Shared Execution Strategy for Neighbor-Based Pattern Mining Requests over Streaming Windows

DI YANG, WPI
ELKE A. RUNDENSTEINER, WPI
MATTHEW O. WARD, WPI

In diverse applications ranging from stock trading to traffic monitoring, data streams are continuously monitored by multiple analysts for extracting patterns of interest in real-time. These analysts often submit similar pattern mining requests yet customized with different parameter settings. In this work, we present shared execution strategies for processing a large number of neighbor-based pattern mining requests of the same type yet with arbitrary parameter settings. Such neighbor-based pattern mining requests cover a broad range of popular mining query types, including detection of clusters, outliers and nearest neighbors. Given the high algorithmic complexity of the mining process, serving multiple such queries in a single system is extremely resource intensive. The naive method of detecting and maintaining patterns for different queries independently is often infeasible in practice, as its demands on system resources increase dramatically with the cardinality of the query workload. In order to maximize the efficiency of the system resource utilization for executing multiple queries simultaneous, we analyze the commonalities of the neighbor-based pattern mining queries, and identify several general optimization principles which lead to significant system resource sharing among multiple queries. In particular, as a preliminary sharing effort, we observe that the computation needed for the range query searches (the process of searching the neighbors for each object) can be shared among multiple queries and thus saves the CPU consumption. Then we analyze the interrelations between the patterns identified by queries with different parameters settings, including both pattern-specific and window-specific parameters. For that, we first introduce an incremental pattern representation, which represents the patterns identified by queries with different pattern-specific parameters within a single compact structure. This enables integrated pattern maintenance for multiple queries. Second, by leveraging the potential overlaps among sliding windows, we propose a meta-query strategy which utilizes a single query to answer multiple queries with different window-specific parameters. By combining these three techniques, namely the range query search sharing, integrated pattern maintenance and meta-query strategy, our framework realizes fully shared execution of multiple queries with arbitrary parameter settings. It achieves significant savings of computational and memory resources due to shared execution. Our comprehensive experimental study, using real data streams from domains of stock trades and moving object monitoring, demonstrates that our solution is significantly faster than the independent execution strategy, while using only a small portion of memory space compared to the independent execution. We also show that our solution scales in handling large numbers of queries in the order of hundreds or even thousands under high input data rates.

Categories and Subject Descriptors: H.2.8 [**Database Applications**]: Data Mining

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Pattern Mining, Multiple Query Optimization, Stream Processing

---

This work is supported by the National Science Foundation, under grants CCF-0811510, IIS 0812027 and IIS 1018443.
Author's addresses: D. Yang and E. Rundensteiner and M. Ward, Computer Science Department,WPI.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

ACM Reference Format:
Yang, D., Rundensteiner, A. E., Ward, O. M.. 2011. Shared Execution Strategy for Neighbor-Based Pattern Mining Requests over Streaming Windows. ACM Trans. DataBase. Syst. 9, 4, Article 39 (March 2010), 43 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

1.1. Motivation

Neighbor-based Patterns in Streaming Windows The discovery of complex patterns such as clusters, outliers, and associations from huge volumes of streaming data has been recognized as critical for many domains, ranging from stock market analysis to traffic monitoring. In this work, we focus on shared query execution strategies for neighbor-based pattern mining requests in streaming windows. Neighbor-based pattern mining requests share the important property that their target patterns are defined based on the "neighbor relationships" (links) among objects. Such requirement for identifying neighbors, namely similar or close objects, is essential for many pattern mining tasks. This is because the similarity (distance) is an important interrelationship among objects and thus constitutes a key evidence from which analysts can draw conclusions about the data. In fact, neighbor-based patterns cover a broad range of popular pattern types studied in the literature including density-based cluster detection [Ester et al. 1996; Gorawski and Malczok 2006; Chen and Tu 2007; Cao et al. 2006], distance-based outlier detection [Knorr and Ng 1998; Angiulli and Fassetti 2007], top-k nearest neighbors search (kNN) [Mouratidis and Papadias 2007; Jagadish et al. 2005] and reverse top-k nearest neighbor search (R-kNN) [Achtert et al. 2009; Achtert et al. 2006]. We define the class of the neighbor-based pattern mining requests in our preliminary section (Section 2). Furthermore, we will categorize the data mining tasks in the related work section (Section 8) and show where neighbor-based pattern mining fits in this categorization. related while distinguishable from other data mining tasks.

The previous works listed above have shown that applying such neighbor-based pattern mining requests against streaming windows is of relevance to many important applications. For example, a density-based clustering query can help a financial analyst to continuously monitor the clusters formed in stock transactions within the last 10 minutes, as this may indicate the newest trends in the stock market. A distance-based outlier detection query over sliding windows can help a banker to monitor the potential frauds (outliers) in the latest, say last 1 hour, credit card transactions. A kNN query over sliding windows can help a restaurant owner to monitor the nearest 100 vehicles to her restaurant, whom she would like to send advertisements or promotions to.

In these applications, sliding window semantics need to be applied to form patterns based on the recent portions of the input streams only. Out-of-date information, such as the positions of the vehicles or transactions that were reported a long time ago, should no longer contribute to the recent pattern detection results and must thus be purged from the current query window. For example, the old transactions, namely the transactions that happened 10 minutes ago, will expire (no longer be considered), because the newest trends should only be extracted from the most recent transactions.

