD* (with *CDriven* partitioning and *multi-tactic* detection processing).

## 16.2 Evaluation of Partitioning Methods

First we conduct experiments to evaluate the effectiveness of the partitioning methods.

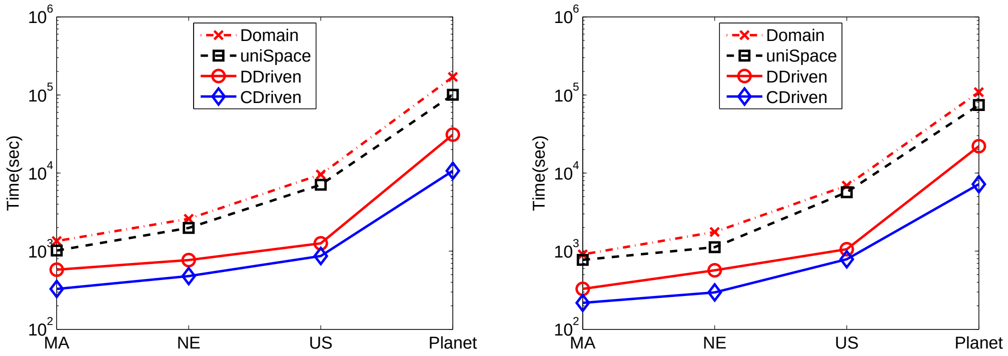
**Effectiveness Evaluation For Various Distributions With Real Datasets.** In this set of experiments we evaluate the performance of our partition methods under diverse data distributions using Ohio, Massachusetts, California, and New York areas. We show the performance of the *Domain*, *uniSpace*, and *DDriven* strategies relative to our proposed *CDriven* partitioning strategy. To exclude the influence of detection algorithm, we fixed the detecting algorithm at the reducer side to be the *Nested-Loop* solution in Figure 16.1(a) and the *Cell-Based* algorithm in Figure 16.1(b).

Clearly, as depicted in both Figures 16.1(a) and 16.1(b), the *cost-driven* partitioning method significantly outperforms all other alternatives up to 5 fold, no matter how the data distribution changes. Our *uniSpace* partitioning strategy outperforms the default *Domain* partitioning method. This is due to the fact that *uniSpace* ensures that the detection task can be done in a single pass, therefore incurring much smaller communication costs.



(a) Nested-Loop (b) Cell-Based

**Figure 16.1:** DOD Partitioning: Effectiveness Evaluation for Various Distributions.

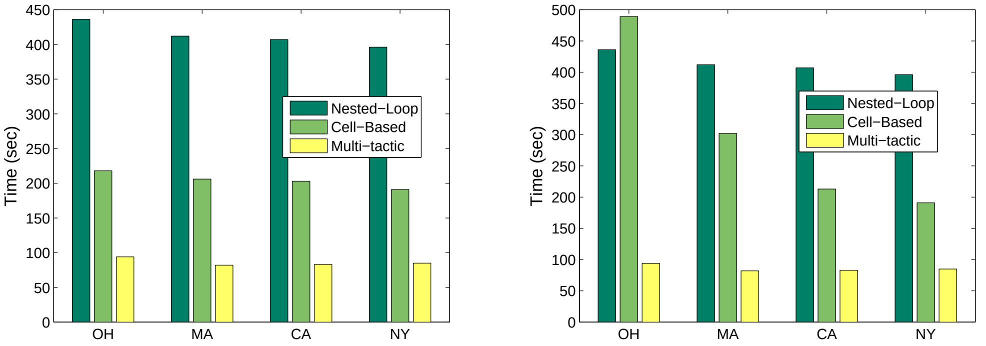


(a) Nested-Loop (log scale) (b) Cell-Based (log scale)

**Figure 16.2:** DOD Partitioning: Scalability Evaluation For Varying Data Sizes.

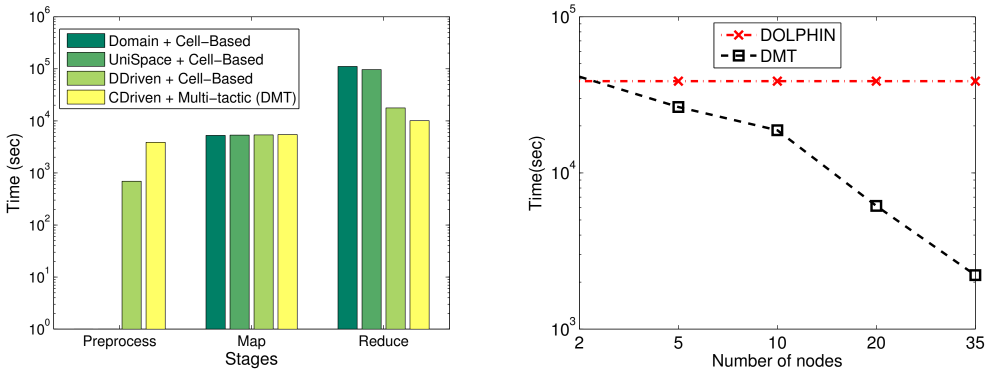
However *uniSpace* is not effective at load balancing, because the real datasets tend to be skewed. Therefore on average, the performance is 40% worse than that of *DDriven*. On the other hand, although *DDriven* ensures that each partition has a similar number of data points, the workload on each reducer is not effectively balanced, confirming the observation that an equal number of data points does not guarantee an equal workload. Our final *CDriven* partitioning strategy instead achieves true load balancing. Therefore it outperforms *DDriven* by at least 50% and all other methods more significantly (up to five fold).

**Scalability Evaluation For Varying Data Sizes.** Next we evaluate the scalability of our partitioning method on increasing dataset sizes, from the Massachusetts dataset, New England dataset, United States dataset, to the entire *OpenMapStreet* dataset. The results in Fig. 16.2 show that the *cost-driven* partitioning method *CDriven* consistently wins in all cases. *Better yet, the larger the dataset, the more it wins*. In particular when the dataset is the largest (the planet dataset), *CDriven* is 6 times faster than the second best partitioning method *DDriven* and 17 times faster than the default *Domain* partitioning. This thus demonstrates that our partitioning method is scalable to real-world large datasets.



(a) Varying Distributions (b) Varying Data Sizes

**Figure 16.3:** DOD Detection Methods: Effectiveness Evaluation.



(a) Time breakdown (b) Against centralized algorithm

**Figure 16.4:** DOD Overall Approach: Performance Breakdown & Scalability.

## 16.3 Evaluation of Detection Methods

In this section, we focus on the evaluation of the outlier detection algorithms applied at the reducer side. Three algorithms are considered in this set of experiments, namely the *Nested-Loop* and *Cell-Based* algorithms and the novel *multi-tactic* algorithm proposed in Chapter 15.

**Effectiveness Evaluation For Varying Data Distributions.** In this experiment, we utilize the Massachusetts, Ohio, California, and New York areas to evaluate the efficiency of the three detection algorithms with diverse data distributions. As shown in Figure 16.3(a), *Cell-Based* is at least two times faster than *Nested-Loop* when processing the California and New York datasets. The reason is that overall, California and New York are densely populated. As proven by Lemma 15.1, *Cell-Based* theoretically performs better than *Nested-Loop* on dense datasets. Our *multi-tactic* algorithm further outperforms *Cell-Based* by a factor of 2, because it integrates the advantages of both *Cell-Based* and *Nested-Loop*. That is, it adapts to *Cell-Based* when handling very dense or very sparse data partitions, while automatically switching to *Nested-Loop* when the density of the data partition is in the middle.

As the dataset gets sparser and sparser, the dataset contains more outliers. In this case, the pruning ability of the *Cell-Based* method becomes less effective. Therefore its running time increases dramatically. On the other hand, the running time of *Nested-Loop* increases at a more steady pace. As shown in Figure 16.3(a) *Nested-Loop* outperforms *Cell-Based* when processing the Ohio data – the most sparse out of the four. This again confirms our cost analysis in Sec. 15.1 with respect to these two typical classes of outlier detection algorithms. The running time of our proposed *multi-tactic* algorithm remains stable as the data distribution changes. It has overall much better performance in all cases.

**Scalability Evaluation For Varying Data Sizes.** In this experiment we evaluate the scalability of different detection algorithms utilizing real datasets. Similar to the scalability test in Sec. 16.2 we increase the size of the real dataset by utilizing first the Massachusetts only dataset, then New England, the United States, up to the entire *OpenStreetMap* dataset. As shown in Figure 16.3(b), our *multi-tactic* algorithm consistently outperforms *Nested-Loop* and *Cell-Based.* The larger the dataset, the more *multi-tactic* wins. This is so because the larger datasets tend to be more skewed. In other words, large data usually contains not only many sparse partitions, but also many dense partitions. However as demonstrated in Sec. 15.1 neither *Nested-Loop* nor *Cell-Based* performs well under all circumstances. Our *multi-tactic* algorithm is able to dynamically adapt to *Nested-Loop* or *Cell-Based* for each partition based on its distribution. Therefore *multi-tactic* scales well to large datasets.

## 16.4 Evaluation of Overall Approach

In this section we focus on the evaluation of the overall approach. We measure the breakdown of the execution time for all key stages of the MapReduce workflow. We also compare our approach against the centralized distance-based outlier detection algorithm DOLPHIN [44].

**The Breakdown of the Execution Time.** We measure the preprocessing time, the partitioning time, and the detection time separately. At the mapper side, four partitioning algorithms are considered, namely the baseline *Domain* partitioning and our *uniSpace*, *DDriven*, and *CDriven* partitioning strategies. At the reducer side for *CDriven* partitioning we apply the *multi-tactic* strategy for outlier detection. *CDriven* and *multi-tactic* in combination in fact constitute our final overall integrated *DMT* approach presented in Chapter 15. For the other three partitioning methods we apply the *Cell-Based* detection algorithm. This is because *Cell-Based* is confirmed by our additional experiments to be the algorithm that on average fits this dataset better than *Nested-Loop*. In this experiment we use the 2TB dataset derived from the *OpenStreetMap* dataset (see Sec. 16.1). Therefore this dataset also confirms the scalability of DOD on terabyte-scale data.

As shown in Fig. 16.4(a), the preprocessing time of *CDriven* (*DMT*) is longer than *DDriven*. This is expected because *CDriven* utilizes a hierarchical clustering approach to group data with similar densities together. This is expensive. *Domain* and *uniSpace* do not feature this preprocessing stage. Therefore no preprocessing costs are experienced. In the partitioning map stage, all four approaches take almost the same amount of time. For all, each datum can be mapped to its corresponding partition in near constant time. At the reduce stage, *DMT* significantly outperforms other alternatives up to 10 fold for the following two reasons. First, the *CDriven* partitioning of *DMT* achieves true workload balancing across the reducers by utilizing our cost models designated for each known detection algorithm. Second, given a particular partition, the *multi-tactic* detection method automatically adapts to the algorithm best fitting the data characteristics of that partition. Although this "dense" dataset on average fits the *Cell-Based* algorithm better than the *Nested-Loop* algorithm, there are still many relatively sparse partitions for which *Nested-Loop* is more appropriate.

**Comparison to State-of-the-Art Centralized Algorithm.** In this experiment, we compare our most advanced *DOD* solution (*DMT*) against the state-of-the-art centralized method DOLPHIN [44]. The centralized DOLPHIN algorithm, supported with an indexing mechanism customized for distance-based outlier detection, executes on a single node in our cluster with a maximum storage capacity of 250GB. Therefore we now use a subset of the real *OpenStreetMap* dataset with size 200GB. Using a small dataset biases the centralized algorithm, since centralized algorithms cannot support datasets larger than the available disk capacity of one machine in the best case. Hence it cannot handle very large datasets. The results shown in Figure 16.4(b) confirm the expected behavior. The centralized algorithm performs better when the distributed algorithm uses few nodes (2 nodes), because the overhead of distribution, partitioning, and communication exceed the speedup from parallel processing. In contrast, as the degree of distribution increases (to 5 nodes), the savings dominate the overhead. Our *DOD* approach beats the centralized DOLPHIN algorithm. When the number of nodes increases to 35, *DOD* is 19 times faster than DOLPHIN and 20 times faster than the 2 node case. This also confirms that our *DOD* approach is scalable in the number of compute nodes.

# 17

# Related Work

**Centralized Outlier Detection.** The concept of distance-based outliers was first proposed in [8] along with popular two detection algorithms described in Sections 14.2 and 15.1. [44] improved upon these prior results [8] by introducing the pivot-based index technique. However, this technique depends on building a global index. This thus does not fit well the popular distributed shared-nothing architectures such as MapReduce because the overhead of building and then sharing the index among different machines can be prohibitively high.

**Distributed Outlier Detection.** Hung and Cheung presented a parallel version of the basic *Nested-loop* algorithm for distance-based outlier detection [66]. However, this technique requires synchronization between the worker nodes. Thus, it is not suitable for MapReduce-like infrastructures where mappers (and similarly reducers) work independently from each other.

Angiulli et al. [35] presented a distributed detection algorithm to support KNN based outlier definition [9]. First, it represents the original dataset using a compact solving set. Then given a data point $p_i$, its status as an outlier can be approximated by comparing $p_i$ to only the elements in the solving set. Therefore [35] provides an approximate result, whereas we instead focus on providing an exact solution for distance-based outlier detection.

Bhaduri et al. [36], also working with the KNN based outlier definition, developed a distributed algorithm on a ring overlay network leveraging a multi-core cluster of machines. Their algorithm passes data blocks around the ring allowing the computation of neighbors to proceed in parallel. Along the way, each point's neighbor information is updated and distributed across all nodes. A central node is utilized to maintain and update the top*n* points with largest *k* nearest neighbor distances. Checking the test blocks in a round robin fashion, which requires *m* iterations, is not suitable for map reduce. Furthermore, MapReduce also does not feature central node.

The distributed outlier detection algorithms presented by Otey et al. [38] and Anna et al. [37] focus on very domain specific outlier definitions instead of the more general notion of distance-based outlier targeted by our work. The first algorithm utilizes the dependencies among all attributes for tackling *mixed-attribute* data, while the second uses value frequency to tackle *categorical data*. Unlike distance-based outliers, such techniques do not utilize *distance* between each pair of data points to detect outliers. Thus their techniques cannot be applied to solve our problem.

**Advanced Analytics in MapReduce.** The efficiency of our *DOD* approach relies on important strategies in distributed systems such as *load balancing*, *efficient partitioning* and *sampling*. These concepts have also been discussed in the MapReduce context with respect to other advanced analytics tasks.

The problem of performing similarity joins using MapReduce has attracted significant attention. In [39], the authors proposed the *MR-MAPSS* algorithm which partitions the input data into work sets with minimal redundancy. It achieves *load balancing* by repartitioning the densely clustered large work sets. However their *load balancing* method relies on the traditional load balance assumption, namely an equal number of data points indicating equal work load. This assumption is proven to be not true for distance-based outlier detection (see Sec. 14.2).

Research work also has been done for KNN-Joins on MapReduce. The approximation algorithm in [40] first maps the multi-dimensional datasets into one dimension using space-filling curves (z-values), and then transforms the KNN join into a sequence of one-dimensional range searches. This way, the *partitioning* of a multi-dimensional dataset is reduced to an equal size one-dimensional partitioning problem. This technique cannot be applied to our context. First, we focus on producing exact and complete results of distance-based outliers instead of approximation. Second, this approximation method is shown to be not suitable for skewed data, while outlier detection usually handles skewed data [18].

[14] studies density-based clustering on MapReduce. Although density-based clustering is the clustering definition most closely related to distance-based outliers, this approach cannot be directly applied to solve our outlier detection problem. This is due to the fact that inroducing outliers as by-products of clustering has already been shown not to be effective in capturing abnormal phenomena [18]. Furthermore, density-based clustering has been shown to be more expensive than distance-based outlier detection [48], because the cluster structure is more complex to detect and update than the individual outlier points due to the inter-dependence among data points. Therefore applying the density-based clustering algorithm to detect outliers is neither effective nor efficient.

In particular the BoW method [14] supports density-based clustering over MapReduce. It focuses on minimizing the I/O and networking cost among all processing nodes. In order to reduce the network traffic cost, a specific *sample phase* is introduced to generate initial clusters. Our proposed methods share this idea. However in our work the sampling is only utilized to to estimate the cost of each partition in the pre-processing phase.

# Part IV

# Distributed Outlier Detection: LOF

# 18

# The Distributed LOF Approach

As shown in Sec. 2.2 computing $LOF$ requires a number of steps. The intermediate $k-distance$ and $LRD$ values must be determine for all points before the $LOF$ score can be calculated.

In the centralized $LOF$ implementation[67], first the k-nearest neighbors (*k*NN) of each point are computed using an index structure to achieve $O(nlogn)$ complexity for this step. The neighbors and $k-distance$ of each point are then materialized in an $n*k$ global database table. Two passes are made over the database to compute the $LRD$ and $LOF$ values. In each of these passes the intermediate values for the neighbors of each point are updated, maintained in the global data table, and then utilized in the next step of the computation.

Applying the above centralized approach in the shared nothing distributed architecture is not practical. First, when computing the (*k*NN) of one given point the centralized approach assumes it has the access to all data points. However in the distributed setting the points in the dataset are distributed among different machines according to some partitioning plan. Within each node in the compute cluster, only part of the data can be accessed. Therefore very possibly the *k*NN of one point is located in other machines.

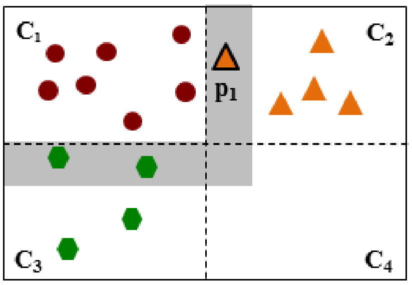
Therefore a distributed *k*NN search approach must be designed.

Second, even if the *k*NN of each point can be computed, it is not feasible to store the neighbors and $k - distance$ of all points in one single compute node as a database table considering the size of the big dataset. Therefore such intermediate values have to be stored in different compute nodes. This inevitably complicates the next two steps of LOF computation, namely the computation of *LRD* and *LOF* values. For example given a point *p* when computing its LRD value, we have to first locate the compute nodes that store its *k*NN and then retrieve such information back from the corresponding tables. Therefore distributed mechanism has to be designed for the efficient maintenance and retrieval of intermediate results. This is challenging, since the shared nothing architecture does allow data exchange at will.

To solve these problems in this section we propose the first distributed LOF approach or in short *DLOF* that conducts each step of LOF computation in a highly distributed fashion. *DLOF* features two key strategies, namely the *supporting area* partitioning and *localized intermediate data maintenance*.

**Supporting Area Partitioning**. As shown above the design of a partitioning plan which facilitates a *k*NN search is a key element in the distributed $LOF$ approach. Intuitively, a good partitioning plan must take the locality of each data point into consideration. In other words the data points close to each other should be grouped together. Therefore as a first step, we can partition the entire domain space of a dataset $D$ into *n* disjoint cells $C_i$ such that $C_1 \cup C_2 \cup ... \cup C_n = D$. Then points are grouped based on their membership in a cell. However, even with nearby points grouped together, the neighbors of some points, particularly those that lie along the boundary of each partition, may have nearest neighbors mapped to another partition. For instance, in Figure 18.1 the point $p_1$ in cell $C_2$ may be the neighbor of points in cell $C_1$.

To solve this *boundary problem* we introduce the concept of *supporting area* parti-



Point $p_1$ will be mapped to both cell $C_1$ as a supporting point, and cell $C_2$ as a core point.

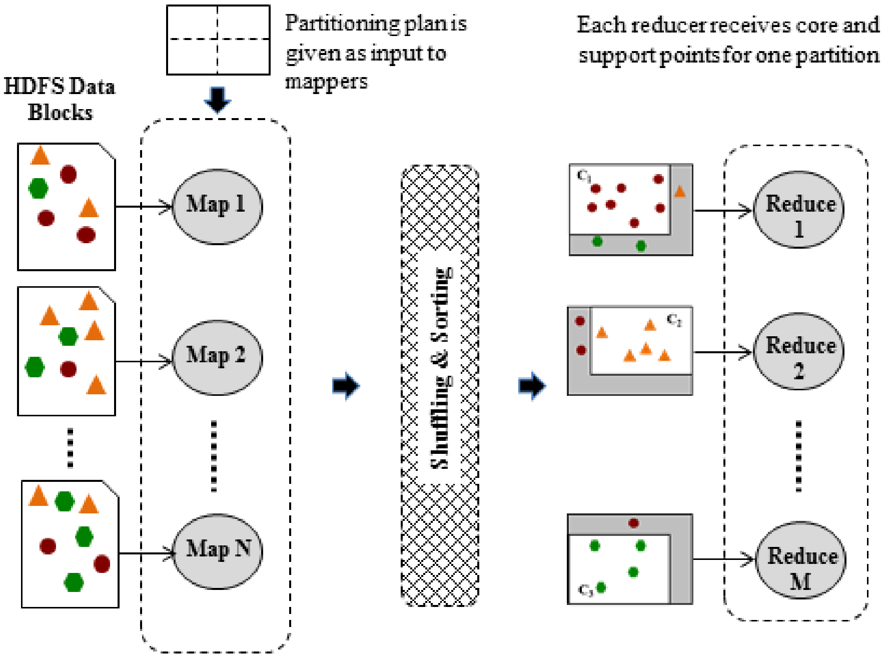
**Figure 18.1:** DLOF: Supporting Area of Partition $C_1$ .

tioning, as formally defined in Def. 18.1. The points inside a cell $C_i$, referred to as the $C_i$'s **core points**, are denoted as $C_i.core = \{p_j \mid p_j \in C_i\}$. The data points within the *supporting area* of cell $C_i$ may affect the *k*NN decision of at least one core point of $C_i$.