Parameterized Queries. Complex pattern detection queries are usually parameterized, because pattern detection processes are driven by the domain knowledge of the analysts and the specific analysis tasks. A neighbor-based pattern mining request over sliding windows typically has two sets of input parameters, namely a set of pattern-specific parameters and a set of window-specific parameters. To illustrate this, we show the query templates for three different types of such queries in Figures 1, 2 and 3. Using the density-based clustering as example, it has two pattern-specific parameters: a

range threshold θ^{range} and a count threshold θ^{cnt}, and two window-specific parameters: window size win and slide size $slide$.

Q_i: ***DETECT Density-Based Clusters FROM*** $stream$
USING $\theta^{range} = r$ ***and*** $\theta^{cnt} = c$
IN Windows WITH $win = w$ ***and*** $slide = s$

Fig. 1. Templated density-based cluster detection query for sliding windows over a data steam

Q_i: ***DETECT Distance-Based Outliers FROM*** $stream$
USING $\theta^{range} = r$ ***and*** $\theta^{fra} = f$
IN Windows WITH $win = w$ ***and*** $slide = s$

Fig. 2. Templated distance-based outlier detection query for sliding windows over a data steam

Q_i: ***DETECT*** $p_i.kNN$ ***FROM*** $stream$
USING K=k
IN Windows WITH $win = w$ ***and*** $slide = s$

Fig. 3. Templated kNN detection query for sliding windows over a data steam

Why Multiple Queries. Given the prevalence of parameterized pattern mining queries, stream processing systems often need to handle a large number of such queries. This is caused for two major reasons. First, it is well known that in many applications a popular data stream is monitored by a large number of analysts [Wang et al. 2006; Zhang et al. 2005; Hammad et al. 2003; Li et al. 2005; Arasu and Widom 2004]. For example, the stock transaction stream from NYSE is monitored by thousands of financial analysts every day. In our case, due to the specific domain knowledge and analytical tasks of different analysts, the analysts may submit the same types of neighbor-based pattern mining queries but with different parameter settings. For example, while many analysts are monitoring the same pattern type, say outliers, in the NYSE stock transaction stream, they may have their own customized interpretation about the pattern mining parameter settings. In particular, some of them may have very strict definitions to what constitutes an outlier in the stream. They may require the system to report only very abnormal transactions (outliers), while others may be interested in all abnormal transaction behaviors and thus request much more frequent updates to the outlier report.

Second, determining a priori the most appropriate parameter settings is a difficult problem for almost all data mining tasks, especially when faced with an unknown input stream or an unpredictable fluctuating input stream. In static environments, this problem is usually tackled by conducting pre-analysis of the static datasets or repeatedly trying different parameter settings until satisfactory results have been obtained. In streaming environments, the nonrepeatability of streaming data requires analysts to supply the most appropriate input parameters early on. Otherwise, they may permanently lose the opportunity to accurately discover the patterns in the portion of the stream just gone by. Therefore, even a single analyst may submit multiple queries of the same type but with different parameter settings, when she is not sure

which parameter setting is the best. An ideal stream processing system should be able to accommodate such multiple query workloads covering many, if not all, major parameter settings of a parameterized query. Note that given the number of parameters required by streaming neighbor-based pattern mining queries, even allowing a very limited number of optional settings on each parameter, say four or five, can easily end up with hundreds of parameter combinations, namely hundreds of different queries.

Research Goal – Multiple Query Optimization. Previous research efforts that have developed efficient algorithms for streaming pattern detection focused on processing single mining requests [Cao et al. 2006; Chen and Tu 2007; Yang et al. 2009a; Angiulli and Fassetti 2007]. Little effort has been made towards the efficient execution of multiple pattern mining queries.

In this work, we aim to present multiple query execution strategies that efficiently execute a workload composed of a large number of neighbor-based pattern mining queries (as defined in 2). More precisely, we aim to design efficient algorithms to simultaneously execute a group of density-based clustering queries or distance-based outliers or kNN queries against the same data stream but with varying parameter settings. Our goal is to achieve real-time responsiveness required by stream applications even when the input rate of the stream is high and the number of queries in the query group is large (on the order of hundreds or even thousands).

1.2. Research Challenges

Execution of even a single neighbor-based mining request in streaming environments is expensive in terms of system resource utilization. In particular, the "neighbor-based" property of such pattern mining requests requires a potentially large number of neighbor searches during query execution. Each neighbor search has high system resource costs. More specifically, a complete neighbor search for even just one single object may take a full scan through the window, consuming not only a large amount of CPU processing resources but also forcing the full storage of the whole window. Given such high algorithmic complexity of neighbor-based pattern mining requests, serving a large number of them in a single system is extremely resource intensive. The naive method of executing multiple queries independently has prohibitively high demands on both computational and memory resources. Thus it is not feasible in practice, especially when the number of queries to be executed is large.

Therefore, the key problem that we solve in this work is to design shared execution strategies that achieve effective sharing of system resources among multiple queries. In particular, we aim to not only minimize the total number of neighbor searches by sharing the neighbor search computation among multiple queries but also to share the maintenance effort for the progressive pattern construction among queries. This is a challenging problem, because the meta-information required to be maintained by neighbor-based pattern queries is generally more complex than that for SQL query operators. More specifically, we need to maintain the identified neighbor-based pattern structures, such as clusters and outliers, which are defined by their member tuples and the global topological relationships among tuples. While SQL operators, such as join or aggregation operators, usually maintain pair-wise relations between two individual tuples (independent from the rest of the tuples) or simply numbers (aggregation results). The techniques introduced previously in the database community regarding sharing among SQL queries [Hammad et al. 2003; Krishnamurthy et al. 2004] are thus not adequate to solve our problem.

1.3. Proposed Solution

In order to maximize the efficiency of the system resource utilization for executing multiple neighbor-based pattern mining queries simultaneously, we analyze the common-

alities of such queries. This helps us to identify several general optimization principles which lead to significant system resource sharing among multiple queries.