**Definition 18.1** ***Supporting Area.*** *The* ***supporting area*** *of a grid cell $C_i$, denoted as $C_i.suppArea$, is an extension of the boundaries of $C_i$ in each dimension of $D$. All data points $p_j$ (also called support points)$\in$ $C_i.suppArea$ satisfy the following two conditions: (1) $p_j \notin C_i.core$, and (2) there exists at least one point $p_k \in C_i.core$ such that dist($p_j$, $p_k$) $\leq k - distance(p_k)$.*

This strategy categorizes data points into two classes, namely *core points* and *support points*, ensuring that each point $p$ will have all information within its local partition needed to find it's nearest neighbors. Figure 18.1 highlights in grey the supporting areas of cell $C_1$. Each cell $C_i$ will be augmented with its support points in addition to its core points. For example, $C_1$ will be extended to contain *support points* along with its circle-shaped *core points*. Given a point it will only be the core point of one

Clearly the key of the supporting area partitioning strategy is to define the boundary of the supporting area for each cell. In the next two sections we will discuss two alternative solutions in depth. In the rest of this section, we treat the partitioning solution as a black box for the simplicity of presentation.



HDFS Data Blocks
Partitioning plan is given as input to mappers
Map 1
Map 2
Map N
Shuffling & Sorting
Each reducer receives core and support points for one partition
C1
C2
C3
Reduce 1
Reduce 2
Reduce M

**Figure 18.2:** DLOF: Data Point Close to the Partition Boundary.

**Localized Intermediate Data Maintenance** Given a solution to the partitioning problem, the $k - distance$ of each point in the dataset can be determined as a first step in the distributed $LOF$ algorithm. Figure 18.2 shows how the partitioning plan is used in the execution of a single MapReduce job.

Initially, data points may be scattered among different HDFS data blocks. These blocks are input to mappers, which assign the points to cells, and label each point as either a *core point* or *support point*. Then, each reducer receives one cell as input. Within that cell the *k*NN of each *core point* is found and used to determine its $k - distance$. The $k - distance$ value is then used in the next step of the algorithm to compute the *Local Reachability Density* ($LRD$). For each point $p$, computing the $LRD$ depends not only on the $k - distance$ of $p$, but also the $k - distance$ of each neighbor of $p$.

Herein lies another challenge. Although the *supporting area* strategy allows us to identify the neighbors of each *core point* within a given partition, some of these neighbors may be *support points*, which have their $k - distance$ computed in a different partition. Therefore, the information necessary to compute the $LRD$ of the *core points* is not available because of lacking global information about the dataset.

To resolve this problem we introduce our *localized intermediate data maintenance strategy*. First, each point is assigned a unique *ID* as its identity. Then each reducer writes back its core points along with the IDs of their *k*NN and $k-distance$s to its local HDFS. Furthermore, at the same time given a core point, its partitioning related information will also be written out such as which partition it is assigned to as core point and in what partitions it is classified as support point.

Then in the next step for the computation of *LRD*, mappers read in the new points and assign them to the corresponding partitions based on the partitioning information embedded in each point. Each reducer then map the received points (both core points and support points) to a hash table with the *ID* as key and $k-distance$ as value. Now for each core point p, we have sufficient information to calculate its *LRD* even if its neighbors are support points, since each support point now also has the $k-distance$ associated with it. Furthermore, the $k-distance$s of p's neighbors can be located in constant time utilizing the hash table.

Therefore this localized intermediate data maintenance strategy, although intuitive, successfully solves the problems caused by the lack of global information about the dataset in the distributed setting.

This process of calculating values for each point and localized maintenance of intermediate results must be repeated again, because the $LRD$ value is required for each neighbor of a *core point* to finally compute the $LOF$ score.

Algorithm 8 gives a general approach to distributed $LOF$, comprised of the three steps outlined above corresponding to three separate MapReduce jobs, where the output of each is the input to the next. Figure 18.3 illustrates this framework. The approach correctly discovers outliers, and represents the solid foundation upon which key optimization

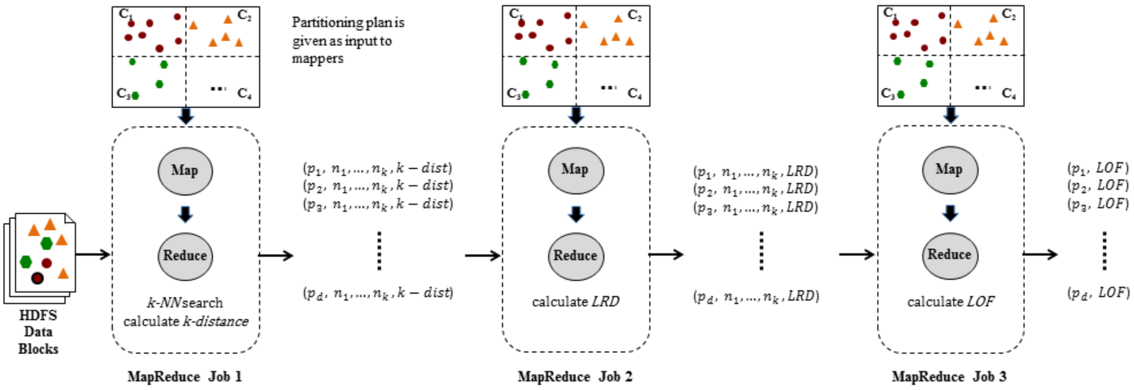

C1
C2
C3
C4
Partitioning plan is given as input to mappers
HDFS Data Blocks
Map
Reduce
k-NN search
calculate k-distance
MapReduce Job 1
$(p_1, n_1, \ldots, n_k, k-dist)$
$(p_2, n_1, \ldots, n_k, k-dist)$
$(p_3, n_1, \ldots, n_k, k-dist)$
$(p_d, n_1, \ldots, n_k, k-dist)$
Map
Reduce
calculate LRD
MapReduce Job 2
$(p_1, n_1, \ldots, n_k, LRD)$
$(p_2, n_1, \ldots, n_k, LRD)$
$(p_3, n_1, \ldots, n_k, LRD)$
$(p_d, n_1, \ldots, n_k, LRD)$
Map
Reduce
calculate LOF
MapReduce Job 3
$(p_1, LOF)$
$(p_2, LOF)$
$(p_3, LOF)$
$(p_d, LOF)$

**Figure 18.3:** DLOF: Overall Process With Multiple MapReduce Jobs.

strategies are then introduced.

**Algorithm 8** Baseline Distributed LOF Algorithm

```
$s \leftarrow partitioning\ strategy$
$k \leftarrow number\ of\ nearest\ neighbors$
Function {Map1(key k, value p) }
output: $(s(p), p)$
map each point to partition
Function {Reduce1(key part-id, val-list [$p_1$,..., $p_m$]) }
for each point $p \in$ val-list do
    find $k$ neighbors
    output: ($p$, $\{n_1, \ldots, n_k, k - dist\}$)

Function {Map2(key $p$, value $\{n_1, \ldots, n_k, k - distance\}$) }
output: $(s(p), \{p, n_1, \ldots, n_k, k - dist\})$
partition
Function{Reduce2(key part-id, val-list [$p_1$,..., $p_m$]) }
for each point $p \in$ val-list do
    find $k$ neighbors
    compute $LRD(p)$
    output: ($p$, $\{n_1, \ldots, n_k, LRD(p)\}$)

Function{Map3(key $p$, value $\{n_1, \ldots, n_k, LRD(p)\}$) }
output: $(s(p), \{p, n_1, \ldots, n_k, LRD(p)\})$
partition
Function{Reduce3(key part-id, value-list [$p_1$,..., $p_m$]) }
for each point $p \in$ val-list do
    find $k$ neighbors
    compute $LOF(p)$
    output: ($p$, $LOF(p)$)
```

# 19

# Pivot-Based Distributed LOF

As shown in Chapter 18 the key of the *DLOF* approach is to design an effective partitioning method that divides the data set into cells (containing core points), and then define the supporting area for each cell (support points) such that each compute node can find the *k*NN for the core points assigned to it, independent of other nodes.

Before presenting our partitioning method, we first introduce the *duplication rate* metric that is used to measure the efficiency of the generated partitioning plan.

The cost of a typical MapReduce algorithm is usually determined by two factors, namely communication and computation costs. The communication costs correspond to the costs of transmitting data from mappers to reducers. Often, if not always, the communication costs are the dominant costs of a MapReduce job [60]. Similar to the communication costs, the computation costs (especially those of the reducers) are also directly related to the number of the data points transmitted from mappers to reducers, i.e., the more data points that are received by the reducers, the more computational work is performed by them.

**Definition 19.1** ***Duplication Rate (dr).*** *For a dataset $D$ and a MapReduce algorithm $A$ for computing $LOF$ for all points in $D$, the "duplication rate" dr(D, A)* $\in [1, \infty]$

*represents the average number of duplicates that the mapper phases of $A$ generate per data point $p_i \in D$.*

Using the *supporting area* partitioning strategy, each data point has to be transmitted at least once from mappers to reducers, as a *core point* of one cell, and possibly many more times, as a *support point* of other cells. Therefore, the efficiency of a partitioning method can be modeled using the notion of *"Duplication Rate"*, which refers to the average number of duplicates mappers need to create for each input data point. The larger the duplication rate, the more data points must be transmitted from mappers to reducers, and thus the higher the communication and computation costs. The duplication rate is defined next.

Obviously if we assign the whole data set to each partition as its supporting area, each reducer is able to independently discover the *k*NN for its core points. However the duplication rate will be extremely high. Therefore this is not a practical solution. Next we present our pivot-based partitioning method that yields a lower duplication rate than this naive approach by only including support points that have the potential to be in the *k*NN of the core points in a given cell. Applying this partitioning method in our $DLOF$ approach, our pivot-based partitioning based LOF approach (PDLOF) provides the first full-fledged distributed LOF solution.

## 19.1 Pivot-Based Partitioning

The central idea of pivot-based partitioning is to divide the dataset by choosing a small set of $n$ initial points, or $pivots$, from the domain space in a pre-processing step. Each input point in the dataset is then assigned to its closest pivot. Grouping data according to their proximity to these $n$ *pivots* results in a division of the data space into *n* disjoint *Voronoi cells*. This division comprises a unique partitioning of the domain space known as a Voronoi diagram, which is depicted in Figure 19.1 for $n = 5$ pivots. A formal definition of a Voronoi cell is as follows:

**Definition 19.2** ***Voronoi Cell*** *Given a dataset $D$ and a set of pivots $P = \{p_1, p_2, \ldots, p_n\}$ we have $n$ corresponding Voronoi cells $V_1 \ldots V_n$ where $V_1 \cup V_2 \cup \ldots \cup V_n = D$. If $i \neq j$, $V_i \cap V_j = \emptyset$ and*

$$V_i = \{q \mid distance(q, p_i) \leq distance(q, p_j)\} \, \forall q \in D, i \neq j$$

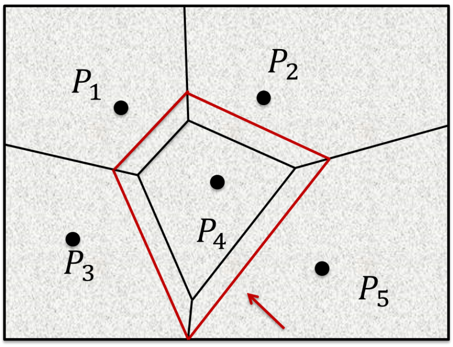


**Figure 19.1:** DLOF: Voronoi Diagram-Based Partitioning

By partitioning data into Voronoi cells, nearby data points are grouped together. Therefore the locality of the data points are preserved. However still the nearest neighbors of some points may fall in other partitions such as the points at the edge of each Voronoi cell. A *supporting area* is required to determine the *k*NN of such points as shown in Fig. 19.1.

A major benefit of using a pivot-based strategy is that in the process of partitioning the data we can learn information about each cell, namely, the distance from each point to the pivots. This information can be utilized to derive bounds on the possible distance from any point in a partition to it's neighbors, and therefore a bound on the $k - distance$ of all points in the cell. This bound can then can be utilized to determine which points must be included in the *supporting area* of the cell.

To establish this bound we first introduce the upper bound on the distance from one point in a Voronoi cell $V_j$ to any point in a Voronoi cell $V_i$.

**Definition 19.3** *Given a Voronoi cell $V_i$ with pivot $p_i$, the upper bound on the distance from one point $s \in V_j, i \neq j$ to any point $\in V_i$ denoted as $ub(s, V_i)$:*

$$ub(s, V_i) = maxdist(V_i) + distance(p_i, p_j) + distance(p_j, s)$$

*where $maxdist(V_i)$ is the greatest distance from the pivot of $p_i$ to any point within its cell $V_i$.*

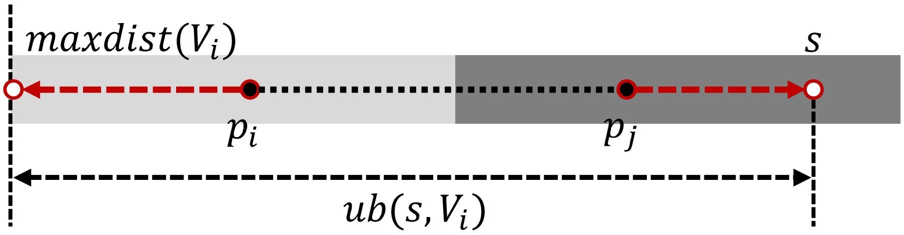


**Figure 19.2:** DLOF: Upper Bound On K-distance For Points in Partition $V_i$

The geometric meaning of this bound is illustrated in Figure 19.2. Intuitively given one point $t$ in a Voronoi cell $V_i$, in the worst case its distance to one point $s$ in another Voronoi cell $V_j$ $ub(t, s)$ is the distance between the pivot $p_i$ of $V_i$ and the pivot $p_j$ of $V_j$ plus the distance between $t$ and its pivot $p_i$ and the distance between $s$ and its pivot $p_j$. This worst case happens only when: (1) the *t*, $p_i$,$p_j$, and *s* can be connected by one straight line and t,s; (2) $s$ and $t$ are located at the opposite side of their corresponding pivots. Then the upper bound on the distance from $s \in V_j$ to *any* point in $V_i$ $ub(s,\, V_i)$ is $ub(t_{max}, s)$ where $t_{max}$ is the furthest point to pivot $p_i$ in Voronoi cell $V_i$. The formal proof can be found in [68].

Calculating this upper bound is straightforward if for each Voronoi cell we track and maintain the point which is furthest to its pivot during the processing of pivot partitioning.

Utilizing this bound, we can derive an upper bound $\phi$ of the *k-distance* for all points in a Voronoi cell $V_i$.

First, we find a set of k points $\mathbb{S}_j$ in each Voronoi cell $V_j$ with the smallest upper bound distances to all points in Voronoi cell $V_i$. By definition 19.3 these k points in fact corresponds to the *k*NN of pivot $p_j$. Similar to the furthest point to $p_j$, these k points can be discovered and maintained in the partitioning process.

Second, after we acquire the *k * (n-1)* points $\mathbb{S}$ from the n -1 Voronoi cells (excluding $V_i$ itself), we find the *k* points from $\mathbb{S}$ $\{s_1 \ldots s_k\}$ with the smallest $ub(s_i, V_i)$ as the $k$NN of all points in $V_i$ denoted as $KNN(V_i)$. Then the upper bound *k-distance* can be determined by Lemma 19.1

**Lemma 19.1** ***Upper Bound k-distance*** *For each Voronoi cell $V_i$, the upper bound k-distance $\phi_i$ for all points t $\in$ $V_i$ is given as:*

$$\phi_i = \max_{\forall s \in kNN(V_i)} \|ub(s, V_i)\|$$

Intuitively for any point *t* in $P_i$ we can find at least k points around *t* within distance range $\phi_i$, since in this range it includes at least the k points in $KNN(V_i)$. Naturally the actual *k*NN of *t* would not be out of this scope. Thus the distance by which each cell must be extended to include *support points* can be safely bounded by $\phi_i$. $\phi_i$ can be calculated almost for free if the furthest point and the $k$NN of each pivot $p_i$ are maintained in the partitioning process as discussed above.

Note in step (1) in fact the k points $\mathbb{S}_j$ found in one Voronoi cell $V_j$ is already sufficient to bound the $k$NN of all points of Voronoi $V_j$, since utilizing the largest $ub(s \in \mathbb{S}_j,\ V_i)$ is also able to cover at least k points around any point t in $V_i$. However in step (2) we further tighten this bound by defining $KNN(V_i)$.

Next we show how to utilize $\phi_i$ to determine whether a point *t* in Voronoi cell $V_j$ is a supporting point of a given Voronoi cell $V_i$. The evaluation rule is defined in Lemma 19.2.

**Lemma 19.2** *Given a Voronoi cell* $V_i$*, the necessary condition that a point* $s \in V_j$ *be assigned to the supporting area of* $V_i$ *is:*

$$distance(s, p_i) - maxdist(V_i) \leq \phi_i$$

Here we give an intuitive proof of Lemma 19.2. The distance between point *t* and any point in $V_i$ is guaranteed to be larger than $dist(s, p_i) - maxdist(V_i)$ (1). If (1) is larger than $\phi_i$, then s would not be the $k$NN of any point in $V_i$ by Lemma 19.1. Therefore Lemma 19.2 holds.

**Overall Approach.** In the pre-processing phase each point in HDFS is assigned to a Voronoi cell as a core point. In this phase some statistics information including $maxdis(V_i)$ and $k$NN of $p_i$ is also collected and utilized to compute $\phi_i$, namely the upper bound on the $k - distance$ of all points in each Voronoi cell $V_i$. In the map phase of the second mapReduce job Lemma 19.2 is applied to evaluate whether a core point in $V_i$ should be mapped to one or more supporting areas of other cells. This ensures that at the reduce phase each voronoi cell is self-sufficient to discover the $k$NN of all its core points. The $k$NN search is then conducted independent of other reducers.

## 19.2 Pivot-Based kNN search

Finding $k$NNs in each partition is a quite expensive process. Although the $k$NN search is completely parallelized in our PDLOF approach, it is still shown as the bottleneck of the whole LOF computation process. To mitigate this problem here we propose a novel $k$NN search algorithm that fully utilizing the information collected during the Voronoi-based partitioning process.

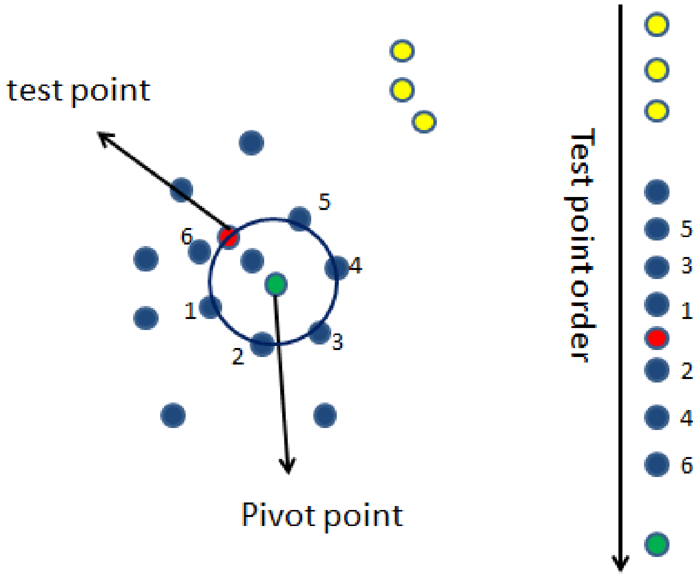


**Figure 19.3:** DLOF: Pivot-Based Index.

In each partition, we rearrange data points according to their distance to the pivot $R$. Figure 19.3 shows the index for a dataset. The left part shows the original dataset. The green circle at the center is the pivot point. The right part of the figure shows the index. The index is simply a one dimensional list of ordered points where the ordering is determined by the distance of all the points in the dataset to the pivot point(highest to lowest). Instead of traversing through the data in the original order, we test the points along this index order(up and down). With this index, we can discover the $k$NN for the testing point without traversing through the whole data set by utilizing the following termination criterion.

**Lemma 19.3** *[stopping rule] Let $R$ be the pivot points used to build index. Let $T$ be the test point that we want to calculate its $k$NN and $T_k$ be the k-distance(k-th nearest* *neightbor of $T$ currently). $L_i$ is the point that currently checked. If*

$$\Big| \| T - R \| - \| L_i - R \| \Big| > T_k$$

*then we can terminate the search immediately, where $\| \, . \, \|$ denotes the distance measurement and $| \, . \, |$ denotes the absolute value.*

*Proof:* By applying triangle inequality for the testing point $T$ and $L_i$, it follows that,

$$\| T, L_i \| > \Big| \| L_i - R \| - \| T - R \| \Big|$$

Therefore, it is obvious that if

$$\Big| \| T - R \| - \| L_i - R \| \Big| > T_k, then \| T, L_i \| > T_k$$

Therefore, we cannot find any point that is closer to $T$ than the current kth nearest neighbor of $T$. Therefore the $k$NNs of $T$ can be terminated immediately. ■

# 20

# Data Driven Distributed LOF

Although the pivot-based partitioning method introduced in Chapter 19 ensures that each compute node can perform the $k$NN search in parallel independent of other nodes, it still has fundamental drawback. Using the worst case estimation to compute the $k$NN upper bound for each Voronoi cell leads to a rate of data duplication that is still quite high. This is also confirmed by our experimental evaluation. Due to the prohibitive duplication rate, the pivot-based method cannot scale to handle really large datasets.

To overcome the high duplication rate drawback of the above *PDLOF* approach utilizing the pivot based partitioning, in this section we propose a further optimized distributed LOF approach based on a novel data driven partitioning method so called *DDLOF*. The key idea of *DDLOF* is to *dynamically* determine the boundary of the supporting area for each partition during the $k$NN search process rather than deciding the supporting area beforehand based on the *worst case* estimation using the triangle inequality.

**Convergence Property.** The effectiveness of the *DDLOF* approach is originated from our crucial observation on the property of $k$NN, namely the *convergence property*. More specifically, given a data set *D* suppose the points in *D* are divided into a set of disjoint data partitions based on their domain values. Each partition *P* contains similar number of data points. Then given any partition $P$ most of the points in $\mathbb{P}$ named $D_P$ are the $k$NNs of each other. In other words, although theoretically the $k$NNs of $D_P$ named $kNN(P)$ could be far away from $P$ and spread into the entire domain space of $D$, in reality $kNN(P)$ tends to converge and be located in $P$ itself.

Leveraging this property *DDLOF* significantly outperforms *PDLOF* in duplication rate and in turn the overall end to end execution time as confirmed in our experiments (Chapter 21).

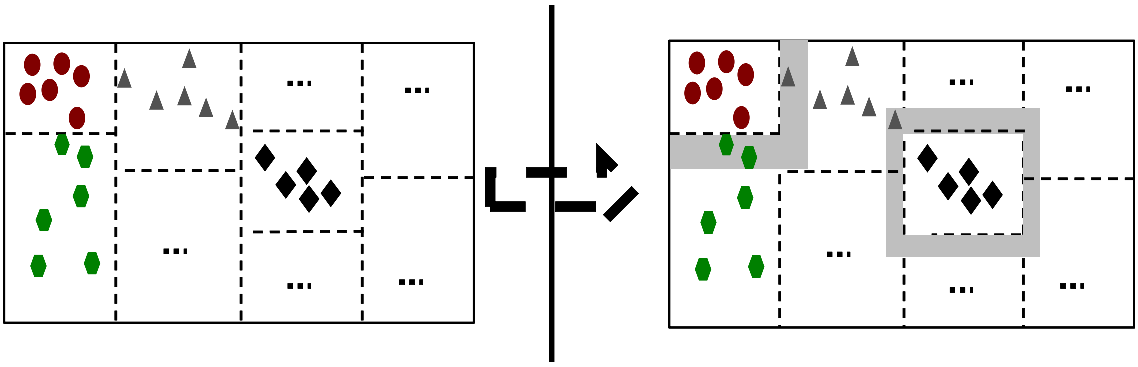


**Figure 20.1:** DDLOF: Data-Driven Distributed LOF.

As shown in Fig. 20.1 overall the *DDLOF* approach is composed of the following three steps:

- **(1) Data Driven Partitioning;**
- **(2) Inner Partition $k$NN Search;**
- **(3) Outer Partition $k$NN Search;**

In step (1) *data driven partitioning* partitions the input data set $D$ into a set of partitions $\mathbb{P}$. Each partition contains similar number of points. This not only ensures the load balancing across different reducers, but also guarantees that each partition has more than *k + 1* points.

In step (2) *inner Partition kNN search* since each partition$\mathbb{P}$ contains at least *k + 1* points, each point $p_i$ can locate k points in $\mathbb{P}$ that are closest to $p_i$ so called local $k$NN of

$p_i$. This local $k$NN is then attached to point $p_i$ and spilled out to HDFS. The k-distances of such local $k$NNs then can be utilized to bound the supporting area of $\mathbb{P}$;

In step (3) outer partition $k$NN search first extends each partition $\mathbb{P}$ to include the supporting area defined in step (2) and then discover for each point $p_i$ in $\mathbb{P}$ the actual $k$NN by searching through the points in supporting area.

Next we discuss each step in detail.

## 20.1 Data Driven Partitioning

In this section, we introduce our data-driven partitioning method or in short *DDriven*. *DDriven* partitions the entire domain space of a dataset *D* $Domain(D)$ into *n* disjoint grid cells $C_i$ such that $C_1 \cup C_2 \cup ... \cup C_n$ = $Domain(D)$. A grid cell is formally defined below:

**Definition 20.1** ***Grid Cell.*** *A grid cell $C_i$ in a d-dimensional domain space is a hyper rectangle $C_i = \langle (low_{1i}, high_{1i}),\ (low_{2i}, high_{2i}), ..., (low_{di}, high_{di}) \rangle$, where $(low_{xi}, high_{xi})$ are the boundaries of $C_i$ in the $x^{th}$ dimension, where $1 \leq x \leq d$.*

The areas of the domain space covered by each grid cell may or may not be of equal size. In general, any partitioning strategy could be utilized to produce such grid cells. *DDriven* generates grid cells that, in spite of having different grid sizes, contain a *similar number of data points* (*equi-cardinality* cells). Figure 20.1 depicts a two-dimensional space partitioned into grid cells using *DDriven*. Points are grouped based on their membership in a particular grid cell. Fig. 20.1 shows such a grouping by representing the points in each grid cell with the same shape.

*DDriven* relies on a lightweight pre-processing strategy to determine a plan for partitioning the domain space into grid cells, henceforth called the *partitioning plan*. This new pre-processing phase is composed of two steps, namely *distribution estimation* and *partitioning-plan generation*. These two steps can be performed in one MapReduce job.

In the first step, *DDriven* estimates the distribution of the data by drawing a sample from the input dataset. We opt for random sampling since it preserves the distribution of the underlying dataset [62]. Since we only need to roughly estimate the distribution, the sampling rate $\Upsilon$ as an input parameter by default is set to a small value, e.g., 0.5 %. Considering the size of big datasets, the sample is generated in a distributed fashion, for example by drawing samples within the map phase of a MapReduce job. Then, the mappers' output is passed to a centralized node, i.e., a single reducer, for the plan generation step. Since drawing the random sample at the map phase is intuitive [62], we ignore the details here.

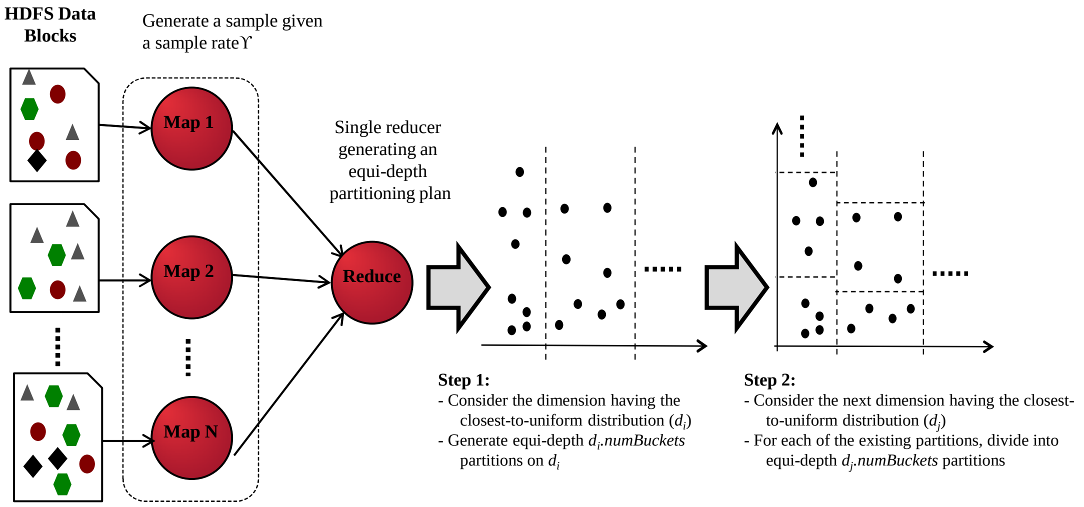

HDFS Data Blocks
Generate a sample given a sample rate ϒ
Map 1
Map 2
Map N
Single reducer generating an equi-depth partitioning plan
Reduce
Step 1:
- Consider the dimension having the closest-to-uniform distribution ($d_i$)
- Generate equi-depth $d_i$.*numBuckets* partitions on $d_i$
Step 2:
- Consider the next dimension having the closest-to-uniform distribution ($d_j$)
- For each of the existing partitions, divide into equi-depth $d_j$.*numBuckets* partitions

**Figure 20.2:** DDLOF: Data-Driven Partitioning Strategy (*DDriven*).

In the next step, the partitioning plan is generated by the single reducer (Figure 20.2). In addition to the sample data, the reducer receives a list that specifies the number of desired partitions in each dimension. For instance, $d_i.numBuckets$ is the number of desired partitions for dimension $d_i$. This list is calculated based on the number of reducers *n* assigned to the outlier detection task. By default, each dimension has the same number of partitions, with $d_i.numBuckets$ set as $\sqrt[d]{n}$ for a d-dimensional dataset.

The reducer computes for each dimension $d_i$ a *uniformity score*, i.e., a chi-square test score that measures whether or not the values on $d_i$ are close to a uniform distribution. The smaller the score $d_i.score$, the more uniformly the points are distributed over $d_i$. Hence the better it is to start with this dimension in partitioning the dataset. The dimensions are sorted in an ascending order according to their chi-square test score. The $1^{st}$ dimension is selected to be partitioned first. Then the subsequent dimension having the next-smallest score, say $d_x$, is selected for further partitioning (lines 7-11). This is performed by considering each of the existing cells, and dividing that cell over $d_x$ into $d_x.numBuckets$. Using this strategy, although the distribution of the dataset as a whole may be skewed, the subsets falling into each grid cell tend to be relatively uniform.

Figure 20.2 illustrates an example of the partitioning process. We assume that the x-axis is the $1^{st}$ selected dimension. Thus, the points are divided vertically into $d_x.numBuckets$ equi-cardinality cells. The next selected dimension is the y-axis. Each of the existing vertical partitions will be further divided into $d_y.numBuckets$ equi-cardinality cells. This process proceeds until all dimensions are partitioned.

## 20.2 Inner Partition $k$NN Search

The *partitioning plan* generated at the pre-processing phase is then passed to the *inner partition kNN search step* which corresponds to another MapReduce job. Its map phase utilizes the *partitioning plan* to divide the complete input dataset into multiple grid cells. The grid cells are then assigned and transmitted to the reducers. At the reduce phase each reducer calculates the local $k$NNs for each point assigned to this reducer.

For the point located at the edge of each cell such as $p_1$ in cell $C_1$ there might be points in its adjacent cells such as $C_5$ that are closer to $p_1$ than the points in $C_1$. Therefore the

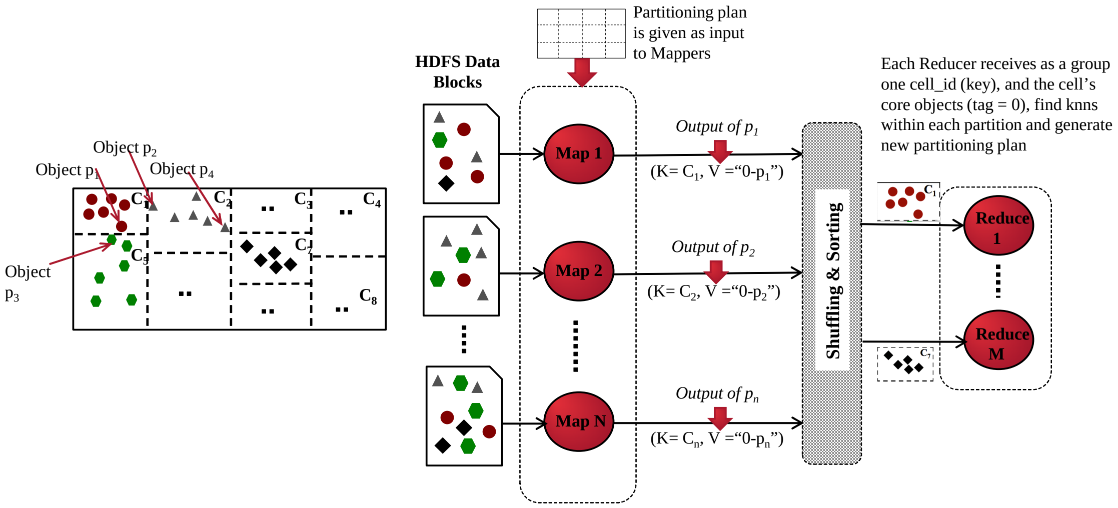


**Figure 20.3:** DDLOF: Inner Partition KNN Search.

$k$NN of $p_1$ found in $C_1$ (the local $k$NN) might not be the actual $k$NN of $p_1$. However such local $k$NN can be utilized to bound the supporting area of each cell.

**Lemma 20.1** *Given a cell $C_i = \langle(low_{1i}, high_{1i}),\ (low_{2i}, high_{2i}),\ ..., (low_{di}, high_{di})\rangle$ of data set D with domain space $Domain(D) = \langle(min_1, max_1),\ (min_2, max_2), ..., (min_d, max_d)$, suppose the k-distance of $p_j \in C_i$ is r. The vertical distance between $p_j$ and any boundary of cell $C_i$ is denoted as $dist(p_j, low_{xi})$ or $dist(p_j, high_{xi})$. Then the actual kNN of $p_j$ $kNN(p_j)$ is guaranteed to be discovered in cell $\hat{C}_i = \langle(max\{min_1, low_{1i} - Ext(low_{1i})\}, min\{max_1, high_{1i} + Ext(high_{1i})\}),\ (max\{min_2, low_{2i} - Ext(low_{2i})\}, min\{max_2, high_{2i} + Ext(high_{2i})\}), ..., (max\{min_d, low_{di} - Ext(low_{di})\}, min\{max_d, high_{di} + Ext(high_{di})\})\rangle$, where $Ext(low_{xi}) = max\{0, r - dist(p_j, low_{xi})\}$ and $Ext(high_{xi}) = max\{0, r - dist(p_j, high_{xi})\}$.*

**Proof:** We prove Lemma 20.1 by proving that for any given point $p_o \notin \hat{C}_i$ the distance between $p_o$ and $p_j$ $dist(p_o, p_j) \geq r$.

Here we denote the domain value of $p_o$ as $p_o(p_o^1, p_o^2, ..., p_o^d)$. If $p_o \notin \hat{C}_i$, then $p_o^x > min\{max_x, high_{xi} + Ext(high_{xi})\})$ or $p_o^x < (max\{min_x, low_{xi} - Ext(low_{xi})$.

Suppose $p_o^x > min\{max_x, high_{xi} + Ext(high_{xi})\})$. Since $p_o^x \leq max_x$ then $min\{max_x, high_{xi} + Ext(high_{xi})\} = high_{xi} + Ext(high_{xi})$. Otherwise $p_o^x$ cannot be larger than $min\{max_x, high_{xi}$

$+ Ext(high_{xi})\})$. Since $dist(p_o, p_j) \geq p_o^x - p_j^x > high_{xi} + Ext(high_{xi}) - p_j^x = (high_{xi} - p_j^x)$ $+Ext(high_{xi}) = dist(p_j, high_{xi}) + Ext(high_{xi}) = dist(p_j, high_{xi}) + max\{0, r - dist(p_j, high_{xi})\}$

.

If $dist(p_j, high_{xi}) \geq r$ then $max\{0, r - dist(p_j, high_{xi})\} = 0$. On the other hand if $dist(p_j, high_{xi}) < r$ then $max\{0, r - dist(p_j, high_{xi})\} = r - dist(p_j, high_{xi})$. In either case $dist(p_j, high_{xi}) + max\{0, r - dist(p_j, high_{xi})\} \geq r$. Since $dist(p_o, p_j) > dist(p_j, high_{xi})$ $+ max\{0, r - dist(p_j, high_{xi})\}$, we get $dist(p_o, p_j) > r$.

The condition of $p_o^x < (max\{min_x, low_{xi} - Ext(low_{xi})$ can be proven in the similar way. Due to space restriction, we omit the proof here.

Since for any given point $p_o \notin \hat{C}_i$ $dist(p_o, p_j) \geq r$, then any point out of $\hat{C}_i$ will not be $k$NN of $p_j$. Lemma 20.1 is proven. ■

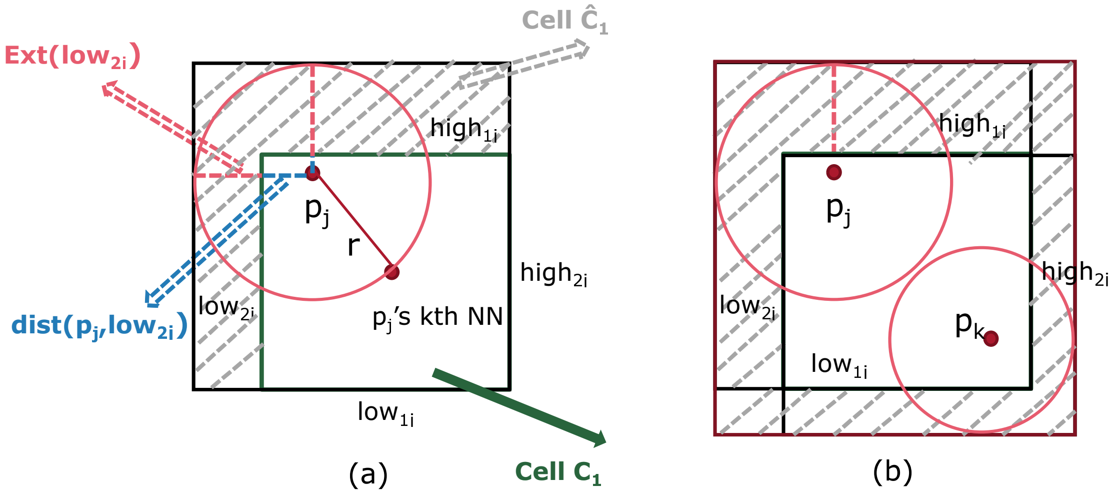


**Figure 20.4:** DDLOF: supporting area.

Fig. 20.4 (a) shows a two dimensional supporting area example. The $k$NNs of point $p_j$ are bounded in the red circle with radius as $r$. Clearly cell $\hat{C}_i$ covers the whole area of the red circle. Therefore the $k$NNs of $p_j$ is guaranteed to be discovered in $\hat{C}_i$. In $\hat{C}_i$ the area filled by the backslash pattern is the supporting area with respect to point $p_j$.

Once acquiring the supporting area for each point $p_j$ in cell $C_i$, it is intuitive to derive the supporting area of cell $C_i$ that covers all possible $k$NNs for all points in $C_i$. More specifically, it adopts the maximum $high_{xi}$ of each $\hat{C}_i$ as the final $high_{xi}$ and the minimum $low_{xi}$ of each $\hat{C}_i$ as the final $low_{xi}$ as shown in Fig. 20.4 (b).

During the reduce phase the "local" $k$NNs is attached to each point and written out to HDFS. For each cell $C_i$ the boundaries of its supporting area ($\hat{C}_i$) as well as the boundaries of $C_i$ itself together forms the partition plan of the next MapReduce job corresponding to outer partition $k$NN search.

## 20.3 Outer Partition $k$NN Search

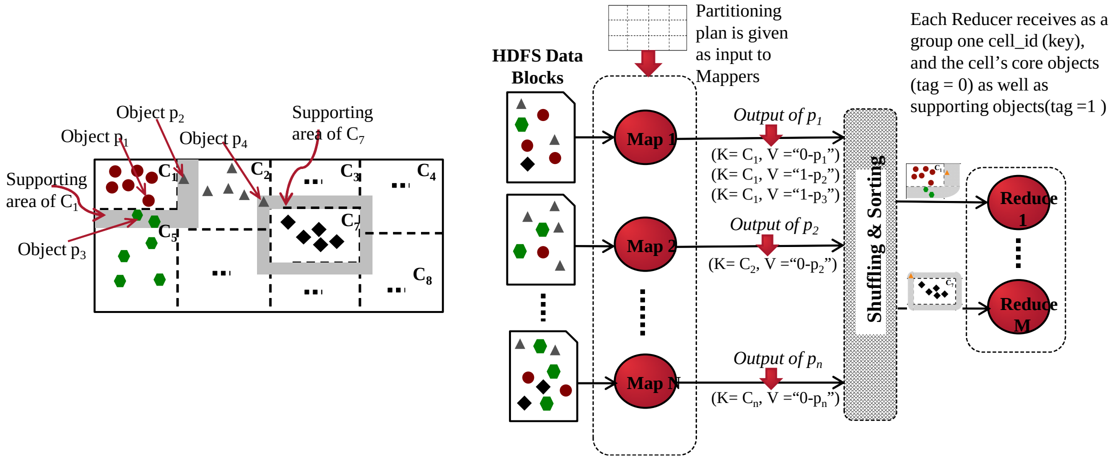


**Figure 20.5:** DDLOF: Outer Partition KNN Search.

Unlike the inner partition $k$NN search, now in the partition plan passed to the outer partition $k$NN search MapReduce job each cell is augmented with the supporting area. The points in the original cell $C_i$ are the points that we have to calculate their $k$NNs so cored core points, while the points in the supporting area, namely the points within $\hat{C}_i$ but out of $C_i$ are only utilized to support the $k$NN search of the core points so called support points. Therefore at the map phase each mapper retrieves one data block as well as the space partitioning strategy (Figure 20.5). Then for each data point $p_i$, the map function produces two types of output records, i.e., core- and supporting-related records.

The core-related record is one key-value pair record in the form of (K = $C_i$, V ="0-$p_i$"), where the key is the ID of the grid cell for which $p_i$ is a core point, i.e., $p_i \in C_i$. The prefixed flag "0" in the value component indicates that $p_i$ is a core point for $C_i$. For example, referring to Figure 20.5, the mapper *Map 1* generates output record (K = $C_1$, V ="0-$p_1$") for data point $p_1$.

Mappers also create zero or more supporting-related records for an input data point $p_i$ in the form of (K = $C_j$, V ="1-$p_i$"), where the key $p_i \in C_j$ is the ID of the grid cell for which $p_i$ is a support point. The prefixed flag "1" in the value component indicates that $p_i$ is a support point for $C_j$. For example, in Fig. 20.5, the mapper *Map 1* generates one output record (K = $C_1$, V ="1-$p_2$") for point $p_2$ since it is a support point for $C_1$.

After the internal shuffling and sorting phase based on the cell ID, each group received by a reducer will correspond to a specific grid cell, say $C_i$, and will consist of the union of the core and support points belonging to $C_i$ (See Figure 20.5. The reducer function categorizes the data points according to their attached flag encoded in the value. Lastly, it executes a $k$NN search algorithm on each core point.

In this step the local $k$NNs attached to each core point will be fully utilized. Given a core point $p_j$, first the local $k$NNs of $p_j$ will be parsed and stored in a list $tempKNN(p)$. The points in this list are sorted in the ascendent order by their distances to $p_j$. Then its $k$NN will only be searched within the support points. If one support point $p_s$ is closer to $p_j$ than at least one point in $tempKNN(p)$, $p_s$ is inserted into $tempKNN(p)$. The point at the tail will then be removed. This process proceeds until all support points are examined. Then the remaining points in $tempKNN(p)$ will be the actual $k$NN of $p_j$. Therefore the duplicate $k$NN search between inner partition $k$NN search and outlier partition $k$NN search is completely avoided.