As a preliminary sharing effort, we observe that the range query searches (the process of searching for "neighbors" for each object) can be shared among multiple queries and thus save overall CPU consumption (See Section 3). Although this is a straightforward sharing strategy, since the range query searches are frequently needed during neighbor-based pattern mining processes, it constitutes an important multiple query optimization principle for such queries.

However, range query search sharing alone is far from sufficient to achieve the goal of scaling to workloads composed of many such queries within a single system. Therefore, we further analyze the interrelations between the patterns identified by queries with different parameters settings, including both pattern-specific and window-specific parameters. First, we study the conditions under which all queries have the same window parameters. We observe that, if the pattern parameters of a query are "more restricted" than those of another one, a "containment" relationship holds between the patterns identified by them. We exploit this foundation of pattern containment to incrementally organize the patterns identified by multiple queries into an integrated structure. We call it *IntView* (See Section 4). As a highly compact structure, *IntView* saves the memory space needed for storing the patterns identified by multiple queries. More importantly, *IntView* also enables integrated maintenance for the progressive patterns of multiple workload queries, and thus effectively saves the computational resources for maintaining them independently.

Second, we proposed a "*meta query* strategy", which uses a single meta query to represent all workload queries whose pattern parameters are the same while their window parameters differ (See Section 5). The proposed meta query strategy adopts a flexible window management mechanism to efficiently organize the query windows that need to be maintained by multiple queries. By leveraging the overlap among query windows, it minimizes the number of windows that are actually maintained in the system. We show in Section 5 that our meta query technique successfully transforms the problem of maintaining multiple queries into the execution of a single query.

Finally, we combine the range query search sharing, *IntView* technique and *meta query* strategy to form our proposed comprehensive solution for each specific pattern type, namely the shared execution strategies for multiple density-based clustering, distance-based outlier or kNN queries over sliding windows. Computation-wise, all these three proposed algorithms, require only a single pass through the new objects at each window slide. In particular, they only run one range query search for each new object, and each new object only communicates with its neighbors once for a group of shared queries. Memory-wise, given the maximum window size allowed, the upper bound of the memory consumption of our solution for a group of shared queries is independent of the number of queries in the group (see Section 6).

Our experimental study (in Section 7) shows that our proposed solution clearly outperforms all the alternative methods, and has great scalability to a large number of queries. In particular, for density-based clustering queries, the system using our proposed algorithm comfortably handles a workload composed of 100 arbitrary queries under a 1K tuples per second data rate. If the number of workload queries increases to 1K, the system still works stably with a 300 tuples per second input rate. On the same experimental platform, given the 300 tuples per second input rate, the existing execution strategies from the literature, such as *IncDBSCAN* [Ester et al. 1998] and *Extra-N* [Yang et al. 2009a], can only handle less than 1.7 and 12 percent of the same 1K query workload, respectively. Our performance analysis for distance-based outlier and kNN queries shows that a similar performance can be expected from our proposed strategies for those two pattern types as well.

1.4. Contributions

The contributions of this work include:

1) We analyze the commonalities of mining neighbor-based patterns over streaming windows, including three popular pattern types, namely density-based clusters, distance-based outliers and k nearest neighbors (kNN). We identify that for each individual neighbor-based pattern mining query, the process of maintaining progressive pattern structures is the most system-resource-consuming operation of the query processing task. Therefore, it constitutes the operation that would bring the largest performance gain if shared properly among such pattern mining queries.

2) We characterize the notion of pattern containment among neighbor-based patterns. Based on this foundation, we present the integrated pattern maintenance technique for neighbor-based patterns identified by same type of mining queries yet with different pattern-specific parameter settings.

3) We analyze the notion of window overlap in multiple query execution for sliding window queries. Based on this analysis, we present a technique, called meta query strategy, to efficiently share computation among multiple sliding window queries, which detect the same pattern type using same pattern specific parameters yet have arbitrary window specific parameters.

4) We combine those two proposed principles, namely the integrated pattern maintenance and the meta-query strategy, along with range query search sharing, to realize full sharing for multiple neighbor-based pattern mining queries of the same query type in streaming environments.

5) Lastly, our comprehensive experiments on several real streaming datasets confirm the effectiveness of our proposed techniques and also their superiority over the state-of-art alternatives in both CPU time and memory utilization.

Extension from the Conference Version. An earlier version of this article has appeared in a conference [Yang et al. 2009b]. The exposition of this manuscript has been updated significantly to not only ensure easier understanding but also to extend the scope of the conference version. First, the earlier conference version only discussed multiple query optimization techniques for density-based clusters, while in this article we abstract these techniques into general principles to serve as shared query execution strategies for other major pattern types in the neighbor-based pattern family as well. Second, in the earlier version, we keep the discussion of optimization strategies for shared processing among queries with arbitrary window parameters brief, while we now elaborate on it in depth in this manuscript. Third, we present the complete pseudo code for shared execution methods of all three pattern types that we studied in this work, namley density-based clusters, distance-based outliers and kNN queries. Fourth, we add discussion on the performance analysis for distance-based outlier and kNN queries. Fifth, we include additional lemmas and develop proofs for all lemmas and theorems. Sixth, we add more intuitive examples for each of the major techniques proposed.

2. PRELIMINARIES

2.1. Definition for Neighbor-Based Pattern Mining Queries

We now demonstrate that the neighbor-based pattern mining queries tackled by our work correspond to a particular subclass of the general class of graph mining. This subclass of graph mining tasks is composed of two phases, namely graph definiton and graph mining.

Graph Definition Phase. First, unlike the traditional graph mining tasks, in which the target graph structures are given as input, neighbor-based pattern mining queries require a *graph definition* step before mining on the graph. In particular,

given an input dataset D, a neighbor-based pattern mining query first defines a graph $G = (V, E)$, in which $V = D$ corresponds to the set of vertices and E corresponds to the set of edges, modeling all the pair-wise "neighbor relationships" among the vertices. The edges in E can be either directed or undirected. Such *graph definition* step takes two inputs from the query specification.

1): A user-defined distance function $Dist(v_a, v_b)$, $\forall v_a, v_b \in V$, which returns a value $dist\_v_a\_v_b$ reflecting the distance between v_a and v_b.

2): An edge definition function $M(v_a, v_b, dist\_v_a\_v_b)$, which decides whether (true/false) an edge exits $e(v_a, v_b)$ between v_a and v_b. The distance between v_a and v_b, $dist\_v_a\_v_b$, is the base for $M(v_a, v_b, dist\_v_a\_v_b)$ to make this decision, while the specific edge definition mechanism may vary depending on the specific neighbor-based pattern mining query type. For example, $M(v_a, v_b, dist\_v_a\_v_b)$ for density-based cluster [Ester et al. 1996; Ester et al. 1998] and distance-based outlier [Knorr and Ng 1998; Angiulli and Fassetti 2007] mining imposes a range threshold value to define an undirected edge $e(v_i, v_j)$ between v_a and v_b, while $M(v_a, v_b, dist\_v_a\_v_b)$ for k nearest neighbor (kNN) [Mouratidis and Papadias 2007; Jagadish et al. 2005] and reverse k nearest neighbor mining queries [Mouratidis and Papadias 2007] take a count threshold to define directed edges $e(v_a, v_b)$ and $e(v_b, v_a)$ We will introduce the precise edge definition function for each specific neighbor-based pattern type in their formal definitions later in of this section.

Graph Mining Phase. Given the graph G defined in the graph definition phase, in the second phase, each neighbor-based pattern mining query type mines for a particular type of sub-graph(s) in G that exhibits certain characteristics. We will explain the specific sub-graph(s) that each neighbor-based pattern mining query type mines for in the formal definition of each pattern type.

2.2. Definitions for Specific Neighbor-Based Patterns

We use the term *data point* to refer to a multi-dimensional tuple (object) in the data stream. To be consistent with the graph mining problem definition given above, we use v_i to represent each data point in the following definitions.

Definition 2.1. **Density-Based Cluster Mining Query:** Besides the input dataset D and a distance function $Dist(v_a, v_b)$, density-based cluster mining takes two input parameters, namely a range threshod θ^{range} and a count threshold θ^{cnt}.

In the **graph definition phase**, density-based cluster mining defines an undirected edge between any v_a and $v_b \in D$, if $Dist(v_a, v_b) < \theta^{range}$. We say that v_a and v_b are neighbors of each other in this situation. It thus defines an undirected graph $G = (V, E)$, with V corresponding to all data points in D and E corresponding to all the undirected edges among data points in V.

Then, in the **graph mining phase**, we use the function $NumNei(v_i, \theta^{range})$ to denote the number of neighbors a data point v_i has, given the θ^{range} threshold. A data point v_i with $NumNei(v_i, \theta^{range}) \geq \theta^{cnt}$ is defined as a *core point*. Otherwise, if v_i is a neighbor of any *core point*, v_i is an *edge point*. v_i is a *noise* point if it is neither a *core point* nor an *edge point*. Two *core points* v_0 and v_n are connected if they are neighbors of each other, or there exists a sequence of *core points* $v_0, v_1, ...v_{n-1}, c_n$, where for any i with $0 \leq i \leq n-1$, a pair of *core points* v_i and v_{i+1} are neighbors of each other. Each density-based cluster is a group of connected *core points* and the *edge points* attached to them. Density-based cluster mining mines for all such clusters in the defined graph G.

Figure 4 shows an example of a density-based cluster composed of 11 *core points* (black) and 2 *edge points* (grey) in W_0.

Definition 2.2. **Distance-Based Outlier:** Besides the input dataset D and a distance function $Dist(v_a, v_b)$, distance-based outlier detection takes two input parameters, namely a range threshod θ^{range} and fraction threshold θ^{fra}.

In the **graph definition phase**, distance-based outlier mining defines an undirected edge between any v_a and $v_b \in D$, if $Dist(v_a, v_b) < \theta^{range}$. We say that v_a and v_b are neighbors of each other in this situation. It thus defines an undirected graph $G = (V, E)$, with V corresponding to all data points in D and E corresponding to all the undirected edges among data points in V.

Then, in the **graph mining phase**, we use the function $NumNei(v_i, \theta^{range})$ to denote the number of neighbors a data point v_i has, given the θ^{range} threshold. Distance-based outlier mining mines for all data points v_i in the defined graph G, where $NumNei(v_i, \theta^{range}) < |D| * \theta^{fra}$, with N the number of vertices in G.

Definition 2.3. **Top-k Nearest Neighbor Query (kNN):** Besides the input dataset D and a distance function $Dist(v_a, v_b)$, top-k mining query takes a query object v_q and a count threshold k.

In the **graph definition phase**, a top-k mining query defines a directed edge from v_a to $v_b \in D$, if there exist less then k data points v_0, v_1, ... ,$v_{k-1} \in D$ that $dist\_v_b\_v_i < dist\_v_b\_v_a (0 \leq i \leq k-1)$. We say v_b is a neighbor of v_a if there exists an edge from v_a to v_b. It thus defines a directed graph $G = (V, E)$, with V corresponding to all data points in D and E corresponding to all the directed edges among data points in V.