# 21

# Performance Evaluation

## 21.1 Experimental Setup & Methodologies

**Experimental Infrastructure.** All experiments are conducted on a shared-nothing cluster with one master node and 40 slave nodes. Each node consists of 16 core AMD 3.0GHz processors, 32GB RAM, 250GB disk, and nodes are interconnected with 1Gbps Ethernet. Each server runs CentOS Linux (kernel version 2.6.32), Java 1.6, Hadoop 1.0.1. Each node is configured to run up to 8 map and 8 reduce tasks concurrently. The sort buffer size is set to 512MB. Speculative execution is disabled to boost performance. The replication factor is set to 3.

**Dataset.** We evaluated the performance of our proposed distributed local outlier detection algorithms (*PDLOF* and *DDLOF*) on the open data set: OpenStreetMap(OSM: $http : //wiki.openstreetmap.org/wiki/Main_Page$). OSM contains points with geographic position, stored as coordinates (pairs of a latitude and a longitude). The raw data was stored in a 500G XML file. Each row in the dataset represents a building. 3 attributes are utilized in the experiments, namely ID, longitude and latitude. In order to adjust parameters and evaluate the basic properties of our proposed algorithms, we extracted from

OpenStreetMap the Massachusetts data (3,000,000 records), about 750M in size.

**Methodology.** We used the following measures in our evaluation. First, we measured the total end to end time elapsed between launching the program and receiving the results. To provide more insight into the potential bottlenecks, we also broke down the total time into time spent at map phase and reduce phase. Since these phases are overlapping in MapReduce, we report the average time spent by each phase. Second, we measured the total number of records emitted and received by mappers in the MapReduce job that produces the supporting area for each partition. The number of records then is utilized to measure the duplicate rate of each partitioning method.

## 21.2 Evaluation Results

### 21.2.1 Evaluation of Duplication Rate

| | 75M | 750M |
|---|---|---|
| DDLOF | 0.24 | 0.97 |
| PDLOF | 21.3 | NULL |

**Table 21.1:** DLOF: Duplication Rate

First we measure the duplication rate of PDLOF and DDLOF. The duplication rate of PDLOF is measured by the ratio of the number of output records and the number of the input records at the map phase of the second mapreduce job when assigning support points to each partition. Similarly the duplication rate of DDLOF is measured at the map phase of map-reduce job corresponding to the outer$k$NN search. We measure the duplication rate using the 750M whole Massachusetts building data and a 75M subset of the Massachusetts building data. As shown in Table 21.1, the duplication rate of DDLOF is only about 1 percent of PDLOF at the 75M data case. Due to the high duplication rate, PDLOF cannot even handle the 750M dataset. The duplication of DDLOF increases a little when the size of the data set increases. However it is still smaller than 1. This indicates that DDLOF is very stable on duplication rate and scalable to big dataset.

This is expected, since as shown in Chapter 19 PDLOF utilizes the distances between the points and the pivots to estimate the largest possible k-distance for each partition. This worst case estimation, although theoretically correct, inevitably includes huge amount of support points that in fact far away from the core points and hence have no chance to be the $k$NN of any core point. On the other hand the two steps $k$NN strategy of DDLOF utilizes the "local" k-distance generated in the inner $k$NN search to bound the supporting area of the outer $k$NN search. This bound is much tighter than the worst case upper bound of the PDLOF. This is based on the convergence property of $k$NN. Namely the close points tend to be $k$NN with each other. Therefore the most of the points in one partition $P$ can locate their $k$NNs in $P$ itself.

### 21.2.2 Evaluation of CPU Time

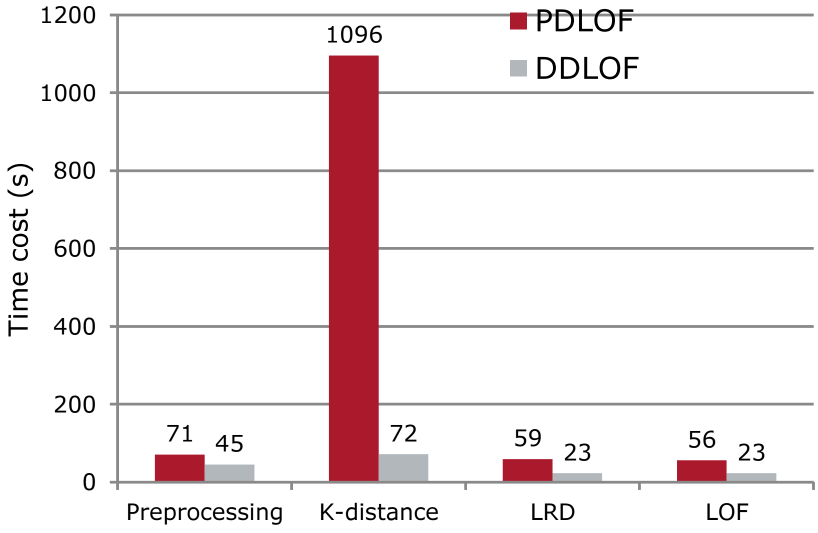


**Figure 21.1:** DLOF CPU Time Comparison: PDLOF VS DDLOF

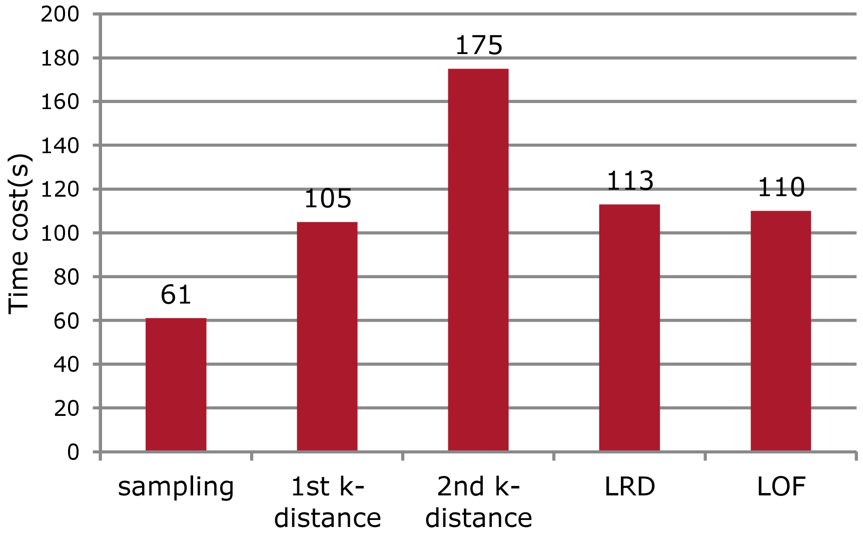


**Figure 21.2:** DLOF CPU time: DDLOF

In this experiment we evaluate the CPU time of PDLOF and DDLOF. As shown in Figure 21.1 DDLOF is 10 times faster than PDLOF even when handling the relatively small 75M data set. The performance gain of DDLOF is from the small duplication rate of its partitioning method as we have discussed in Section 21.2.1. The small duplication rate ensures each reducer only searches the $k$NNs for the core points in a small area and therefore significantly reduce the cost of $k$NN search, while $k$NN search is the bottleneck of the whole LOF computation process.

Since PDLOF cannot handle the 750M dataset, we only evaluate DDLOF using this larger dataset. As shown in Figure 21.2, the most time consuming step is the second MapReduce job corresponding the outer $k$NN search. In this job, mappers decide the supporting area for each partition. Reducers generate the $k$NNs for all core points that have not found their $k$NNs in the inner $k$NN search. Overall the two $k$NN search jobs consumes most of the time of the whole LOF computation. This also further confirms our analysis that $k$NN search is the bottleneck of LOF computation.

# 22

# Related Work

Breunig et al. proposed the notion of local outliers in opposition to that of distance-based outliers proposed by Knorr and Ng in [69]. Another distance-based outlier method also had been proposed in [70] to rank outliers using nearest neighbor relationships. To overcome what they saw as shortcomings of a binary definition of outliers, they sought to define a degree of outlierness by looking at the density of the point relative to its neighbors. Inspiration for LOF came from density-based clustering algorithms including DBSCAN [71] and BIRCH[72]. These methods can identify outlying points, however they classify them as noise.

Parallel versions of distance-based outlier detection can serve as a model for a distributed solution for LOF. A distributed detection algorithm is presented in [73] which uses the idea of a solving set of points sampled from the original dataset. A data point can be approximated as an outlier by comparison only to the elements in this set. This method only provides an approximate result, however it serves as inspiration for the use of sample points to understand areas of the dataset. In [74] another distributed approach is taken where data blocks are passed around a ring overlay network, and neighbors are computed in parallel. In-lying points are pruned along the way. This strategy can be applied to our work, since MapReduce does not allow communications among mappers or reducers.

The idea of the detection of top-n local outliers only was proposed in [75]. This method uses micro clustering of points to prune the points that are guaranteed not to be outliers. However this method is highly coupled with an expensive pre-processing strategy. Hence it is not scalable to big data.

The data partitioning method introduced in [76] can be utilized for KNN computation on map-reduce. In general it partitions the data into $n$ partitions. Then each distinct pair of partitions will be assigned to one reducer on which the $k$NNs are computed using the local data. One additional machine is utilized to derive the global $k$NN by collecting and summarizing the local $k$NNs. The duplicate rate of this approach is linear to the number of partitions. Therefore it is not scalable to big data set. The Voronoi diagram-based partitioning scheme is first introduced in [68]. However unlike our work, this work focuses on supporting KNN join utilizing MapReduce instead of local outlier detection. Furthermore, as shown in their experiments, this method cannot support data set larger than hundred megabyte.

Finally, data sampling has been well studied in databases and data mining. Uniform random sampling [77] is utilized in our work for data distribution estimation due to its simplicity. Density Biased Sampling [78] and Stratified Sampling [79] are well known in the literature. To relieve the problems caused by skewed data they either probabilistically under-sample dense regions and over-sample sparse regions or select sample size of each stratum proportional to the size of the stratum. Such methods may also be considered in the future.

# Part V

# Interactive Outlier Exploration

# 23

# ONION Model

We propose the online outlier exploration model or in short ONION for modeling and exploring the characteristics of distance-based outliers in a dataset D.

Fig. 23.1 sketches a high-level view of the ONION model. It is composed of the *multi-space* abstraction capturing the key characteristics and interrelationships of outliers and a rich set of *outlier exploration operations*.

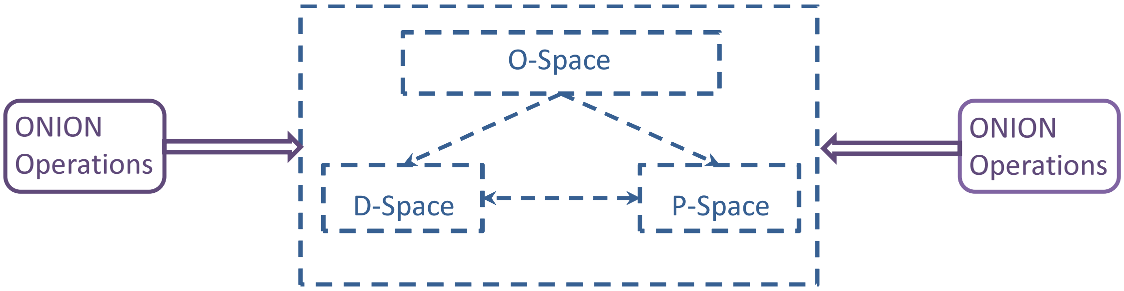


**Figure 23.1:** ONION Model

## 23.1 Multi-Space Abstraction

Our *multi-space* abstraction is composed of three interlinked spaces that we will now define below.

**ONION Space.** ONION space or in short *O-Space* is a three-dimensional space that models the distribution of the outliers with respect to their associated parameter settings.

**Definition 23.1** ***O-Space*** *denoted as* $\mathbb{O}^S$ $(Dim_k, Dim_r,\ Dim_d)$ *is a three-dimensional space with the possible settings of parameters r, k and data points p in dataset D being its three dimensions. The dimension* $Dim_k$ *ranges over the values that the parameter k can take in the universe of natural number* $U_k$ : $[k_{min}, k_{max}]$*, where* $k_{min}$ *and* $k_{max}$ *are the user-specified lower and upper bounds of the k values. Similarly the dimension* $Dim_r$ *corresponds to the domain of real numbers* $U_r$ : $[r_{min}, r_{max}]$ *with* $r_{min}$ *and* $r_{max}$ *the lower and upper bounds of the values of parameter r. Lastly the dimension* $Dim_d$ *represents all points* $p \in$ *D randomly organized into a linear order. Each point is assigned a position in [1,*$\mid$ *D* $\mid$*]. Each coordinate* $(k_i, r_i, p_i) \in \mathbb{O}^S$ *maps to a boolean value v* $\in$ *{0,1} indicating whether point* $p_i$ *is an outlier with respect to parameter values* $k_i$ *and* $r_i$*.*

In this O-Space any combination of k and r values on the dimensions $Dim_k$ and $Dim_r$ forms a parameter setting $ps_i$ denoted by $ps_i(k_i, r_i)$. Conceptually O-Space encodes the outlier status of all points in *D* with respect to all possible parameter settings.

Since dimension $Dim_d$ represents all data points in dataset D, $Dim_d$ corresponds to a discrete domain of positions. In other words the three-dimensional O-Space can be thought as a sequence of two dimensional slices as shown in Fig. 23.2. Each slice models the outlier status distribution with respect to all possible parameter settings for one particular point $p_i$ in dataset D.

Based on this O-Space, we further design two additional *higher level* abstractions called *parameter space* and *data space* respectively as shown below.

**Parameter Space.** Parameter space or in short *P-Space* is based on the observation that despite the infinite number of possible parameter settings, a large range of continuous parameter settings often generate the same set of outliers.

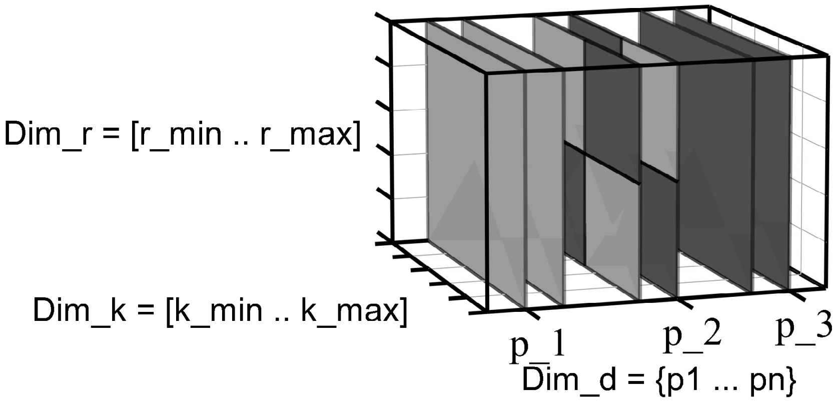


**Figure 23.2:** ONION Space

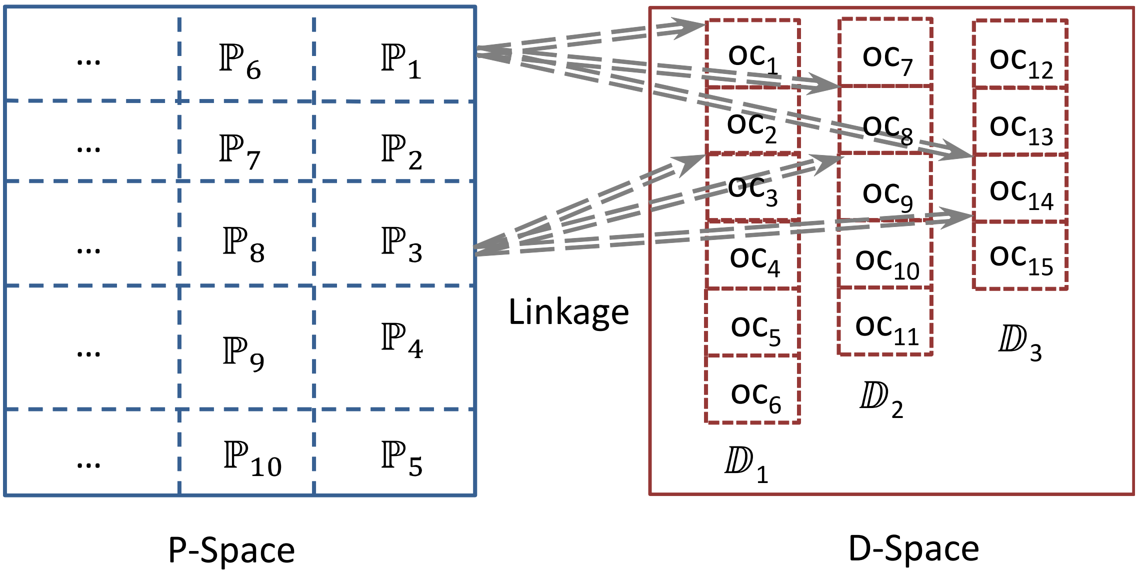


**Figure 23.3:** ONION: P-Space & D-Space

**Definition 23.2** ***P-Space*** $\mathbb{P}$ = $\mathbb{P}_1 \bigcup \mathbb{P}_2$ ... $\bigcup \mathbb{P}_m$*, such that:*

*(1) given any two parameter setting subsets* $\mathbb{P}_i$ *and* $\mathbb{P}_j$ *of* $\mathbb{P}$ *(1* $\leq$ *i, j* $\leq$ *m),* $\mathbb{P}_i \bigcap \mathbb{P}_j$ = $\emptyset$*;*

*(2) given any two parameter settings* $ps_j$ *and* $ps_l$ *in the same* $\mathbb{P}_i$ *(1* $\leq i \leq m$*),* $ps_j$ *and* $ps_l$ *generate the same set of outliers.*

In other words P-Space divides the two-dimensional space formed by the set of all possible values on the $Dim_k$, $Dim_r$ axes into a set of disjoint regions. Within each region no matter how the parameter settings are adjusted, the set of outliers generated from dataset D remains unchanged. Each such region is called a *stable region*.

P-Space, partitioning the infinite number of parameter settings into finite number of stable regions, explicitly reveals the influence of the parameter setting adjustment. This offers the analysts an opportunity to determine the appropriate parameter settings using a systematic methodology instead of a random trial and error process.

**Data Space.** Data space or in short *D-Space* leverages the key abnormality properties demonstrated in the points of dataset D, namely *outlier candidacy* and *domination relationship*.

**Outlier Candidacy.** Despite the infinite cardinality of P-Space $\mathbb{P}$, given a point $p_i$ in dataset D, its outlier status might be constant with respect to all parameter settings in $\mathbb{P}$. In other words, some points are guaranteed to be outliers in the entire P-Space so called *const outliers*, while some other points are guaranteed to be permanent inliers through the entire P-Space so called *const inliers*. In our O-Space these points would thus correspond to a slice that is all 1's for constant outlier or all 0's for constant inlier. For example, as shown in Fig. 23.2 $p_1$ is a const inlier, while $p_3$ is a const outlier.

Any point $p_i$ in D, that is neither a const outlier nor const inlier, is called an *outlier candidate oc* with respect to $\mathbb{P}$, meaning $p_i$ has opportunity to be classified as outlier for at least some of the parameter setting $ps_i$ in $\mathbb{P}$. In O-Space, an outlier candidate would have at least one cell in its corresponding slice that is 0 (white) and one that is 1 (black). In Fig. 23.2 $p_2$ is an outlier candidate.

In practice as confirmed by our experiments (Sec. 25.2), outlier candidates tend to be a strict minority among all points in D. This important *outlier candidacy observation* allows us to significantly reduce the number of data points to be maintained in D-Space. By this, ONION can concentrate the resource utilization on strictly serving these minority outlier candidates, rather than on computing and recording neighborhoods for the general and much larger data population when exploring outliers. Therefore ONION is able to efficiently explore outliers over even big datasets.

**Domination Relationship.** In dataset D some outlier candidates demonstrate a much *stronger abnormality* than others independent of any particular parameter setting in $\mathbb{P}$. In other words, some data points *dominate* others in abnormality as defined below.

**Definition 23.3** *Given a P-Space $\mathbb{P}$, outlier candidate $oc_i$ in dataset D dominates $oc_j$ if for **all** parameter settings in $\mathbb{P}$ $oc_j$ is guaranteed to be outlier when $oc_i$ is classified as outlier.*

By Def. 23.3 if outlier candidate $oc_i$ dominates $oc_j$, we say that the abnormality of $oc_i$ is stronger than $oc_j$.

Revealing the domination relationships among outlier candidates ONION offers the analysts an opportunity to better understand several characteristics of the detected outliers from sensitivity to stability. Without such understanding the detected outliers might only be some abstract points indistinguishable from each other for the analysts instead of some true unique abnormal phenomena.

Now we are ready to define our *data space* or in short *D-Space*.

**Definition 23.4** ***D-Space*** *$\mathbb{D}$ = $\mathbb{D}_1 \bigcup \mathbb{D}_2$ ... $\bigcup \mathbb{D}_m$, such that:*

*(1) $\forall$ $oc_i \in \mathbb{D}$, $oc_i$ is an outlier candidate;*

*(2) given any two outlier candidate subsets $\mathbb{D}_i$ and $\mathbb{D}_j$ of $\mathbb{D}$ (1 $\leq i, j \leq m$), $\mathbb{D}_i \bigcap \mathbb{D}_j$ = $\emptyset$.*

*(3) the outlier candidates in the same group $\mathbb{D}_i$ are sorted into a linear structure. Given two points $oc_j$ and $oc_l$ with j, l representing their positions in $\mathbb{D}_i$, $oc_j$ dominates $oc_l$ if $j < l$.*

In general, leveraging the outlier candidacy and domination relationship properties, D-Space partitions all outlier candidates into multiple disjoint groups. Within each group the domination relationship holds among all members of the group, so called *domination group*. Furthermore the outlier candidates falling in the same domination group are ordered based on the strongness of their abnormality. For example, in Fig. 23.3, D-Space $\mathbb{D}$ contains three subspaces $\mathbb{D}_1$, $\mathbb{D}_2$, and $\mathbb{D}_3$. In $\mathbb{D}_1$ candidates $oc_1$ to $oc_6$ are ordered by the domination relationship. That is, $oc_1$ dominates other members in $\mathbb{D}_1$. $oc_2$ is dominated by $oc_1$, but dominates $oc_3$ to $oc_6$.

**Linkage between P-Space and D-Space.** Furthermore as shown in Fig. 23.3, our ONION model explicitly establishes *linkages* between P-Space and D-Space, or in short *PD-linkage*.

**Definition 23.5** ***PD-Linkage.*** *Given a stable region $\mathbb{P}_i$ in P-Space $\mathbb{P}$ and a domination group $\mathbb{D}_j$ in D-Space $\mathbb{D}$, there exists a link $l(i, j)$ connecting $\mathbb{P}_i$ to an outlier candidate $oc_t \in \mathbb{D}_j$ such that:*

*(1) $\forall$ parameter setting $ps_i$ in $\mathbb{P}_i$, $oc_t$ is classified as outlier;*

*(2) $\forall$ parameter setting $ps_i$ in $\mathbb{P}_i$, $oc_{t-1}$ is classified as inlier.*

By the domination relationship definition in Def. 23.3, if $oc_t$ is an outlier with respect to $ps_i$, then any outlier candidates listed behind $oc_t$ in $\mathbb{D}_j$ ($oc_{t+1}$, $oc_{t+2}$, ...) are guaranteed to be outliers. Therefore the PD-Linkage explicitly connects the stable regions with their generated outliers. In Fig. 23.3, stable region $\mathbb{P}_1$ is linked to $oc_1$ of $\mathbb{D}_1$, $oc_8$ of $\mathbb{D}_2$, and

$oc_{14}$ of $\mathbb{D}_3$. Based on the links we immediately get the outlier set $\mathbb{O}_1$ generated by $\mathbb{P}_1$, that is $\{oc_1, ..., oc_6, oc_8, ..., oc_{11}, oc_{14}, oc_{15}\}$.

Overall the multi-space abstraction explicitly models the distribution of the outliers over all parameter settings, the relationships among the parameter settings, the stability and uniqueness of the outlier candidates. It establishes an innovative "outlier-centric panorama" into the outliers within dataset D.

## 23.2 ONION Operations

Based on the multi-space abstraction we further envision a rich classes of outlier exploration operations that allow users to explore and interpret outliers as well as pinpoint appropriate parameters.

**Definition 23.6** ***Comparative Outlier Analytics (CO).*** *Given an outlier set $\mathbb{O}_{in}$ as input, we report set of outliers $\mathbb{O}_D$ from dataset D, such that:*

*(1) $\forall$ point $p_i \in \mathbb{O}_{in}$, $p_i \in \mathbb{O}_D$; and*

*(3) $\forall$ point $p_i \in \mathbb{O}_D - \mathbb{O}_{in}$, if any $p_j \in \mathbb{O}_{in}$ is classified as outlier with respect to one $ps_l \in \mathbb{P}$, $p_i$ is guaranteed to be classified as outlier by $ps_l$.*

Leveraging the domination relationship in D-Space, CO operation returns all outliers dominated by the outliers specified in the input set $\mathbb{O}_{in}$. CO offers users a "parameter-free" approach to identify outliers based on their domain knowledge about the dataset. More specifically this CO operation helps analysts to identify outliers in a dataset based on sampling some typical outliers.

**Definition 23.7** ***Outlier-Centric Parameter Space Exploration (PSE).*** *Given an outlier set $\mathbb{O}_{in}$ and a $\delta$ ($-1 < \delta < 1$) as input, report all parameter settings $ps_j \in$ P-Space $\mathbb{P}$, such that:*

*(1) if $\delta \geq 0$, $ps_j$ identifies an outlier set $\mathbb{O}_j \subseteq \mathbb{O}_{in}$ where $\mid \mathbb{O}_j \mid$ = (1 - $\delta$) $\mid \mathbb{O}_{in} \mid$;*

*(2) if $\delta \leq 0$, $ps_j$ identifies an outlier set $\mathbb{O}_j \supseteq \mathbb{O}_{in}$ where $\mid \mathbb{O}_j \mid$ = (1 - $\delta$) $\mid \mathbb{O}_{in} \mid$.*

PSE leverages the stable region property of P-Space and allows analysts to conveniently evaluate the stability of a given outlier set $\mathbb{O}_{in}$. This is one important indicator of how significant the observed abnormal phenomena is. For example, if we set the $\delta$ as 0, PSE will return all the parameter settings that are guaranteed to generate the outliers identical to $\mathbb{O}_{in}$, namely a stable region of $\mathbb{P}$. The scope of the returned parameter settings (the size of the stable region) represents how stable the outlier set is across P-Space.

Furthermore PSE provides a tool for analysts to examine how changes in parameter settings may impact the resulting outliers. PSE achieves this, for example, by allowing the analysts to apply PSE to ask for the parameter settings that would return around (1 - $\delta$)% of $\mathbb{O}_{in}$ as the results and then compare them against the parameter settings that generate $\mathbb{O}_{in}$.

**Definition 23.8** ***Outlier Detection (OD).*** *Given a dataset D and a parameter setting $ps_i$ as input, outlier detection returns:*

*(1) all outliers $p_j \in$ D with respect to $ps_i$ if $ps_i \in$ P-Space $\mathbb{P}$; or*

*(2) all points $p_j \in$ D that are classified as outliers with respect to any parameter setting $\in \mathbb{P}$ if $ps_i$ = NULL.*

As shown in Def. 23.8 unlike the traditional distance-based outlier definition, OD leverages the *outlier candidacy* observation of D-Space to allow the input parameter set as NULL. This will return all points that are guaranteed to be outliers with respect to the entire P-Space, that is, the *constant outliers*.

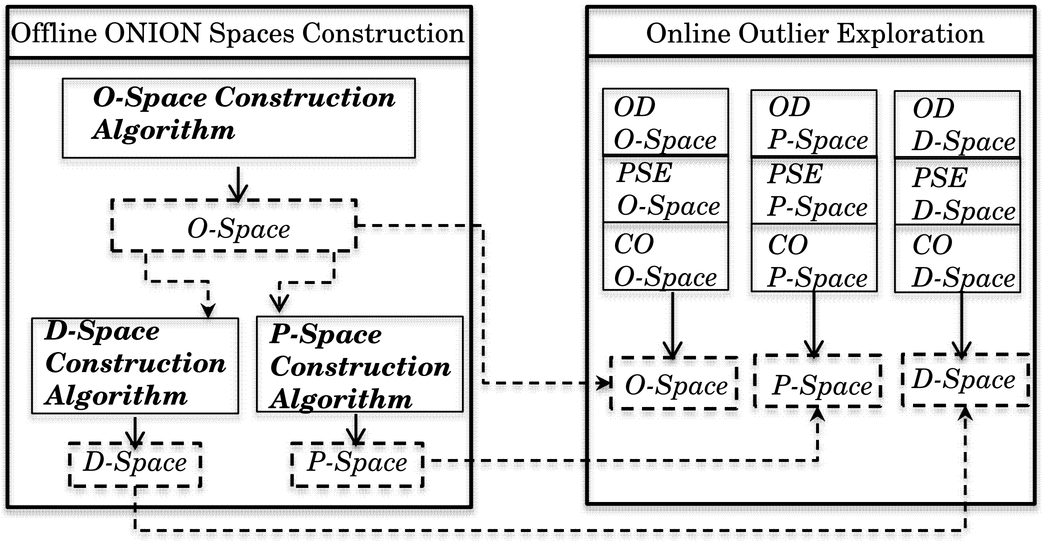
**Use Case.** Those operations in combination provide a powerful tool for analysts to quickly approach the parameter settings appropriate for her application. For example, when facing a new dataset recording stock market transactions, an analyst may not have any experience to be able to appropriately determine values for parameters k and r. However, given her domain expertise, she may be aware that certain records are abnormal (outliers). Then a CO operation can be applied to help her identify all outliers satisfying her intuition. If the analyst finds the volume of the outliers $\mathbb{O}$ returned by the CO request too overwhelming, then she could apply PSE to ask for the parameter settings that would return, for example, around 60% of $\mathbb{O}$ as the result by setting $\delta$ as 0.4. Eventually the OD operation is applied to catch the true outliers to her interest.

Note that the above running example we gave here is just one of many combinative usages of our proposed operations. Those operations can be used individually or in other combinations to serve the ever changing outlier analysis demands.

# 24

# ONION Framework

To achieve the ONION model we designed the novel ONION framework. As shown in Fig. 24.1 ONION framework consists of two phases (a) offline multi-space abstraction construction and (b) online exploration operation processing using the corresponding ONION spaces.



Offline ONION Spaces Construction
O-Space Construction Algorithm
O-Space
D-Space Construction Algorithm
P-Space Construction Algorithm
D-Space
P-Space
Online Outlier Exploration
OD O-Space
PSE O-Space
CO O-Space
OD P-Space
PSE P-Space
CO P-Space
OD D-Space
PSE D-Space
CO D-Space
O-Space
P-Space
D-Space

**Figure 24.1:** ONION Framework

## 24.1 O-Space

### 24.1.1 Offline O-Space Construction

As shown in Fig. 23.2, the three-dimensional O-Space can be decomposed into a set of two dimensional slices. Each slice corresponds to the outlier status of one point $p_i$ in dataset D with respect to all parameter settings $ps_i$ in the two-dimensional space $\mathcal{P}$ formed by the dimensions $Dim_k$ and $Dim_r$. Therefore O-Space can be established by modeling the outlier status distribution in $\mathcal{P}$ for each point $p_i$ in dataset *D*, called O-Space($p_i$).

The key insight here is that given a point $p_i$ in dataset *D*, it is not necessary to establish O-Space($p_i$) by evaluating $p_i$ for each possible $ps_i$ in $\mathcal{P}$. In fact the outlier status of $p_i$ with respect to any $ps_i$ in $\mathcal{P}$ can be correctly determined by collecting only a small amount of meta information.

We first introduce our *k-distance* observation. Generally speaking given a set of outlier detection requests with the same parameter value k for $Dim_k$, but random values for $Dim_r$, the outlier status of any point $p_i$ for any of those requests can be determined by checking the distance of $p_i$ towards *one single point* in D. This observation is formally defined in Lemma 24.1.

**Lemma 24.1** *Given a set of parameter settings $\mathcal{P}_k \subset \mathcal{P}$, where $\forall$ two parameter settings $ps_x(k_x, r_x)$, $ps_y(k_y, r_y) \in \mathcal{P}_k$, $k_x = k_y = k$, then the outlier status of $p_i$ with respect to any $ps_x$ in $\mathcal{P}_k$ is determined by the distance between $p_i$ and its kth-nearest neighbor $p_j$ denoted as $D^k_{p_i}$.*

**Proof.** Given any parameter setting $ps_x(k, r_x) \in \mathcal{P}_k$, if $D^k_{p_i} > r_x$, then by the definition of the kth-nearest neighbor, there are at most k-1 other points $p_j \in$ D whose distance towards $p_i$ is not larger than $r_x$. In other words, $p_i$ has at most k-1 neighbors. By Def. 2.1, $p_i$ is an outlier. On the other hand, if $D^k_{p_i} \leq r_x$, then there are at least k points $p_j$ with

$d(p_i, p_j) \leq r_x$, namely $p_j$ are all neighbors of $p_i$. $p_i$ is then classified as an inlier by Def. 2.1. Therefore $\forall\ ps_x(k, r_x) \in \mathcal{P}_k$, the outlier status of $p_i$ can be correctly determined by comparing $r_x$ against $D^k_{p_i}$. Lemma 24.1 is proven. ■

Now we are ready to introduce the *space delimiter* insight as the foundation for building O-Space.

**Lemma 24.2** *Given a dataset D and parameter setting space* $\mathcal{P}$*,* $\forall\ p_i \in$ *D the distance set* $\mathbb{DS}(p_i)$ = $\{D^{k_x}_{p_i} | k_{min} \leq k_x \leq k_{max}\}$ *is sufficient to determine the outlier status of* $p_i$ *with respect to any parameter setting ps* $\in \mathcal{P}$*.*

**Proof.** $\mathcal{P}$ = $\mathcal{P}_{k_{min}} \cup \mathcal{P}_{k_{min+1}} \cup \mathcal{P}_{k_{min+2}} \ldots \cup \mathcal{P}_{k_j} \ldots \cup \mathcal{P}_{k_{max-1}} \cup \mathcal{P}_{k_{max}}$, where $\mathcal{P}_{k_j}$ is composed by any $ps_x(k_x, r_x) \in \mathcal{P}$ with $k_x = k_j$ ($k_{min} \leq k_j \leq k_{max}$). Therefore given any $ps \in \mathbb{P}\ ps$ is guaranteed to be covered by some $\mathcal{P}_{k_j}$. By Lemma 24.1, $\forall ps \in \mathcal{P}_{k_j}$ the status of $p_i$ can be determined by examining $D^{k_j}_{p_i}$. Since $D^{k_j}_{p_i} \in \mathbb{DS}(p_i)$, therefore $\mathbb{DS}(p_i)$ is sufficient to determine the status of $p_i$ with respect to any $ps \in \mathcal{P}$. Lemma 24.2 is proven.■

As shown in Fig. 24.2 this distance set $\mathbb{DS}(p_i)$ delimits $\mathcal{P}$ into two segments. The parameter settings in different segments will classify $p_i$ to different outlier status. Therefore $\mathbb{DS}(p_i)$ is called *space delimiter* of $p_i$. The set of space delimiters $\{\ \mathbb{DS}(p_i)\ | p_i \in D\}$ effectively represents the three dimensional *O-Space*.

Furthermore the space delimiter structure also provides us an approach to quickly discover constant inliers and constant outliers.

**Lemma 24.3** *A point* $p_i$ *is a const inlier if* $D^{k_{max}}_{p_i} \leq r_{min}$*.*

**Proof.** If the distance to $p_i$'s $k_{max}$th nearest neighbor is $\leq r_{min}$, then $p_i$ has at least $k_{max}$ neighbors or more even under most restricted neighbor criteria, namely $Dim_r$ = $r_{min}$. Then $p_i$ is an inlier for *ps*($k_{max}$,$r_{min}$) that is the most restricted parameter setting in

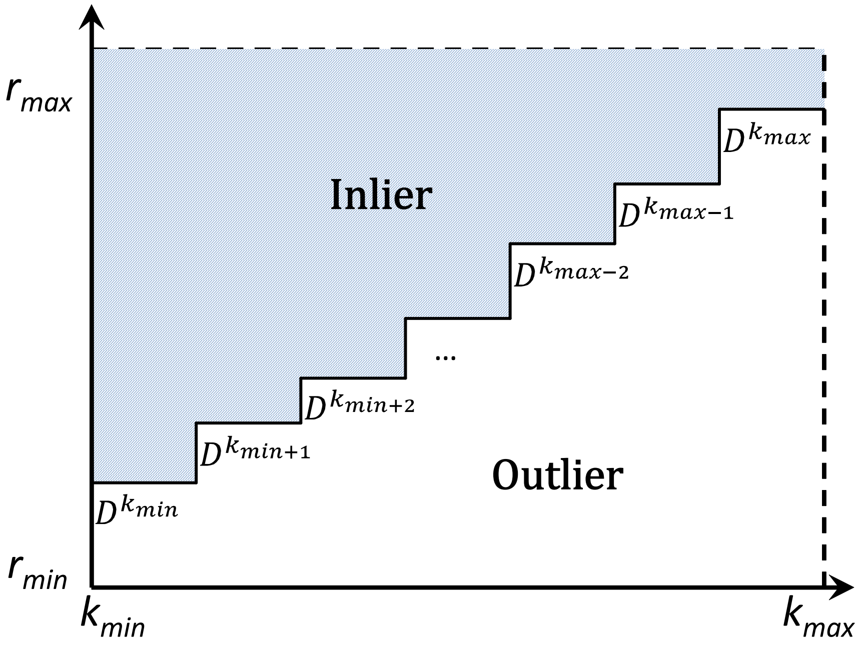


**Figure 24.2:** ONION: Space Delimiter

$\mathcal{P}$ in terms of recognizing outlier. If $p_i$ is not an outlier in the most restricted setting, then of course it cannot be outlier in any part of $\mathcal{P}$. Therefore $p_i$ is a const inlier. ■

**Lemma 24.4** *A point $p_i$ is a const outlier if $D_{p_i}^{k_{min}} > r_{max}$.*

Lemma 24.4 can be proven in the similar way of proving Lemma 24.3. Due to space limitation, the proof is omitted.

Naturally any point that is not a const outlier nor a const inlier, is an *outlier candidate* $oc$. Among all points only for $oc$ it is necessary to maintain its *space delimiter*.

Therefore constructing O-Space has two tasks, namely: (1) discovering all ocs and (2) collecting the space delimiter $\mathbb{DS}(oc)$ for each $oc$. Intuitively this can be done by first collecting $\mathbb{DS}(p_i)$ for each point $p_i$, then locating the constant inliers and outliers by applying Lemmas 24.3 and 24.4. Collecting $\mathbb{DS}(p_i)$ is straightforward. We can acquire the k nearest neighbors ($k$NN) of $p_i$ by applying any $k$NN algorithm with k set as $k_{max}$. Since we only care for the range $k_{min}$ to $k_{max}$, we then discard the $k_{min-1}$ nearest neighbors.

However to discover const inliers it is not necessary to acquire the actual $k_{max}$ nearest neighbors. Once $p_i$ acquires $k_{max}$ neighbors whose distance to $p_i$ is not larger than $r_{min}$, $p_i$ is guaranteed to be const inlier. Then the $k$NN search can be terminated immediately. Since const inliers are typically the majority of the dataset, this optimization significantly speeds up the preprocessing process.

**Space Complexity.** The O-Space data structure is composed of a set of arrays. Each of the arrays contains $(k_{max} - k_{min} + 1)$ float values (distance) corresponding to the space delimiter of one outlier candidate. Therefore the space complexity is linear in the number of outlier candidates $\mid \mathbb{OC} \mid$. More precisely it is $O(\mid \mathbb{OC} \mid (k_{max} - k_{min} + 1))$.

As confirmed by our experiments (Fig. 25.1(c), Sec. 25.3.1), only a small fraction of points is classified as outlier candidates. Most of the points are recognized as const inliers. Therefore *L* is much smaller than the actual cardinality *n* of the input dataset. Hence the O-Space structure is found to be rather compact and in fact small enough to be accommodated in the main memory of a standard PC even when handling a fairly large dataset in order of 10GB.

**Time Complexity.** The time complexity of constructing O-Space is $O(n^2)$ because of the potential $KNN$ search on each point. Here *n* represents the cardinality of the input dataset D. Furthermore it is worth to emphasize that in fact the cost of building O-Space is similar to the cost of answering one single outlier detection request as confirmed in our experiments (Figures 25.1(a), 25.1(b), Sec. 25.2.1). In ONION, the expensive exact $KNN$ search is only conducted on outlier candidates. This significantly speeds up the construction of O-Space.

### 24.1.2 Online Outlier Exploration

O-Space is sufficient to support all three classes of outlier exploration operations. In particular by intelligently maintaining the space delimiter information of each outlier candidate, we are able to drive down the time complexity of supporting online outlier detection (OD) operation from quadratic to linear.

**Outlier Detection (OD).** For each outlier candidate $oc$ we maintain its space delimiter $\mathbb{DS}$ in an array structure by the order of its $k_{min}$th neighbor at the head and the $k_{max}$th neighbor at the end. Then for any parameter *ps* $\in \mathcal{P}$, the outlier status of $oc$ can be immediately determined by applying the following *examination rule*.

**Definition 24.1** *Given an outlier candidate oc and its* $\mathbb{DS}$ *structure,* $\forall$ *parameter setting* $ps(k_x, r_x)$ *in* $\mathcal{P}$*, oc is an outlier if* $\mathbb{DS}[k_x - k_{min}] > r_x$*. Otherwise* $p_i$ *is an inlier.*

Therefore to answer OD we only need to perform one scan on the outlier candidate set $\mathbb{OC}$ and sequentially apply the examination rule in Def. 24.1 on each $oc$. Hence the time complexity is linear to the cardinality of $\mathbb{OC}$.

**Outlier-Centric Parameter Space Exploration (PSE).** Given an outlier set $\mathbb{O}_{in}$, the parameter settings that recognize $\mathbb{O}_{in}$ as outliers is the intersection of a set of parameter space segments $\mathbb{S}_i$ with respect to each point $p_i$ in $\mathbb{O}_{in}$. All parameter settings in $\mathbb{S}_i$ with respect to $p_i$ will classify $p_i$ as outlier. By Lemma 24.2 this can be done by checking and comparing the space delimiters $\mathbb{DS}$ of all points in $\mathbb{O}_{in}$. The time complexity is O(| $\mathbb{O}_{in}$ |($k_{max}$ - $k_{min}$)).

**Comparative Outlier Analytics (CO).** Similar to PSE, given an outlier set $\mathbb{O}_{in}$, CO can be answered by checking the parameter space segment $\mathbb{S}_i$ with respect to each outlier candidate $oc_i$ in $\mathbb{OC}$ - $\mathbb{O}_{in}$. Point $oc_i$ is dominated by all points $o_j$ in $\mathbb{O}_{in}$ if $\mathbb{S}_i \supseteq \mathbb{S}_j$ for all $oc_j$. The time complexity is O(| $\mathbb{OC}$ |($k_{max}$ - $k_{min}$)).

## 24.2 P-Space

### 24.2.1 Offline P-Space Construction

To construct the P-Space we first introduce the concept of $k$**-domination** between two outlier candidates.

**Definition 24.2** *Given two outlier candidates $oc_i$ and $oc_j$ and a k value of $Dim_k \in [k_{min}, k_{max}]$, if $D^k_{oc_i} \leqslant D^k_{oc_j}$, then $oc_i$ k-**dominates** $oc_j$ .*

The following *monotonic property* holds if the $k$-domination relationship holds between $oc_i$ and $oc_j$.