In the **graph mining phase**, a kNN mining query mines for all neighbors for the query object v_q in the defined graph G.

The templates for these pattern mining queries in streaming environments can be found in Figures 1, 2 and 3 in Section 1.

2.3. Sliding Window Semantics

We focus on periodic sliding window semantics as proposed by Continuous Query Language (CQL) [Arasu et al. 2006] and widely used in the literature [Yang et al. 2009a; Arasu and Widom 2004; Yang et al. 2009a]. Such semantics can be either time-based or count-based. For both cases, each query Q has a window size $Q.win$ (either a time interval or a tuple count) and a slide size $Q.slide$. The patterns will be generated only based on the data points falling into the window. The query window slides periodically either when a certain number of tuples arrives or a certain amount of time elapses. By sliding, a new window will be built to replace the old window, and thus again cover only the most recent portion of the stream at that moment. The templates of neighbor-based pattern mining queries using this query semantics have been shown in Figures 1, 2 and 3 in Section 1.

"Data Expiration" in Sliding Window Semantics. The notion of "data expiration" that we used in this work is defined by this sliding window query semantics, indicating that the data points are no longer included in the current query window. This means that the data points are no longer of interest to the analyst who specified the query. In such context , each data point carries a single time stamp indicating the time when this object was observed or arrived at the system. However, it does not carry validity information, which would indicate when it will expire. When a specific data point will "expire" (be no longer of the analyst's interest) is solely decided by the relationship between the data point's time stamp, current system time and the window size specified in the query. The specification of the window parameters are driven by the domain knowledge of the analysts and their specific analytical task.

Data-Driven Object Expiration. There is a notion of "expiration" that is different from sliding window semantics, namely data-driven expiration of tuples instead of query-semantics driven expiration of tuples, as we will explain below. Namely, in-

dependent from the sliding window semantics, the stream objects may have their own natural periods of "validity", or also called "lifetimes" [StreamInsight]. For example, each object in a stream reporting the sales items from online stores has a natural valid period. This is because a particular sale for each item may only be valid for a certain period of time, say a 50% off MacBook deal will only be available for the first hour of the Black Friday Sale, while the other 20% off sale items from Apple may be available for that whole day. This data-driven expiration semantics is not the target of this work.

2.4. Optimization for Multiple Queries

We support multiple neighbor-based pattern detection queries of the same query type that are specified on a common input stream but with arbitrary pattern and window parameters. We call all the queries submitted to the system together a Query Group QG, and each of them a Member Query of QG. All the queries within a same query group should have the same query type, be it density-based clustering, distance-based outlier detection or kNN queries. We use a common assumption that all the member queries are registered to and pre-analyzed by our system before the arrival of the input stream, indicating that all the member queries will be started simultaneously. Our goal is to minimize the overall CPU- and memory- resource consumption for executing all the member queries registered to our system.

2.5. Review of Existing Single Query Execution Strategy

Alternative methods for processing a single density-based clustering and distance-based outlier query over sliding windows are discussed in [Yang et al. 2009a]. Both analytical and experimental studies conducted in [Yang et al. 2009a] show that *Extra-N* and *Abstract-M* are the best existing approach for executing a single query of this type. They realize efficient evaluation by incrementally maintaining the cluster structures identified in the query window. Technically, they are based on a key idea, namely the notion of predicted views.

General Notion of Predicted Views. The key challenge that needs to be solved for incremental maintenance of neighbor-based patterns is to efficiently discount the effect of expired data points from the previously formed patterns. The expiration of existing data points may cause complex pattern structure changes, ranging from shrinkage, splitting to the termination of the patterns. Detecting and handling these changes caused by expirations, especially splitting, may require large amount of computation, which could be as expensive as recomputing the patterns from scratch.

To address this problem, *Extra-N* and *Abstract-M* exploit the general notion of predicted views. It is well known that, since sliding windows tend to partially overlap ($slide < win$), some of the data points falling into a window W_i will also participate in some of the windows right after W_i. Based on the data points in the current window, say a dataset D_{cur}, and the slide size, we can exactly "predict" the specific subset of D_{cur} that will participate in each of the future windows. We can thus pre-determine some properties of these future windows (referred as "predicted windows") based on these known-to-participate data points and thus form the "predicted views" for them. This general concept is widely used in the literature [Li et al. 2005; Yang et al. 2009a; Krishnamurthy et al. 2006] to process sliding window queries, and is called *Predicted View* technique. As an example, Figure 4 shows the predicted views for three future windows when detecting density-based clusters. The black, grey and white spots represent the core, edge and noise points identified in each predicted view. The lines among any two data points represent the neighborship between them. By using the predicted view technique, we can avoid the computational effort needed for discounting the effect of such expired data points from the detected clusters. The idea is to pre-generate the partial clusters for the future windows based on the data points that are in the current

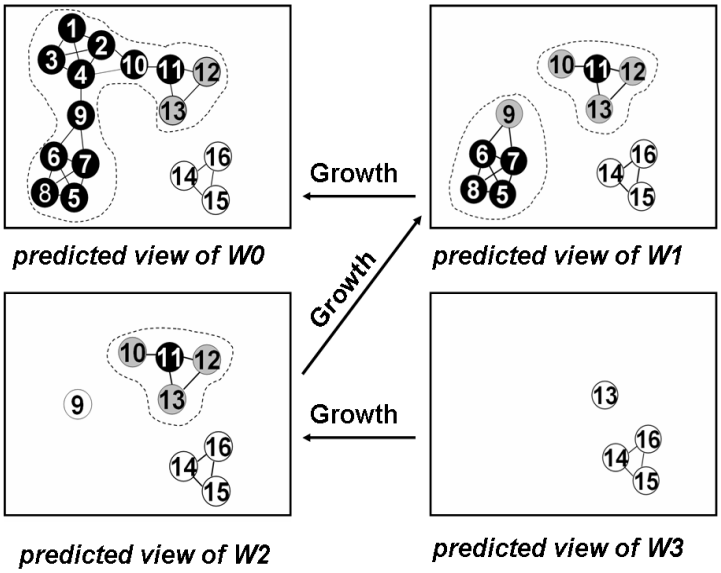

Fig. 4. Predicted views of four consecutive windows at W_0

window and known to participate in those future windows (without considering the to-be-expired ones). Then when the window slides, we can simply use the new data points to update the pre-generated patterns in the predicted views and form the up-to-date patterns in each window. Figures 4 and 5 respectively demonstrate examples of the "pre-generated" clusters in future windows and the updated clusters after the window slides.

Discussion. Generally, at each window slide, *Extra-N* and *Abstract-M* run one range query search for each new data point to update the progressive patterns, which are represented by the predicted views. As the best existing algorithms for single query execution, they achieve the minimum number of range query searches needed at each window. However, executing such single query algorithm for each member query independently is not a scalable solution for handling a QG with large $|QG|$. This is because the consumption of both computational and memory resources will increase linearly with the increase of $|QG|$. We thus need to design an optimized processing mechanism for multiple queries to handle a large query group against high speed data streams.

3. A PRELIMINARY SHARING EFFORT: SHARING RANGE QUERY SEARCHES

The basic strategy to share the computations among multiple neighbor-based pattern mining queries is to share the range query searches. Generally, to execute a query group QG with $|QG| = N$, we can execute N single query algorithms, each for a member query, independently (with each query maintaining its own progressive patterns independently), yet share the computations needed by the range query searches. Specifically, at each new query window, the single query algorithms require every new data point p_{new} to run a range query search to identify its neighbors, and communicate with them to update progressive patterns in the window (as discussed in Section 2.5). This means that, if executed independently, for a query group QG with $|QG| = N$, we need to run N range queries for each new data point p_{new}. However, by using range query

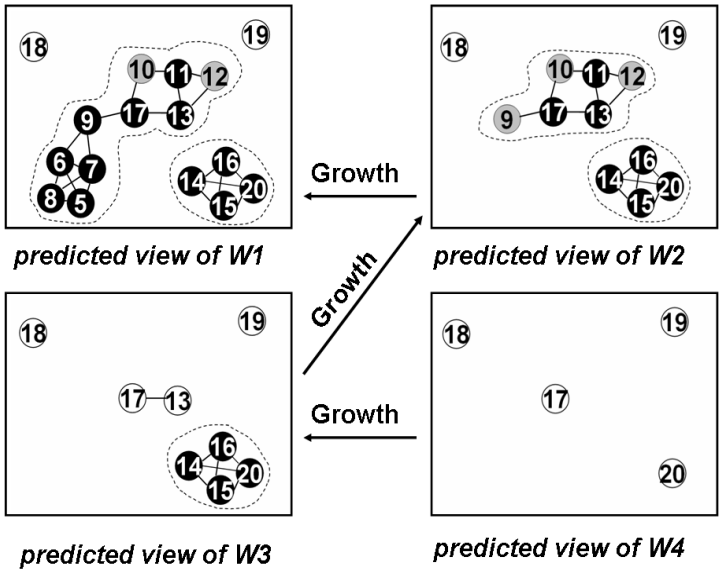

Fig. 5. Updated predicted views of four windows at W_1

search sharing, we could instead run just one range query search for each p_{new}, even if the queries in QG have different range thresholds θ^{range}.

In particular, we run the range query search for each p_{new} using $Q_i.\theta^{range}$, with $Q_i.\theta^{range}$ larger or equal to any $Q_j.\theta^{range}$ in QG. Using the result set of this "broadest" range query search, we then gradually filter out the results for the other queries with smaller and smaller θ^{range}. Clearly, for a given data point, the result set of a range query search using smaller θ^{range} is always a subset of that using a larger one. Also, since the range query search with largest θ^{range} is in any case needed for the particular query, no extra computation is introduced by this process. This general principle of sharing range query searches can be applied to any neighbor-based pattern mining requests that requires neighbor searches for data points, such as distance-based outlier detection. Sharing range query searches can be very beneficial for optimizing the system resource utilizations, especially when the window size is large.

3.1. Discussion

However, sharing range query searches alone is not sufficient for handling a heavy workload containing hundreds or even thousands of queries. Two critical problems still remain: 1) Since every member query still stores its progressive patterns independently, the memory space needed by executing a query group QG grows linearly with $|QG|$. 2) Because of the independent pattern storage, the pattern maintenance computation of different queries cannot be shared. To solve these two problems, we need to further analyze and exploit the commonalities among the member queries. Our goal is thus to design an integrated pattern maintenance mechanism that effectively shares both the storage and computational resources needed for multiple queries.

In our experimental studies shown in Figures 30, 31 and 32 in Section 7 we demonstrate how much performance gains that can be achieved by using this range query search sharing strategy alone. Also, in the same figures we compare such gains with

that can be achieved by our complete proposed solutions, which include more sophisticated sharing strategies introduced in the following Sections 4 and 5.