**Lemma 24.5** *Given two outlier candidates $oc_i$ and $oc_j$ with $oc_i$ k-dominating $oc_j$, then for any parameter setting $ps(k, r_x) \in \mathcal{P}$ ($r_{min} \leq r_x \leq r_{max}$), if $oc_i$ is classified as outlier by ps, then $oc_j$ is guaranteed to be outlier with respect to ps.*

**Proof.** If $oc_i$ is an outlier with respect to $ps(k, r_x)$, $D^k_{oc_i} > r_x$. Since $D^k_{oc_j} \geq D^k_{oc_i}$ by the k-domination definition in Def. 24.2, $D^k_{oc_j} > r_x$. Therefore $oc_j$ is an outlier with respect to *ps*. ■

In other words, if one parameter setting $ps(k, r_x)$ classifies $p_i$ as an outlier, then any point $k$-dominated by $p_i$ is guaranteed to also be an outlier. On the other hand, if one parameter setting classifies $p_i$ as an inlier, then any point that $k$-dominates $p_i$ is also guaranteed to be an inlier as well.

It is straightforward to prove that the k-domination relationship also satisfies the *transitive property*.

**Lemma 24.6** *Given three candidates $oc_h$, $oc_i$, and $oc_j$, if $oc_h$ k-dominates $oc_i$ and $oc_i$ k-dominates $oc_j$, then $oc_h$ k-dominates $oc_j$.*

The above properties of the k-domination relationship now enable us to divide the infinite parameter setting space $\mathcal{P}$ into a finite number of *stable parameter regions*.

**Lemma 24.7** *Given the outlier candidate set $\mathbb{OC} \subset$ dataset D and $\mathcal{P}_{k_i} \subset \mathcal{P}$, where $\mid \mathbb{OC} \mid$ = n and $\mathcal{P}_{k_i}$ is composed by any parameter setting ps in $\mathcal{P}$ sharing the same $Dim_k$ value $k_i$, then $\mathcal{P}_{k_i}$ can be divided into n+1 stable regions $\mathbb{P}^j_{k_i}$, where $Dim_r$ of $\mathbb{P}^1_{k_i} \in [r_{min}, D^{k_i}_{oc_1})$, $Dim_r$ of $\mathbb{P}^2_{k_i} \in [D^{k_i}_{oc_1}, D^{k_i}_{oc_2})$, ..., $Dim_r$ of $\mathbb{P}^{j+1}_{k_i} \in [D^{k_i}_{oc_j}, D^{k_i}_{oc_{j+1}})$ ,...., $Dim_r$ of $\mathbb{P}^{n+1}_{k_i} \in [D^{k_i}_{oc_n}, r_{max}]$ ($D^{k_i}_{oc_1} < D^{k_i}_{oc_2}$, ..., $< D^{k_i}_{oc_j} < D^{k_i}_{oc_{j+1}}$, ..., $< D^{k_i}_{oc_n}$). The identical set of outliers are guaranteed to be generated for all ps $\in \mathbb{P}^j_{k_i}$.*

**Proof.** $\forall\ ps(k_i, r_x) \in \mathbb{P}^j_{k_i}$, since $D^{k_i}_{oc_{j-1}} \leq r_x < D^{k_i}_{oc_j}$, $ps(k_j, r_x)$ will classify $oc_{j-1}$ as inlier, while $oc_j$ would be classified as outlier. Since $D^{k_i}_{oc_{j-2}} < D^{k_i}_{oc_{j-1}}$ and $D^{k_i}_{oc_j} < D^{k_i}_{oc_{j+1}}$, we get $oc_{j-2}$ k-dominates $oc_{j-1}$ and $oc_j$ dominates $oc_{j+1}$. Based on the monotonic property of k-domination, $oc_{j-2}$ will also be classified as an inlier, while $oc_{j+1}$ remains as outlier. Furthermore by the transitive property of k-domination, $\forall\ ps(k_i, r_x) \in \mathcal{P}^{j+1}_{k_i}$, $oc_1, oc_2, ..., oc_{j-2}$, $oc_{j-1}$ are guaranteed to be inliers, while $oc_j, oc_{j+1}, ..., oc_n$ are guaranteed to be outliers. Therefore the identical set of outliers will be generated for any *ps* $\in \mathbb{P}^{j+1}_{k_i}$. Lemma 24.7 has thus been proven.■

Leveraging Lemma 24.7 we design an light-weight algorithm (Alg. 9) to build P-Space $\mathbb{P}$. We first define the *parameter node* structure.

**Definition 24.3** *A parameter node, or in short pn, is a data structure composed of the following three elements:*

*-pn.obj: an outlier candidate oc in $\mathbb{OC}$;*

*-pn.k: k parameter value (k $\in [k_{min}, k_{max}]$);*

*-pn.r. r parameter value, pn.r = $D^{pn.k}_{oc}$;*

By Def. 24.3, each outlier candidate *oc* in $\mathbb{OC}$ will be mapped to *m* nodes, where m = $k_{max} - k_{min} + 1$. Each of the nodes corresponds to one element in the space delimiter

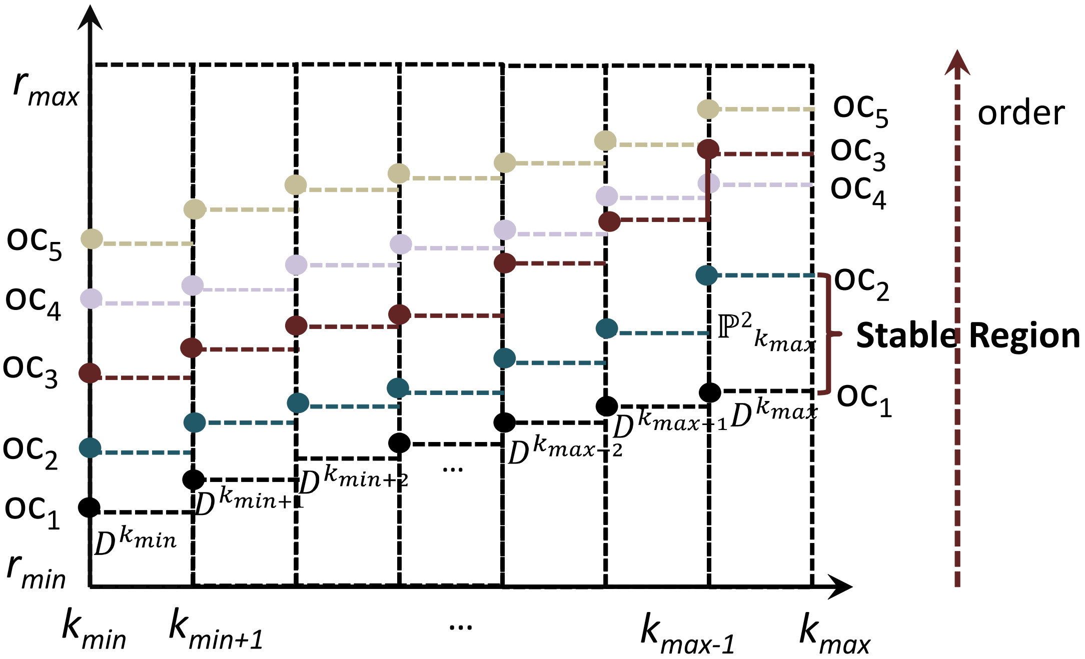


**Figure 24.3:** ONION: Stable Region

$\mathbb{DS}$ of $oc$, that is $D^{k_i}_{oc}$ ($k_{min} \leq k_i \leq k_{max}$). Then we organize the parameter nodes based on *pn.k* into m array lists. Each array list called $kthList$ contains the nodes with the same *pn.k* value. Therefore each outlier candidate *oc* is represented by exactly one node *pn* in each $kthList$.

The key idea behind Alg. 9 is to sort the parameter nodes in each $kthList$ in ascending order based on their *pn.r* values. By this each subspace $\mathcal{P}_{k_i}$ ($k_{min} \leq k_i \leq k_{max}$) of $\mathcal{P}$ is divided into multiple stable regions $\mathbb{P}^j_{k_i}$ by $D^{k_i}_{oc}$ in $\mathbb{DS}(oc)$ with respect to each *oc* in $\mathbb{OC}$. For example as shown in Fig. 24.3, $\mathbb{P}^2_{k_{max}}$ is a stable region of $\mathcal{P}_{k_{max}}$ bounded by $D^{k_{max}}_{oc_1}$ of $oc_1$ and $D^{k_{max}}_{oc_2}$ of $oc_2$ ($[D^{k_{max}}_{oc_1}, D^{k_{max}}_{oc_2})$). All parameter settings in $\mathbb{P}^2_{k_{max}}$ classify $oc_2$, ..., $oc_5$ as outliers.

P-Space then is represented by a hash map with pn.k as the *key* and the corresponding $kthList$ as *value*.

**Algorithm 9** constructPSpace

```
$P\text{-}Space$ = $\emptyset$;
for each k from $k_{min}$ to $k_{max}$ do
    $kthList$ = $\emptyset$;
    for each $oc \in ocs$ do
        $kthList$.add(new pn(oc, oc.$\mathbb{DS}$, k));
    $kthList.sort()$;
    $P\text{-}Space$.put(k, kthList);
return $P\text{-}Space$;
```

### 24.2.2 Online Outlier Exploration

**Outlier Detection (OD).** Given a parameter setting $ps_i(k_i, r_i)$ to detect the outliers we only need to locate a particular parameter node $pn_{min}$ in P-Space where $pn_{min}.k = k_i$ and $pn_{min}.r = min(\{pn.r \mid pn.r > r_i\})$. Then the outliers for $ps_i$ will be the outlier candidates corresponding to the parameter nodes in the $k_i thList$ of P-Space and listed behind $pn_{min}$.

**Complexity Analysis.** Since each array list is sorted by the pn.r value, $pn_{min}$ can be located in $O(log(\mid \mathbb{OC} \mid))$ time using a binary search style algorithm [80].

**Outlier-Centric Parameter Space Exploration (PSE).** Utilizing P-Space to support PSE operation is straightforward. We can traverse through each $kthList$ of P-Space to locate the stable regions that return the outlier set $\mathbb{O}_{in}$ specified in the input. Given one particular array list $k_i thList$, we first locate the parameter node $pn_{1st}$ of $oc_j$ corresponding to the first outlier in $\mathbb{O}_{in}$. Then we compare the outliers in $\mathbb{O}_{in}$ with the objects listed behind $pn_{1st}$ in $k_i thList$ one by one. If all objects match, one stable region $\mathbb{P}^j_{k_i}$: [$D^{k_i}_{oc_{j-1}}$, $D^{k_i}_{oc_j}$) will be returned.

**Complexity Analysis.** The cost of supporting PSE relies on the number of $kthList$ and the outliers in $O_{in}$. Therefore the time complexity is $O(m \mid \mathbb{O}_{in} \mid)$, where m = $k_{max} - k_{min} + 1$.

**Comparative Outlier Analytics (CO).** Given an outlier set $\mathbb{O}_{in}$, CO operation can be answered by checking each outlier candidate $oc_i$ in $\mathbb{OC}$ - $\mathbb{OC}_{in}$. $oc_i$ is dominated by all points $o_j$ in $\mathbb{O}_{in}$ if $oc_i$ is listed behind all $o_j$ in every $kthList$.

**Complexity Analysis.** The time complexity is $O(m \mid \mathbb{OC} \mid)$, where m = $k_{max} - k_{min} + 1$.

## 24.3 D-Space

### 24.3.1 Offline D-Space Construction

By Def. 23.4, to construct D-Space $\mathbb{D}$, we have to divide all outlier candidates *oc* into multiple domination groups $\mathbb{D}_i$. The domination relationship holds among all *oc*s falling in the same group $\mathbb{D}_i$.

Next we introduce the *domination rule* in Lemma 24.8 to evaluate whether the domination relationship holds between two outlier candidates based on their space delimiters in O-Space.

**Lemma 24.8** *Given two outlier candidates $oc_i$ and $oc_j$, $oc_i$ dominates $oc_j$ if $\forall$ $k_l \in [k_{min}, k_{max}]$, $D^{k_l}_{oc_i} \leq D^{k_l}_{oc_j}$*

**Proof.** By Def. 24.2, given one $k \in [k_{min}, k_{max}]$, if $D^k_{oc_i} \leq D^k_{oc_j}$, then $oc_i$ $k$-dominates. By Lemma 24.5 given any parameter setting *ps* $\in$ parameter subspace $\mathcal{P}_k \subset \mathcal{P}$, $oc_j$ is guaranteed to be outlier if $oc_i$ is classified as outlier by *ps*. Since $D^{k_l}_{oc_i} \leq D^{k_l}_{oc_j}$ holds for any $k_l \in [k_{min}, k_{max}]$, then $oc_i$ $k_l$-dominates $oc_j$ for any $k_l$. Therefore if $oc_i$ is classified as an outlier by any parameter setting *ps* in $\mathcal{P}$, then $oc_j$ is guaranteed to be an outlier. By the definition of domination relation in Def. 23.3, Lemma 24.8 is proven. ■

As shown in Fig. 24.3, $oc_1$ dominates $oc_2$, because $D^k_{oc_1} < D^k_{oc_2}$ for any k $\in$ [$k_{min}$, $k_{max}$].

It is straightforward to prove that domination relationship satisfies the transitivity property.

**Lemma 24.9** *Given three outlier candidates $oc_h$, $oc_i$, and $oc_j$, if $oc_h$ dominates $oc_i$ and $oc_i$ dominates $oc_j$, then $oc_h$ dominates $oc_j$.*

Next we propose a graph-based solution that successfully constructs D-Space. First we construct an undirected graph based on the domination relationships among all outlier candidates.

**Definition 24.4** ***Domination Graph**. The domination graph of the outlier candidate set $\mathbb{OC}$ is a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, such that (1) a node $v_i$ exists in $\mathcal{V}$ to represent a point $oc_i$ in $\mathbb{OC}$, and (2) an edge $e_{ij} = (v_i, v_j)$ exists in $\mathcal{E}$ if domination relationship does not hold between $oc_i$ and $oc_j \in \mathbb{OC}$ corresponding to nodes $v_i$ and $v_j$ in $\mathcal{V}$.*

This domination graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ tends to be a sparse graph, because the domination relationship tends to hold among most points in $\mathbb{OC}$. This is the case because the distance of $oc_i$ towards its $k$NN usually does not dramatically change. If $D^{k_i}_{oc_i}$ is smaller than $D^{k_i}_{oc_j}$, then $D^{k_{i+1}}_{oc_i}$ also tends to be smaller than $D^{k_{i+1}}_{oc_j}$.

Given a domination graph $\mathcal{G}$, a completely disjointed graph with zero edge can always be derived by removing some nodes and the corresponding edges. This indicates by removing a small number of points corresponding to these nodes, we can get a subset of $\mathbb{OC}$ such that the domination relationship holds among all points in it. If we could determine the *minimal number of nodes* whose removal will completely isolate the remaining nodes, then we could build the largest domination group out of $\mathbb{OC}$. We now note that the problem of finding the minimal number of nodes to remove so that no edge remains in $\mathcal{G}$ can be mapped to the minimum vertex cover problem – a classical *NP-complete* problem.

Clearly any minimum vertex cover algorithm can be applied here. Then D-Space can be built by recursively applying the minimum vertex cover algorithm on the removed nodes as shown in Alg. 10. The domination group built in each iteration is guaranteed to be the largest at that round. Therefore this process concurrently also minimizes the number of the domination groups. Since the domination graph tends to be a sparse graph, the number of the domination groups generated is small. As confirmed in our experiments, usually two or three trees are sufficient to cover all outlier candidates.

**Algorithm 10** $construct\_DForest$

```
Input: $\mathbb{OC}$ // outlier candidates
Output: $domination\_forest$ // constructed domination forest;
if ($\mathbb{OC}$ == $\emptyset$) then
    return $\emptyset$;
else
    $domination\_tree$ = $\emptyset$;
    removed = minVertexCover($\mathbb{OC}$);
    $\mathbb{OC}$ = $\mathbb{OC}$ - removed;
    $domination\_tree$ = buildDtree($\mathbb{OC}$);
    return $domination\_forest$ + $domination\_tree$ + $Construct\_DForest$(removed);
```

Given a domination group $\mathbb{D}_i$ a *domination tree* $tree_i$ can be constructed by sorting the outlier candidates in the ascending order based on the distance to their kth nearest neighbors, where k can be any element in $[k_{min}, k_{max}]$. In this domination tree, each $oc$ will dominate the points listed behind it, while it in turn will be dominated by the points listed in front of it by the transitive property of the domination relationship. Therefore D-Space is represented by a *domination forest* composed of multiple *domination trees*.

Furthermore *domination forest* also incorporates the *stable region* concept of P-Space along its linkage to D-Space.

**Lemma 24.10** *Given two adjacent points $oc_i$ and $oc_{i+1}$ in domination tree $tree_l$, any parameter setting $ps_x(k_x, r_x)$ with $k_{min} \leq k_x \leq k_{max}$ and $D^{k_x}_{oc_i} \leq r_x < D^{k_x}_{oc_{i+1}}$ will classify the same set of points $oc_j$ in $tree_l$ as outliers, where j > i.*

**Proof.** Since $D^{k_x}_{oc_i} \leq r_x < D^{k_x}_{oc_{i+1}}$, by Lemma 24.1 $ps_x$ will classify $oc_{i+1}$ as outlier and $oc_i$ as inlier. Since $oc_{i+1}$ dominates $oc_j$, all $oc_j$s are outliers. Any other point $oc_h$ in $tree_l$ will be classified as inlier because $oc_h$ dominates inlier $oc_i$. Lemma 24.10 is proven.■

As shown in Fig. 24.3, the parameter settings bounded by lines of $oc_1$ and $oc_2$ generate the same set of outliers: $oc_2$, ..., $oc_5$.

## 24.3.2 Online Outlier Exploration

The domination forest can efficiently support all classes of outlier exploration operations.

**Comparative Outlier Analytics (CO).** CO can be supported by locating the first point $p_{1st}$ in each domination tree dominated by the weakest outlier $o_i$ in the outlier input set $\mathbb{O}_{in}$ using a binary search style algorithm. Then all points listed behind $p_{1st}$ in each of the domination trees are guaranteed to be outliers.

**Outlier Detection (OD).** Similar to CO, OD can be supported by applying the binary search style algorithm on each domination tree to locate the first outlier candidate classified as outlier by the input parameter setting $ps_i$.

The time complexity of processing CO and OD is $O(log \mid tree_1 \mid + log \mid tree_2 \mid +...+ log \mid tree_n \mid)$. It relies on the size of each tree and the number of the trees.

**Outlier-Centric Parameter Space Exploration (PSE).** By Lemma 24.10, the parameters that generate the same outliers $\mathbb{O}_{in}$ can be located by examining the strongest outlier $oc$ in $\mathbb{O}_{in}^i$ and the first point in front of $oc$ in each domination tree $tree_i$. Here $\mathbb{O}_{in}^i$ = $\mathbb{O}_{in} \cap tree_i$. The intersection of the parameters returned from each tree will be the final result of PSE. The time complexity is $O(n+ \mid \mathbb{O}_{in} \mid)$, where n is the number of the trees.

In summary the time complexity of the online phase relies on the size of each tree and the number of the trees. It is easy to see that the smaller the number of the trees is, the lower the costs will be.

As for the size of each tree, suppose two forests $ft_1$ and $ft_2$ composed of the same number of trees are derived from outlier candidate set $\mathbb{OC}$. For forest $ft_1$, $\mid ft_1.tree_1 \mid \gg \mid ft_1.tree_2 \mid ... \gg \mid ft_1.tree_n \mid$, while for forest $ft_2$, $\mid ft_2.tree_1 \mid \approx \mid ft_2.tree_2 \mid ... \approx \mid ft_2.tree_n \mid$. Then the cost of the binary search amounts to $binary(ft_1) < binary(ft_2)$.

For example suppose $\mathbb{OC}$ contains $2^m$ points. $ft_1$ consists of two trees including the largest possible tree $\mid ft_1.tree_1 \mid = 2^m - 1$ and the smallest tree $\mid ft_1.tree_2 \mid = 1$, while $\mid ft_2.tree_2 \mid = \mid ft_2.tree_2 \mid = 2^{m-1}$. Then $binary(ft_1)$ = $\log(2^m - 1)$ + 1 $<$ m +1, while $binary(ft_2)$ = $2\log(2^{m-1})$ = 2(m-1). Obviously when m is reasonably large, $binary(ft_1)$ is far smaller than $binary(ft_2)$. Therefore instead of making each tree equal size, the ideal forest construction algorithm should produce the largest possible trees out of $\mathbb{OC}$.

As shown in Sec. 24.3.1 our graph-based D-Space construction algorithm (Alg. 10) not only minimizes the number of trees created, but also maximizes the size of the trees in the forest. Therefore it effectively optimizes the performance of outlier exploration.

# 25

# Performance Evaluation

**Environment.** All experiments ran on a Linux Server with 8 GB memory 2.6GHz Quad-Core CPU using Java 1.6.0 64bit runtime.

**Real Datasets.** We utilize the GMTI (Ground Moving Target Indicator) dataset [81] to conduct user study. GMTI contains around 10,000 records regarding the information of soldiers, vehicles, and helicopters deployed in a certain region. The outliers are detected based on targets' latitude and longitude. We use the outliers manually labeled by the experts familiar with the data as ground truth.

We also use the geolocation data from OpenStreetMap (http://download.geofabrik.de/) to evaluate the performance of ONION when handling large dataset. It contains the geolocation information of 50 million buildings (10G) over Australia and Oceania, such as houses, cafes, stations, etc.. A location on the map is considered to be outlier based on their distances to other locations.

**Methodology.** We evaluate the processing time and scalability of both our offline preprocessing and online mining algorithms by varying the sizes of the dataset D, parameter space $\mathcal{P}$, and the number of mining requests. We compare against the state-of-the-art DOLPHIN [44] in a rich variety of representative use cases.

In particular at the **offline** phase, we evaluate the processing time of constructing O-Space that builds the foundation of ONION in comparison to the index construction cost of DOLPHIN. At the **online** phase, the performance of our online algorithms associated with O-Space, P-Space, and D-Space respectively is evaluated and contrasted for all three outlier exploration types, namely outlier detection (OD), outlier-centric parameter space exploration (PSE), and comparative outlier analytics (CO). The algorithms associated with each ONION abstraction are named in the format of "operation type" + "_" + "Abstraction type". For example the algorithm supporting OD operation on O-Space is named as "OD_OSpace". Furthermore we also compare our ONION against DOLPHIN on the processing time of traditional outlier detection query − the only exploration type that Dolphin supports.