4. SHARING AMONG QUERIES WITH ARBITRARY PATTERN PARAMETERS

In this section, we discuss the shared processing of multiple queries with arbitrary pattern parameters. We first assume that the queries have the same window parameters, namely the same window size win and the same slide size $slide$. In such cases all the member queries will always detect patterns from exactly the same portion of the data streaming data (those fall into the current query window). This assumption will later be relaxed in Section 6 to allow completely arbitrary parameters.

To solve this problem, we analyze the relationships between the pattern sets identified by neighbor-based pattern mining queries with different pattern-specific parameter setting. In particular, we characterize the conditions under which one query is "more restricted" than the other, and discovery that a "containment" relationship holds between the pattern sets identified by the queries following such "strictness order". By exploiting this containment relationship, we incrementally organize the patterns identified by multiple queries into an integrated structure, and thus manage to maintain them in a shared manner. Such shared execution strategy leads to significant savings in both CPU time and memory utilization.

4.1. "Containment" among Neighbor-Based Pattern Sets

The definition of "containment" between neighbor-based pattern sets is generally more complex than the traditional "containment relationship" between the result sets of SPJ queries. In particular, such containment among neighbor-based pattern sets is not restricted to simple super- or sub-set relationships. Here we first use density-based clusters, which have one of most complicated pattern structures and complex containment relationships among neighbor-based pattern family, to explain this concept.

"Growth Property". We call the specification of such containment relationship among density-based cluster sets the "Growth Property". We now first define the "containment" between two density-based clusters.

Definition 4.1. Given two density-based clusters C_i and C_j (each cluster is a set of data points, which are called cluster members of this cluster), if for any data point $p \in C_i$, $p \in C_j$, we say that C_i is ***contained*** by C_j, denoted by $C_i \subset C_j$.

We now give the definition for the "growth property" between two density-based cluster sets.

Definition 4.2. Given two cluster sets Clu_Set1 and Clu_Set2 with for $i = 1, 2$, $Clu\_Set_i = \bigcup_{1 \le x \le n} C_x$, and for any $y \neq z$, $C_y \cap C_z = \emptyset$. If for any C_i in Clu_Set1, there exists exactly one C_j in Clu_Set2 that $C_i \subset C_j$, Clu_Set2 is defined to be a "**growth**" of Clu_Set1. We say the *growth property* holds between Clu_Set1 and Clu_Set2.

Beyond this definition, we now characterize all the possible interrelationships between the clusters belonging to Clu_Set1 and Clu_Set2.

Observation 4.1. *Given Clu_Set1 and Clu_Set2 with Clu_Set2 a **growth** of Clu_Set1, then any cluster C_j in Clu_Set2 must either be a **new** cluster (for any $p \in C_j$, $p \notin C_i$, if C_i is in Clu_Set1), an **expansion** of a single cluster in Clu_Set1 (there exists exactly one C_i in Clu_Set1 such that $C_i \subset C_j$), or a **merge** of multiple clusters in Clu_Set1 (there exist C_i, C_{i+1},...$C_{i+n} (n > 0)$ in Clu_Set1 with C_i, C_{i+1},...$C_{i+n} \subset C_j$.*

Figures 6 and 7 give an example of two cluster sets between which "growth property" holds.

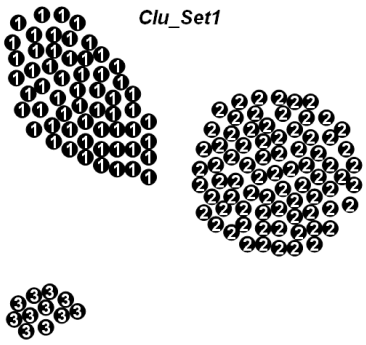

Fig. 6. Cluster Set 1 containing 3 clusters

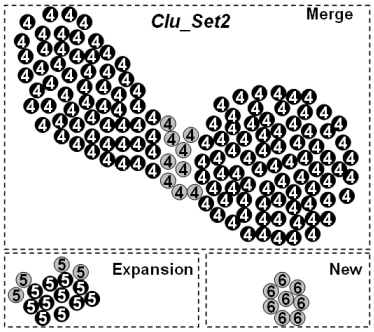

Fig. 7. Cluster Set 2 containing 3 clusters, which is a growth of Cluster Set1

The black spots in the figures represent the data points belonging to both cluster sets, while the gray ones represent those belonging to Clu_Set2 only. As depicted in the figures, the cluster C_4 in Clu_Set2 is a "merge" of clusters C_1 and C_2 in Clu_Set1, while the cluster C_5 and cluster C_6 in Clu_Set2 are an "expansion" of cluster C_2 in Clu_Set1 and a "new" cluster respectively. Generally, if Clu_Set2 is a "growth" of Clu_Set1, any two data points belonging to the same cluster in Clu_Set1 will also be members of the same cluster in Clu_Set2.

Hierarchical Pattern Representation. If the "growth property" transitively holds among a sequence of cluster sets, a hierarchical cluster structure can be built across the clusters in these cluster sets. The key idea is that, instead of storing cluster memberships for different cluster sets independently, we incrementally store the cluster "growth information" from one cluster set to another. Figures 8 and 9 respectively give examples of independent and hierarchical cluster membership structures built for the two cluster sets shown in Figures 6 and 7.