## 25.1 User Study

We conduct a user study to evaluate the effectiveness of ONION in recognizing outliers contrasting against the traditional one-at-a-time query approach (TRAD) that only supports outlier detection operation. Since TRAD takes hours to process a large dataset (10G) as confirmed in Sec. 25.3.1, it is not acceptable for interactive analytics. Therefore in this study we adopt the relative small dataset (GMTI) − a clear bias to TRAD.

We invited 50 users from both WPI and Yantai University, China. The users are divided into two groups. Each group only evaluates one system. Each user is allowed to continuously submit mining requests supported by the target system until the generated results meet the precision and recall requirement (0.9,0.9) set by us. In each round the precision and recall are automatically calculated and feedbacked to the users. In any case the study will terminate after 15 minutes. Users are provided a distribution plot of GMTI dataset that assists them to initialize the parameter setting. For each user, we count the number of trials (the submitted mining requests) on each exploration operation. Then the trial number is averaged on the users belonging to the same group.

| System | Success Rate | Overall | OD | CO | PSE |
|---|---|---|---|---|---|
| **ONION** | 1 | 5.6 | 1.8 | 2.6 | 1.2 |
| **TRAD** | 0.36 | 16.2 | 16.2 | – | – |

**Table 25.1:** ONION: Success Rate Statistics

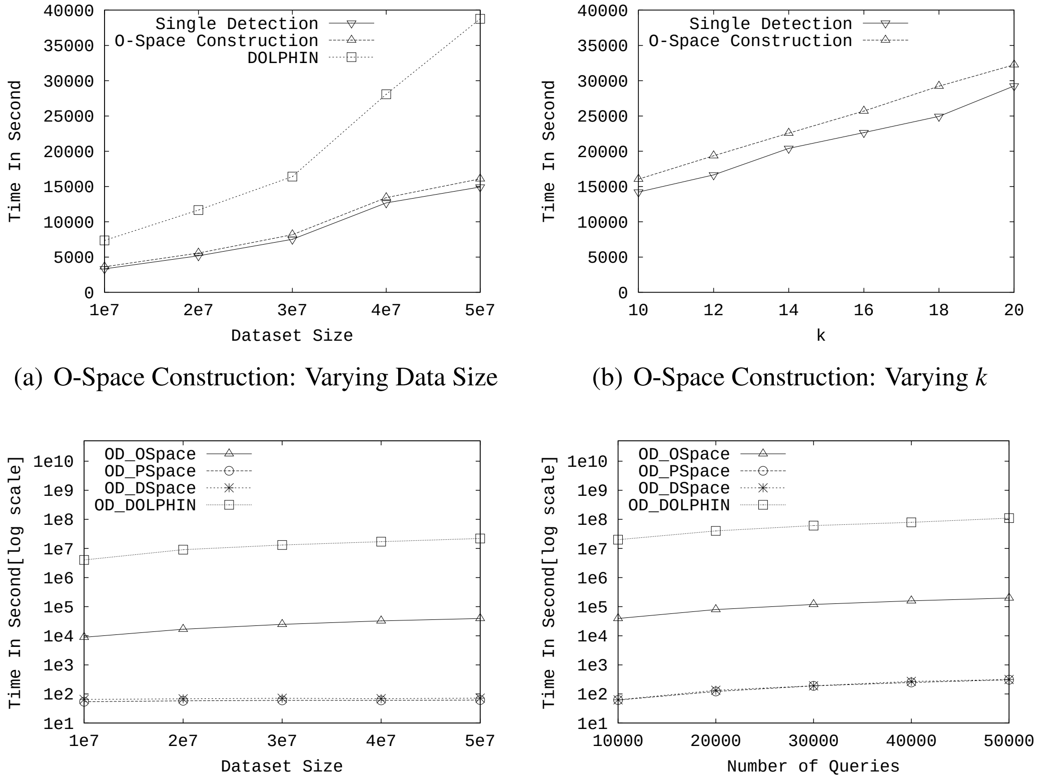
As shown in Table 25.1, only 36% of the TRAD users are able to eventually meet the precision and recall requirement in 15 minutes, while all users using ONION succeed. In average TRAD takes users 16.2 trials to meet the requirement, while the ONION users only need 5.6. In particular in average the ONION users submit CO operation 1.8 times, PSE operation 2.6 times, and the traditional outlier detection (OD) 1.2 times. This confirms that our new outlier exploration operations indeed save users significantly effort on pinpointing appropriate parameter settings.

## 25.2 Offline Preprocessing

### 25.2.1 O-Space Construction

We first focus on the processing time of constructing O-Space (construct_OSpace) from raw data by varying the parameter space size, as well as the dataset size. The costs of one time outlier detection without employing any index is used as the baseline to evaluate the extra overhead introduced by constructing O-Space.

**Varying dataset size.** Fig. 25.1(a) illustrates the results when dataset size increases from 10 million up to 50 million. We vary the dataset size by including more and more buildings belonging to different regions in Australia. The parameter space is fixed with $k_{max}$ as 10 and $r_{max}$ as 4000. Clearly constructing O-Space has ignorable overhead compared to the cost of one time outlier detection when parameter setting *ps* specified as



(a) O-Space Construction: Varying Data Size

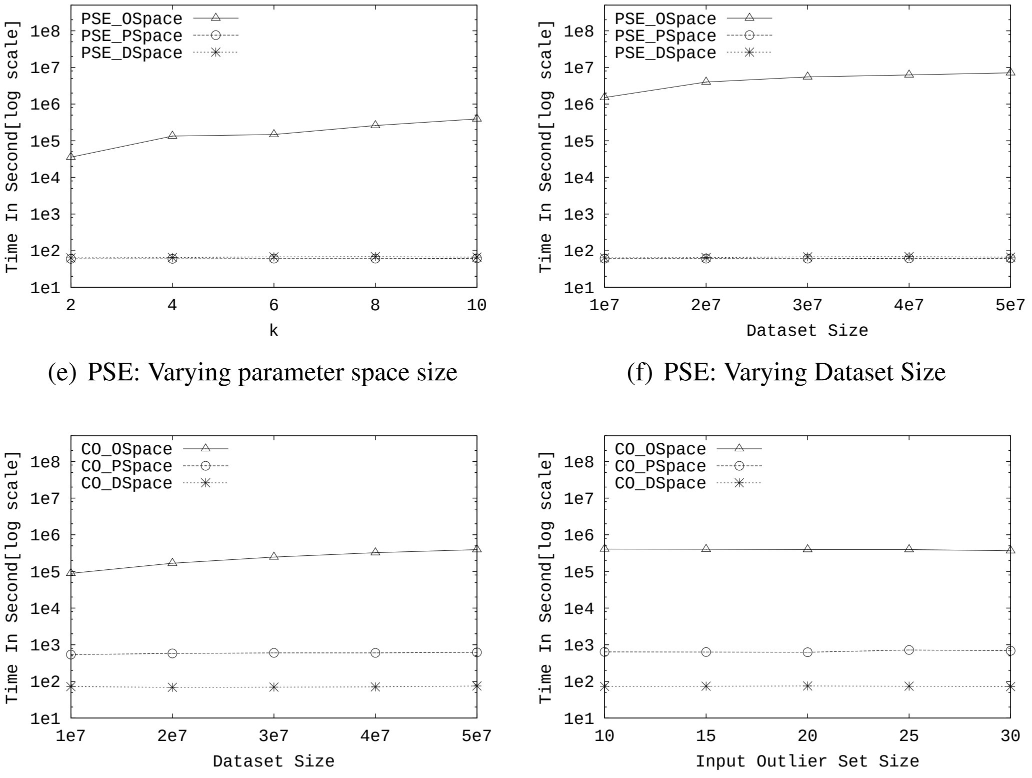
(b) O-Space Construction: Varying $k$

(c) OD: Varying Dataset Size

(d) OD: Varying Number Of Requests

($k_{max}$,$r_{min}$). Both our O-Space construction process and one time detection process need to detect up to $k_{max}$ neighbors within $r_{min}$ radius for each point $p_i$. The additional overhead of O-Space construction is introduced by having to track and maintain all possible outlier candidates with respect to the entire parameter space. However as shown in Fig. 25.1(a) such overhead is small (around 10%). Furthermore constructing O-Space is significantly faster than constructing DOLPHIN index.

**Varying parameter space** $\mathcal{P}$. Dolphin is excluded from this case because it does not have the *parameter space* concept. The influence of varying range of k is evaluated. Fig. 25.1(b) represents the results when varying $k_{max}$ from 10 to 20, while holding $r_{max}$ at 4000m and dataset size at 50 million. The overhead is still around 10% for the same reason explained above. As $k_{max}$ increases, the cost of O-Space construction grows in



(e) PSE: Varying parameter space size

(f) PSE: Varying Dataset Size

(g) CO: Varying Dataset Size

(h) CO: Varying input outlier set size

the trend similar to one time outlier detection. Varying the range of $r$ shows the similar influence. Due to space constraint, the results are not included.

## 25.3 Online Outlier Exploration

### 25.3.1 Online Outlier Detection

We evaluating the processing time of online outlier detection by varying the size of the datasets and the number of the requests.

**Varying dataset size.** Fig. 25.1(c) shows the advantage of ONION for outlier detection. We ran 10,000 requests with randomly chosen parameter settings from the entire parameter space and show the total processing time. P-Space and D-Space methods show very similar performance. Therefore their lines in Fig. 25.1(c) are overlapped. In average each request can be processed in milliseconds. Both consistently outperform DOLPHIN 5 orders of magnitude. Furthermore for D-Space and P-Space the detection cost grows only logarithmically in the size of outlier candidates that are the strict minority of the whole dataset (fewer than 10%), while DOLPHIN grows linearly. Therefore, ONION scales to large dataset.

**Varying number of request.** We increase the number of OD requests from 10,000 up to 50,000, while holding dataset size constant at 50 million. The total detection time is measured. Fig. 25.1(d) shows that our algorithms scale linearly in the number of requests. Again P-Space and D-Space algorithms are at least 5 orders of magnitude faster than DOLPHIN. Even our linear complexity O-Space method is 3 order of magnitude faster than DOLPHIN in average.

### 25.3.2 Outlier-Centric Parameter Space Exploration

We evaluate the performance of processing PSE request not supported by DOLPHIN. Each chart shows the accumulated processing time for 10,000 requests.

**Varying parameter space size.** Fig. 25.1(e) measures the influence to the processing time of PSE when varying the size of the parameter space. This is achieved by increasing $k_{max}$ from 2 to 10. For P-Space and D-Space the cost of supporting PSE relies on the number of domination trees and the parameter node lists. Therefore, the cost of P-Space and D-Space is not sensitive to the change of $k_{max}$. On the other hand O-Space method has to check all outlier candidates. Since the number of outlier candidates grows as $k_{max}$ increases, the cost of O-Space method will also increase lineally.

**Varying dataset size: outlier set as input.** Fig. 25.1(f) demonstrates the performance of our PSE algorithms. That is, given a PSE request, we use a set of randomly selected outlier candidates as input. The size of the datasets is varied from 10 million up to 50 million. Similar to the experiment that uses parameter settings as input, P-Space and D-Space methods significantly outperform O-Space method 3 orders of magnitude.

### 25.3.3 Comparative Outlier Analytics

Next we evaluate the performance of supporting CO operation.

**Varying dataset size.** Fig. 25.1(g) illustrates the processing time of supporting CO operation by varying dataset sizes. We use a randomly selected outlier set as input. The operation returns all outlier candidates that are dominated by the input outliers. D-Space supports CO operation by only looking at each domination tree in the domination forest once, while the number of the domination tree is small (at most 3 when the dataset contains all 50 millions buildings). On the other hand, O-Space method has to scan all candidates, while P-Space method has to search the parameter node lists for every possible k value. Therefore D-Space method is about 1 order of magnitude faster than P-Space method, and about 3 to 4 orders of magnitude faster than O-Space method.

**Varying size of input outlier set.** In Fig. 25.1(h) we vary the size of the input outlier set from 10 to 30, while keeping the sizes of dataset and parameter space stable. For each method we only need to check the weakest outlier of the input outlier set. Since the cost of determining the weakest outlier is negligible, all our three methods are not sensitive to the size of the input outlier set.

# 26

# Related Work

**Outlier Detection.** Outlier detection has been the focus of much research in the statistics literature for over a century [82, 83]. The most common approach is to assume that all points follow a distribution with known distribution parameters (e.g., mean and variance). The points that do not properly fit the model are considered to be outliers. However, such approaches suffer from the serious limitation that the data distribution and underlying parameters must either be explicitly known apriori or be easily inferred.

Approaches that do not rely on data distributions have also been proposed. In [63, 84, 85] all points that are not a core part of any cluster are classified as outliers. In other words the outliers are in this case the by-products of data clustering. However we note here that a point that is not a member of any cluster is not necessarily abnormal. This is so because the goal of clustering is to group together points that are extremely similar to one another. Therefore such approaches lack strong notion of what constitutes an outlier.

To address this limitation, the notion of an outlier based on density (of neighborhood) or based on distance (of neighbors) has been defined. Density-based approaches [10, 11] assign an outlier score to any given point by measuring the density relative to its local neighborhood restricted by a pre-defined threshold. Therefore density-based outliers, regarded as “local outliers”, are able to identify outliers often missed by other methods. However it has been observed that such methods do not scale well to large datasets [86].

Furthermore explicit distance-based approaches, based on the well known nearest-neighbor principle, were first proposed by Ng and Knorr [8]. They employ a well-defined distance metric to detect outliers, that is, the greater is the distance of the point to its neighbors, the more likely it is an outlier. The basic algorithm for such distance-based definition, the nested loop (NL) algorithm, calculates the distance between each pair of points and then set as outliers those that are far from most points. The NL algorithm has quadratic complexity with respect to the number of points. Thus it is not suitable for truly large datasets.

As a result, extensive effort has been focusing on identifying practical sub-quadratic algorithms [36, 44, 53, 87]. Several optimization principles have been proposed such as the use of compact data structures [87], of lightweight outlier detection oriented indices [44], and of pruning and randomization [53]. In particular by indexing the possible neighbors of each point $p_i$ in dataset D based on their distances to $p_i$, [44] is able to approximate whether $p_i$ is an outlier in the time complexity near linear to the cardinality of D. However, while these methods offer improved performance compared to statistical or clustering based approaches, they still suffer from unacceptable response times such that hours or even days for online queries. Furthermore none of these works tackles the important and hard problem of choosing proper parameter setting from the infinite number of possible options. Our work not only successfully satisfies the real time responsiveness requirement, but also saves users the significant effort otherwise spent on parameter tuning.

**Parameter Space Exploration in Clustering.** In [42] the *OPTICS* algorithm creates an augmented ordering of the dataset to represent the clustering structure corresponding to a set of parameter settings. However, the producing of outliers as by-products of clustering has already been shown to be not effective in capturing abnormal phenomena [18]. Furthermore the ordering information is only effective in representing the clusterings with respect to a small range of parameter settings, that is the parameters with only the neighbor range threshold variable. Our work instead supports a full range of possible parameter settings composed of both range and neighbor count thresholds.

# Part VI

# Conclusion and Future Work

# 27

# Conclusion of This Dissertation

The goal of this dissertation is to fill the void in the literature of effectively detecting outliers over big data. Outlier detection is fundamentally important in a wide range of applications from credit fraud prevention, network intrusion detection, stock investment tactical planning to moving object monitoring. In this dissertation we propose novel techniques and systems to address the challenges caused by the *high velocity streaming data*, *the big volume static data* and *the large cardinality of the input parameter space*. The three highlights of this dissertation can be summarized as follows.

Within this scope, we focus on three research aspects, namely techniques and systems for continuous outlier detection over streams, distributed outlier detection over massive-scale static data sets and interactive outlier exploration.

First, we focus on the problem of *continuous outlier detection over streams*. We propose approaches to support not only single particular outlier detection request, but also large outlier analytics workload.

For single outlier detection request we propose the LEAP technique, a general stream outlier detection framework that achieves near real time responsiveness even when handling high velocity streaming data. LEAP offers 3 key innovations. (1) We propose a "minimal probing" strategy, which intelligently use a light-weight operation, called "probing", to gather minimal evidence for outlier detection. (2) We propose a "lifespan-aware prioritization" strategy, which leverages the temporal relationships among stream data points to prioritize the processing sequences among them during the probing process. Our comprehensive experimental studies, using both synthetic as well as real streaming data from stock market and moving object domains, confirm that our methods perform up to 3 magnitude faster than the state-of-the-art.

To support large outlier analytics workload, we propose a shared execution methodology called *SOP* that achieves minimal utilization of both computational and memory resources. The key innovations of LEAP includes: (1) transform the problem of handling a *multi-query outlier* analytics workload into a *single-query skyline* computation problem; (2) design a customized skyline algorithm called K-SKY to minimize the number of data points that must be evaluated for supporting multi-query outlier detection. 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.

Second, we target *distributed outlier detection over massive-scale static data sets*. Our DOD system offers 3 key innovations: (1) an efficient *cost-driven* data partitioning strategy to achieve load balancing across compute nodes; (2) a novel *multi-tactic* strategy which adaptively selects the most appropriate algorithm for each partition based on data characteristics; (3) theoretical results on proving the traditional frequency-based load balancing assumption in the literature is not effective. Our comprehensive experimental study confirms the efficiency of *DOD* and its scalability to TBs of data and large number of compute nodes.

Lastly, we address the problem of interactive outlier exploration. Our ONION system offers two key innovations. (1) ONION features an innovative interactive anomaly exploration model that offers an “outlier-centric panorama” into big datasets along with rich classes of exploration operations. (2) To achieve this model ONION employs an online processing framework composed of a one time offline preprocessing phase followed by an online exploration phase that enables users to interactively explore the data. Our user study with real data confirms the effectiveness of ONION in recognizing “true” outliers. Furthermore as demonstrated by our extensive experiments with large datasets, ONION supports all exploration operations within milliseconds response time.

# 28

# Future Work

## 28.1 Predicates in Outlier Detection

To date, predicates have not yet been considered in the outlier detection context. Yet predicates are crucial in expressing the semantics of outliers [33]. For instance, in the stock market investors may look for the outlier stocks whose behavior significantly differ from that of the majority of their peer stocks to seek short-term investment opportunities. However some analysts might only be interested in energy related stocks, while others might be seeking for opportunities in high tech stocks. Therefore outliers have to be detected in a certain subset of the data instead of the whole data set. It is not only because of the saving of the detection time. Moreover, it will significantly influence the accuracy of the generated results. Taking the distance-based outlier concept as example, certainly Apple will get more neighbors (stocks with similar stocks) in the whole stock market than in the high tech sector and be classified as inliers, although it might perform significantly different from other high tech stocks and therefore should be classified as outliers.

Furthermore, in light of the demands from the real world, outliers to be detected sometimes are not only being confirmed as abnormal from dataset they belong to. Instead, their identification of the outlier status depends on some other correlated dataset. Therefore datasets can be categorized into target and scope based on their roles in outlier detection. Target is the dataset where data reside to be evaluated if they are outliers or not. Scope is the stream that all target data probe into to find neighbors. Therefore the traditional outlier detection semantics have to be enhanced to support predicates and Target/Scope datasets.

Therefore a system that supports hundreds of outlier detection queries with different predicate conditions has to be designed. We plan to design a sharing-aware approach to avoid applying outlier detection algorithm over and over for different outlier detection requests.

The main intuition of how to tackle the sharing problem for varying *Target* predicate is as follows. Namely, we utilize the predicates $p_1$, $p_2$, $p_3$, ..., $p_n$ to partition the data points in a dataset into disjoint subsets, called fragments. For each data point in each fragment, neighbors searching would be applied. In other words, a set of data points are partitioned into $F_0$, $F_1$, ..., $F_k$, a set of $k + 1$ disjoint fragments: *D* = $F_0 \cup F_1 \cup$ ... $\cup F_k$. Each fragment $F_i$ is associated with a subset of the workload $WL_i$ where every data point in the fragment $F_i$ satisfies the predicates of every query in $WL_i$. Then outlier detection algorithm can be applied to individual fragment to detect outliers. By this each fragment is processed only once even if it is included by multiple outlier detection queries.

In the arbitrary *Scope* predicate case, the same intuition can be applied to avoid the duplicate probing on each fragment. However in the arbitrary *Target* predicate case, it does not matter which fragment is looked at first and which one is examined later. This is so because all fragments have to be examined no matter in what order they are processed. However in the arbitrary *Scope* case, the processing order should be prioritized, since possibly only evaluating a subset of the Target fragments is sufficient to prove the inlier status of one target point with respect to all queries. Given a *Scope* fragment, the larger number of queries it is involved in, the higher priority it should be assigned to. The intuition here is that a *Target* fragment involved in more queries potentially can satisfy the inlier criteria of more queries for a given *Target* point.

## 28.2 Approximation in Outlier Detection

Approximation forms another category of strategy that can speed up the outlier detection process [14]. Various approximation approaches can be leveraged such as dimension reduction, sampling, etc. The key of such approach is to quickly approximate outliers, while still guarantee the accuracy of the detected results. Here we roughly sketch several possible approaches. We use LOF as example, because LOF is one of the most expensive techniques in unsupervised outlier detection.

**Approximate KNN on the one dimensional distance space.** We first map all data points in a given dataset $D$ to a one dimension space using their distances to one reference point. Then the KNN of each data point $p$ in $D$ can be approximated using its KNN in the one dimensional distance space. Multiple reference points, therefore multiple one dimensional distance space can be utilized to reduce the approximation error. Since KNN search is the bottleneck of LOF calculation, this approach will significantly speed up the LOF calculation process.

**Approximate LOF values using sampling.** Since the LOF value of one given point $p$ is calculated based on the relative density relative to its neighbors, utilizing a sample data set to approximate the density of $p$ and in turn its LOF value might not significantly deviate from its actual value. More specifically given a large input data set $D$, we first acquire a small data set $D'$ by randomly sampling $D$. Then the KNN of each data point $p \in D$ will be searched in the small sample data set $D'$. This is much more efficient compared to the KNN search conducted on the original large data set $D$.

**Distributed approximation approach.** First, the input data set $D$ is divided into multiple data partitions $D_i$. Each partition is processed by one compute node. Then the LOF values of each point $p \in D_i$ will be calculated based on the data in the same node. If the LOF value of point $p$ is smaller than a pre-defined threshold *t*, p is considered as an inlier and discarded immediately. On the other hand if the LOF value of *p* is larger than *t*, p has to be processed in the next round to confirm whether it is indeed an outlier. Since the number of such points tend to be small, it is practical to broadcast them to all other partitions and then calculate their actual KNNs and LOF values.

## 28.3 The Management of Outliers

Although outliers are considered to be the absolute minority of the input data, the sheer number of outliers detected in a big data set still can be overwhelming. Therefore manually managing and analyzing the large number of outliers is not feasible anymore in this big data era. Therefore an outlier management system must be designed to help users effectively maintain and analyze the detected outliers.

However designing such an outlier management system is challenging. First, a popular approach of managing a large dataset is to group the data points with the similar characteristics together and manage the data at the group level instead of the individual point level, in other words managing data by clustering. However since outliers by definition are the individual anomalous phenomenon detected in the data, not many of them will share the same characteristics. Otherwise they would not be recognized as outliers. Therefore clustering outliers is difficult if not impossible.

Furthermore, to the best of our knowledge all outlier detection models require users to set input parameters as outlier specification. An appropriate input parameter setting is the key to discover the “true” outliers to the interest of the users. Very possibly users have to successively submit a lot of mining requests with different input parameter settings until a good parameter is eventually located. During this process the intermediate detection results have to be maintained such that the users can easily contrast and compare the outliers generated in different rounds. This further complicates the outlier management system, since each data point will be associated with multiple outlier status with respect to different outlier specifications.

Next we roughly sketch several possible approaches for this outlier management problem.

First, given a data point *p*, its outlier status can be modeled as a time series with the parameter setting as the "time" and the outlierness score as the "value". Then outliers showing similar evolving patterns along the parameter settings can be grouped together by applying time series clustering algorithms. In other words this approach clusters outliers based on the evolvement trend of their outlierness score such as the LOF value.

Second, given a data set *D*, the points in *D* and their outlierness scores corresponding to various parameter settings can be modeled as one matrix. More specifically in this matrix one dimension represents the data points in *D*. The other dimension represents the input parameter settings. The value of each element in this matrix represent the outlierness score of one particular data point corresponding to one particular parameter setting. Then a *bi-clustering* [88] algorithm can be applied to this matrix. Both the data points and the input parameters will be clustered into multiple groups. Furthermore, this bi-clustering based approach also explicitly models the linkage between the data points and the input parameter settings. This provides users a powerful yet simplistic tool to analyze and contrast the generated outliers across different parameters and therefore would significantly speed up the process of locating an appropriate input parameter setting. By this we solve *two* critical problems with one *single* approach , namely the management of the detected outliers and the recommendation of the input parameter settings.

## 28.4 Continuous Detection of Local Outliers

In Part II we present the solution of detecting distance-based outliers over data streams. Supporting the continuous detection of density-based outliers in particular LOF will be one important direction of our future research, since LOF is able to discover "local outliers" which cannot be captured by distance-based outlier models as discussed in Chapter 2.

Since the time complexity of LOF calculation is quadratic to the number of the input data points, periodically applying static LOF outlier detection algorithm to the continuously evolving streaming data would be extremely computationally inefficient. To satisfy the stringent response time requirement of stream applications incremental solutions must be designed.

The high computation cost of LOF in fact is introduced by the expensive $k$NN search conducted on each data point. Therefore an incremental streaming $k$NN algorithm will significantly improve the performance of LOF calculation if it is able to effectively maintain and update the $k$NN of a given point as the streaming data evolves. The intuition here is that given a point $p$ usually only a very small fraction of the data points arriving from a data stream is relevant to the $k$NN of $p$. For high throughput streams also only a small fraction of the points can be stored in the system. These general conditions require a decision strategy to discard irrelevant points. In other words we need to develop a criterion to decide upon arrival of a new object from the stream if it may become the nearest neighbor in future, or can definitely not. Among the stored potential relevant points pruning is performed. The basic idea is that a new point arriving from the stream can often exclude many other points which can not become nearest neighbors until the new arrived point expires.

Assume k in $k$NN search is equal to 1. That is, we are looking at the *nearest neighbor* of point p. To decide whether or not an point $p_i$ may become the nearest neighbor of a point *p* we consider the 2-dimensional space of the distance and the time of expiry. Note that this space is always 2-d, regardless of the dimensionality of the feature space of the data. An point $p_i$ may become the nearest neighbor, unless there exists another point $p_j$ which at the same time (1) is closer to point $p$ than $p_i$ and which (2) expires later than $p_i$. If both conditions hold, then $p_i$ cannot be the nearest neighbor now (because at least one closer point $p_j$ is known.) Moreover, it cannot become the nearest neighbor later, because the point $p_j$ lives longer. A set of points which are maximal (minimal) with respect to two (or more) different conditions (such as, in this case, distance and expiry) is called a skyline. It can be proven that the points of the skyline in the distance-expiry space (and only these points) need to be stored as potential nearest neighbors. The current nearest neighbor is also in the skyline at all times. This immediately indicates an incremental streaming $k$NN search algorithm. That is, by maintaining the skyline, we only need to search the new arrivals to update the $k$NN of *p*.

Moreover, the arrival of the new point as well as the expiration of the old point influences only limited number of their closest neighbors. Thus the number of LOF value updates per such insertion/deletion does not depend on the total number of points in the data set. To quickly locate the points being influenced by for example the arrival of a new point *p*, we have to track the points that have *p* as their $k$NNs, in other words the reverse $k$NN of *p* or in short $RKNN$. Therefore to quickly exclude the points whose are influenced by the arrival of the new point or the expiration of the old point *p*, the $RKNN$ should be maintained instead of the $k$NN of *p*.

The observation here is that in fact the streaming $RKNN$ problem is similar to the stream distance-based outlier detection problem. This is so because given a $RKNN$ query with the query point set as *q*, most of the points in the data set are not the RKNN of *q*. In other words similar to outliers, the $RKNN$ points are also the absolute minority of the entire data set. Furthermore, given a point *p*, if it has *k* succeeding neighbors whose distances to *p* is smaller than the distance between *p* and *q*, *p* will never be in the *RKNN* of *q*. Therefore the minimal probing and lifespan-aware prioritization optimization principles introduced in Chapter 3 could be equally applied to support streaming $RKNN$.

In summary, it would be interesting and promising to explore these four directions. We hope the experimental evaluation can confirm the superiority of the approaches/models above. Eventually, future work in this direction will significantly enhance the applicability of our next generation outlier analytics system to a wider spectrum.

# References

[1] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection. *ACM Computing Surveys*, 41(3):1–58, 2009. 1, 2, 11, 14

[2] Vic Barnett and Toby Lewis. *Outliers in Statistical Data*. Wiley, 1994. 2

[3] Eleazar Eskin. Anomaly detection over noisy data using learned probability distributions. In *ICML*, pages 255–262, 2000. 2

[4] Manuel Davy and Simon J. Godsill. Detection of abrupt spectral changes using support vector machines: an application to audio signal segmentation. In *ICASSP*, pages 1313–1316, 2002. 2

[5] Wenjie Hu, Yihua Liao, and V. Rao Vemuri. Robust anomaly detection using support vector machines. In *In Proceedings of the International Conference on Machine Learning*. Morgan Kaufmann Publishers Inc, 2003. 2

[6] Kaustav Das and Jeff G. Schneider. Detecting anomalous records in categorical datasets. In *SIGKDD*, pages 220–229, 2007. 2

[7] Simon Hawkins, Hongxing He, Graham Williams, and Rohan Baxter. Outlier Detection Using Replicator Neural Networks. *Neural Networks*, pages 170–180, 2002. 2

[8] Edwin M. Knorr and Raymond T. Ng. Algorithms for mining distance-based outliers in large datasets. In *VLDB*, pages 392–403, 1998. 2, 12, 14, 22, 23, 57, 63, 110, 120, 121, 125, 142, 208

[9] Sridhar Ramaswamy, Rajeev Rastogi, and Kyuseok Shim. Efficient algorithms for mining outliers from large data sets. In *SIGMOD Conference*, pages 427–438, 2000. 2, 14, 16, 22, 23, 57, 63, 64, 142

[10] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. Lof: Identifying density-based local outliers. In *SIGMOD Conference*, pages 93–104, 2000. 2, 14, 22, 24, 64, 207

[11] Spiros Papadimitriou, Hiroyuki Kitagawa, Phillip B. Gibbons, and Christos Faloutsos. Loci: Fast outlier detection using the local correlation integral. In *ICDE*, pages 315–326, 2003. 2, 22, 64, 207

[12] Christopher R. Palmer and Christos Faloutsos. Density biased sampling: An improved method for data mining and clustering. In *SIGMOD*, pages 82–92, 2000. 3

[13] George Kollios, Dimitrios Gunopulos, Nick Koudas, and Stefan Berchtold. Efficient biased sampling for approximate clustering and outlier detection in large data sets. *IEEE Trans. Knowl. Data Eng.*, 15(5):1170–1187, 2003. 3

[14] Robson Leonardo Ferreira Cordeiro, Caetano Traina Jr., Agma Juci Machado Traina, Julio López, U. Kang, and Christos Faloutsos. Clustering very large multidimensional datasets with mapreduce. In *KDD*, pages 690–698, 2011. 3, 9, 144, 216

[15] Nesime Tatbul, Ugur Çetintemel, Stanley B. Zdonik, Mitch Cherniack, and Michael

Stonebraker. Load shedding in a data stream manager. *VLDB*, pages 309–320, 2003. 3

[16] Bin Liu, Yali Zhu, and Elke A. Rundensteiner. Run-time operator state spilling for memory intensive long-running queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 347–358, 2006. 3

[17] Bugra Gedik, Kun-Lung Wu, Philip S. Yu, and Ling Liu. Grubjoin: An adaptive, multi-way, windowed stream join with time correlation-aware CPU load shedding. *IEEE Trans. Knowl. Data Eng.*, 19(10):1363–1380, 2007. 3

[18] Gustavo Henrique Orair, Carlos Teixeira, Ye Wang, Wagner Meira Jr., and Srinivasan Parthasarathy. Distance-based outlier detection: Consolidation and renewed bearing. *PVLDB*, 3(2):1469–1480, 2010. 3, 11, 124, 144, 209

[19] Lei Cao, Di Yang, Qingyang Wang, Yanwei Yu, Jiayuan Wang, and Elke A. Rundensteiner. Scalable distance-based outlier detection over high-volume data streams. In *ICDE*, pages 76–87, 2014. 3, 7, 15, 17, 95, 96, 98, 105

[20] Alex Nazaruk and Michael Rauchman. Big data in capital markets. In *SIGMOD Conference*, pages 917–918, 2013. 5, 7

[21] Marcos D. Assuncao, Rodrigo N. Calheiros, Silvia Bianchi, Marco A.S. Netto, and Rajkumar Buyya. Big data computing and clouds: Trends and future directions. *Journal of Parallel and Distributed Computing*, 79-80:3 – 15, 2015. 5

[22] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004. 5

[23] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, pages 1–10, 2010. 5

[24] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28, 2012. 5

[25] Lei Cao, Qingyang Wang, Elke A. Rundensteiner, and Mohamed H. Eltabakh. Distance-based outlier detection at scale. In *In submission*, 2016. 6, 9

[26] Fabrizio Angiulli and Fabio Fassetti. Distance-based outlier queries in data streams: the novel task and algorithms. *Data Min. Knowl. Discov.*, 20(2):290–324, 2010. 7, 8, 16, 34, 64, 105

[27] Maria Kontaki, Anastasios Gounaris, Apostolos N. Papadopoulos, Kostas Tsichlas, and Yannis Manolopoulos. Continuous monitoring of distance-based outliers over data streams. In *ICDE*, pages 135–146, 2011. 7, 8, 16, 17, 34, 50, 51, 59, 64, 95, 98, 105

[28] Dragoljub Pokrajac. Incremental local outlier detection for data streams. In *In Proceedings of IEEE Symposium on Computational Intelligence and Data Mining*, pages 504–515, 2007. 7, 8

[29] Moustafa A. Hammad, Michael J. Franklin, Walid G. Aref, and Ahmed K. Elma-

garmid. Scheduling for shared window joins over data streams. In *VLDB*, pages 297–308, 2003. 8, 106

[30] Sailesh Krishnamurthy, Michael J. Franklin, Joseph M. Hellerstein, and Garrett Jacobson. The case for precision sharing. In *VLDB*, pages 972–986, 2004. 8

[31] Song Wang, Elke A. Rundensteiner, Samrat Ganguly, and Sudeept Bhatnagar. Stateslice: New paradigm of multi-query optimization of window-based stream queries. In *VLDB*, pages 619–630, 2006. 8, 106

[32] Arvind Arasu and Jennifer Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, pages 336–347, 2004. 8, 106

[33] Sailesh Krishnamurthy, Chung Wu, and Michael J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD Conference*, pages 623–634, 2006. 8, 106, 214

[34] Di Yang, Elke A. Rundensteiner, and Matthew O. Ward. A shared execution strategy for multiple pattern mining requests over streaming data. *PVLDB*, 2(1):874–885, 2009. 8

[35] Fabrizio Angiulli, Stefano Basta, Stefano Lodi, and Claudio Sartori. A distributed approach to detect outliers in very large data sets. In *Euro-Par*, pages 329–340, 2010. 8, 9, 12, 120, 142

[36] Kanishka Bhaduri, Bryan L. Matthews, and Chris Giannella. Algorithms for speeding up distance-based outlier detection. In *KDD*, pages 859–867, 2011. 8, 9, 12, 120, 143, 208

[37] Anna Koufakou, Jimmy Secretan, John Reeder, Kelvin Cardona, and Michael Georgiopoulos. Fast parallel outlier detection for categorical datasets using mapreduce. In *IJCNN*, pages 3298–3304, 2008. 8, 12, 120, 143

[38] Matthew Eric Otey, Amol Ghoting, and Srinivasan Parthasarathy. Fast distributed outlier detection in mixed-attribute data sets. *Data Min. Knowl. Discov.*, 12(2-3):203–228, 2006. 8, 12, 120, 143

[39] Ye Wang, Ahmed Metwally, and Srinivasan Parthasarathy. Scalable all-pairs similarity search in metric spaces. In *KDD*, pages 829–837, 2013. 9, 143

[40] Chi Zhang, Feifei Li, and Jeffrey Jestes. Efficient parallel knn joins for large data in mapreduce. In *EDBT*, pages 38–49, 2012. 9, 133, 144

[41] Xika Lin, Abhishek Mukherji, Elke A. Rundensteiner, Carolina Ruiz, and Matthew O. Ward. PARAS: A parameter space framework for online association mining. *PVLDB*, 6(3):193–204, 2013. 10

[42] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: Ordering points to identify the clustering structure. In *SIGMOD*. 11, 208

[43] Viswanath Poosala and Yannis E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 486–495, 1997. 12

[44] Fabrizio Angiulli and Fabio Fassetti. Dolphin: An efficient algorithm for mining distance-based outliers in very large datasets. *TKDD*, 3(1), 2009. 12, 135, 139, 140, 142, 200, 208

[45] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011. 13

[46] Erich Schubert, Arthur Zimek, and Hans-Peter Kriegel. Local outlier detection reconsidered: a generalized view on locality with applications to spatial, video, and network outlier detection. *Data Min. Knowl. Discov.*, 28(1):190–237, 2014. 14

[47] Fabrizio Angiulli and Clara Pizzuti. Fast outlier detection in high dimensional spaces. In *PKDD*, pages 15–26, 2002. 16, 23, 53

[48] Di Yang, Elke Rundensteiner, and Matthew Ward. Neighbor-based pattern detection over streaming data. In *EDBT*, pages 529–540, 2009. 16, 34, 64, 65, 144

[49] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language. *VLDB J.*, 15(2):121–142, 2006. 26

[50] Apache hadoop. `https://hadoop.apache.org/`. Accessed: 2015-12-09. 29

[51] Chetan Gupta, Song Wang, Ismail Ari, Ming C. Hao, Umeshwar Dayal, Abhay Mehta, Manish Marwah, and Ratnesh K. Sharma. Chaos: A data stream analysis architecture for enterprise applications. In *CEC*, pages 33–40, 2009. 50, 95

[52] I. INETATS. Stock trade traces. *http://www.inetats.com/*. 50, 95, 101

[53] Stephen D. Bay and Mark Schwabacher. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In *KDD*, pages 29–38, 2003. 51, 63, 208

[54] Kyriakos Mouratidis and Dimitris Papadias. Continuous nearest neighbor queries over sliding windows. *IEEE Trans. Knowl. Data Eng.*, 19(6):789–803, 2007. 51, 105

[55] Sharmila Subramaniam, Themis Palpanas, Dimitris Papadopoulos, Vana Kalogeraki, and Dimitrios Gunopulos. Online outlier detection in sensor data using nonparametric models. In *VLDB*, pages 187–198, 2006. 65

[56] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for clustering evolving data streams. In *VLDB*, pages 81–92, 2003. 65

[57] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1):41–82, 2005. 69, 73

[58] Yufei Tao and Dimitris Papadias. Maintaining sliding window skylines on data streams. *IEEE Trans. Knowl. Data Eng.*, 18(2):377–391, 2006. 73

[59] Christian Böhm, Beng Chin Ooi, Claudia Plant, and Ying Yan. Efficiently processing continuous k-nn queries on data streams. In *ICDE*, pages 156–165, 2007. 105

[60] Foto N. Afrati, Anish Das Sarma, Semih Salihoglu, and Jeffrey D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. *PVLDB*, 6(4):277–288, 2013. 110, 152

[61] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD*, pages 975–986, 2010. 117

[62] Frank Olken, Doron Rotem, and Ping Xu. Random sampling from hash files. In *SIGMOD*, pages 375–386, 1990. 118, 163

[63] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: An efficient data clustering method for very large databases. In H. V. Jagadish and Inderpal Singh Mumick, editors, *SIGMOD*, pages 103–114, 1996. 130, 207

[64] Ian Davidson and S. S. Ravi. Agglomerative hierarchical clustering with constraints: Theoretical and empirical results. In *Knowledge Discovery in Databases: PKDD 2005, 9th European Conference on Principles and Practice of Knowledge Discovery*

*in Databases, Porto, Portugal, October 3-7, 2005, Proceedings*, pages 59–70, 2005. 131

[65] Pierre Lemaire, Gerd Finke, and Nadia Brauner. Models and complexity of multibin packing problems. *J. Math. Model. Algorithms*, 5(3):353–370, 2006. 132

[66] Edward Hung and David Wai-Lok Cheung. Parallel mining of outliers in large database. *Distributed and Parallel Databases*, 12(1):5–26, 2002. 142

[67] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jorg Sander. LOF: Identifying Density-based Local Outliers. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 93–104. ACM, 2000. 146

[68] Wei Lu, Yanyan Shen, Su Chen, and Beng Chin Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *Proceedings of the VLDB Endowment*, 5(10):1016–1027, 2012. 155, 174

[69] Edwin M Knox and Raymond T Ng. Algorithms for mining distance-based outliers in large datasets. In *Proceedings of the International Conference on Very Large Data Bases*, pages 392–403, 1998. 173

[70] Sridhar Ramaswamy, Rajeev Rastogi, and Kyuseok Shim. Efficient algorithms for mining outliers from large data sets. In *ACM SIGMOD Record*, volume 29, pages 427–438, 2000. 173

[71] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996. 173

[72] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: an efficient data clustering method for very large databases. In *ACM SIGMOD Record*, volume 25, pages 103–114. ACM, 1996. 173

[73] 173

[74] Kanishka Bhaduri, Bryan L. Matthews, and Chris R. Giannella. Algorithms for Speeding Up Distance-based Outlier Detection. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 859–867. ACM, 2011. 173

[75] Wen Jin, Anthony K. H. Tung, and Jiawei Han. Mining Top-n Local Outliers in Large Databases. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 293–298. ACM, 2001. 174

[76] Chi Zhang, Feifei Li, and Jeffrey Jestes. Efficient parallel knn joins for large data in mapreduce. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 38–49. ACM, 2012. 174

[77] Frank Olken, Doron Rotem, and Ping Xu. Random sampling from hash files. In *ACM SIGMOD Record*, volume 19, pages 375–386. ACM, 1990. 174

[78] Christopher R Palmer and Christos Faloutsos. *Density biased sampling: an improved method for data mining and clustering*, volume 29. ACM, 2000. 174

[79] Xiangrui Meng. Scalable simple random sampling and stratified sampling. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 531–539, 2013. 174

[80] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009. 194

[81] Jr. Entzminger, J.N., C.A. Fowler, and W.J. Kenneally. Jointstars and gmti: past, present and future. *Aerospace and Electronic Systems, IEEE Transactions on*, 35(2):748 –761, Apr. 1999. 200

[82] Douglas M. Hawkins. *Identification of Outliers*. Springer, 1980. 207

[83] Vic Barnet and Toby Lewis. Outliers in statistical data. *International Journal of Forecasting*, 12(1):175–176, 1996. 207

[84] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996. 207

[85] Raymond T. Ng and Jiawei Han. Efficient and effective clustering methods for spatial data mining. In *VLDB*, pages 144–155, 1994. 207

[86] Hans-Peter Kriegel, Peer Kröger, and Arthur Zimek. Outlier detection techniques. In *In Tutorial of the 13th PAKDD*, 2009. 208

[87] Amol Ghoting, Srinivasan Parthasarathy, and Matthew Eric Otey. Fast mining of distance-based outliers in high-dimensional datasets. *Data Min. Knowl. Discov.*, 16(3):349–364, 2008. 208

[88] Inderjit S. Dhillon. Co-clustering documents and words using bipartite spectral graph partitioning. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining, San Francisco, CA, USA, August 26-29, 2001*, pages 269–274, 2001. 218