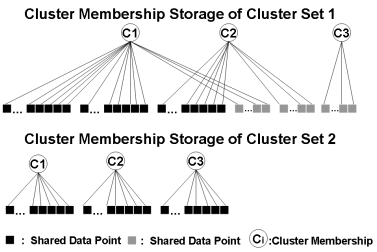

Fig. 8. Independent Cluster Membership Storage for Cluster Sets 1 and 2

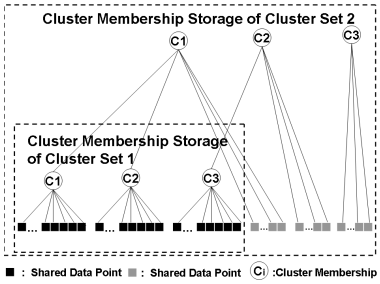

Fig. 9. Hierarchial Cluster Membership Storage for Cluster Sets 1 and 2

As shown in Figure 8, if we store the cluster memberships for cluster members in these two cluster sets independently, each cluster member (black squares) belonging to both clusters has to store two cluster memberships, one for each cluster set. However, if we store them in the hierarchical cluster membership structure as depicted in Figure 9, we no longer need to repeatedly store the cluster memberships for these "shared" cluster members. Instead, we simply store cluster memberships for each cluster member belonging to Clu_Set1, and then store the cluster "growth" information from Clu_Set1 to Clu_Set2. In particular, we just need to correlate each cluster C_i in

Clu_Set1 with a cluster in Clu_Set2 that contains it, and thereafter each cluster member can easily find its cluster membership in a specific cluster set by tracing to the corresponding level of the hierarchical cluster membership structure. Such "growth" information is now based on the **granularity of complete clusters** rather than the **granularity of individual cluster members**. Generally, for a sequence of cluster sets for which the "growth property" transitively holds, the hierarchical cluster structure can largely save the memory space needed for storing them.

Lemma 4.1**.** *Given a query group QG for which the growth property transitively holds among the cluster sets identified by all its member queries, the upper bound of the memory space needed for storing the cluster memberships using hierarchical cluster structure is $2 * N_{core}$ (independent from $|QG|$), with N_{core} the number of distinct data points that are at least once identified as core point in any member query of QG.*

Proof: The relationship between the number of cluster memberships stored and N_{core} is equal to the relationship between the total size of a binary heap and the number of leaf nodes of this heap. This is because a higher level cluster membership will only be stored if a merge of the cluster memberships happened at the lower level. ■

Besides the benefit of potentially huge memory savings, such hierarchical cluster structure can also help us to realize the integrated maintenance for multiple cluster sets identified by different queries, and thus save computational resources from maintaining them independently. In the later parts of this work, we will carefully discuss how this general principle can be used to benefit our multiple query optimization strategy.

Containment and Incremental Representation for Other Pattern Types. The containment relationship of other neighbor-based pattern types, such as distance-based outliers and top-k nearest neighbors are simpler than density-based clusters. In particular, as the outlier set or nearest neighbor set identified by a query is simply an object set, the containment relationship between any two outlier sets or nearest neighbor sets is simply the super- or sub-set relationship. Thus, the incremental representation of such pattern sets is simple as well. More precisely, we can store the smallest sets first and then incrementally store the extra objects for larger and larger sets.

4.2. Integrated Maintenance for Multiple Density-Based Clustering Queries

Now we discuss, for density-based clusters, in which cases such containment relationship holds and how it can help us to conduct shared execution for multiple queries. For a group of density-based clustering queries, they can vary on both pattern parameters, namely θ^{range} and θ^{cnt}. We first look at the cases in which the variations are only allowed on one parameter.

4.2.1. Arbitrary θ^{cnt}/θ^{range} Cases. In the first case, all queries have the same θ^{range} but arbitrary θ^{cnt}. Here, we make a straightforward observation.

Observation 4.2**.** *Given all queries in a query group having the same θ^{range}, the neighbors of each data point identified by these queries are the same.*

This observation indicates that for all our member queries, the neighborships identified in each specific window are exactly the same. However, this does not mean that the cluster structures identified by all queries are same and we can store them in the same way. This is because the different $\theta^{cnt}s$ of the member queries may assign different "roles" to a data point. For example, a data point with 4 neighbors is a "core point" for query Q_1 having $Q_1.\theta^{cnt} = 3$, while it is a "none-core point" for Q_2 having query $Q_2.\theta^{cnt} = 10$. As the hybrid neighborship abstraction (discussed earlier in Sec-

tion 2.5) requires each none-core point to store the links to its exact neighbors, while the core points store the cluster memberships only, a data point may need to store different types of neighborship abstractions depending on its roles identified by different queries.

To solve this problem, we turn to the "growth property" of density-based cluster structure discussed in Section 4.1.

Lemma 4.2. *Given two queries Q_i and Q_j specified on the same dataset, with $Q_i.\theta^{range} = Q_j.\theta^{range}$ and $Q_i.\theta^{cnt} \leq Q_j.\theta^{cnt}$, the cluster set identified by Q_i is a "growth" of the cluster set identified by Q_j (see **growth property** as defined in Definition 4.2).*

Proof: First, since $Q_i.\theta^{cnt} \leq Q_j.\theta^{cnt}$, the "core point" set identified by Q_j is a subset of that identified by Q_i. Second, since all the neighborships identified by Q_i and Q_j are exactly the same, all the "connections" in any cluster structure identified by Q_j will also hold for Q_i. This indicates that the cluster structure identified by Q_j will also be identified by Q_i (although it may be further expanded or merged). Finally, the "additional" core points identified by Q_i may only cause the birth of new clusters or expansion or merge of the clusters identified by Q_j, because they either extend these cluster structures when they are "connected" to one or more of them (causing expansion or union) or form new clusters by themselves when they are not "connected" to any (causing birth). This indicates that the cluster set identified by Q_i is a "growth" of that identified by Q_j (by Observation 4.1). This proves the lemma 4.2. ■

Figure 10 demonstrate an example of the cluster sets identified by three queries having the same θ^{range} but different $\theta^{cnt}s$.

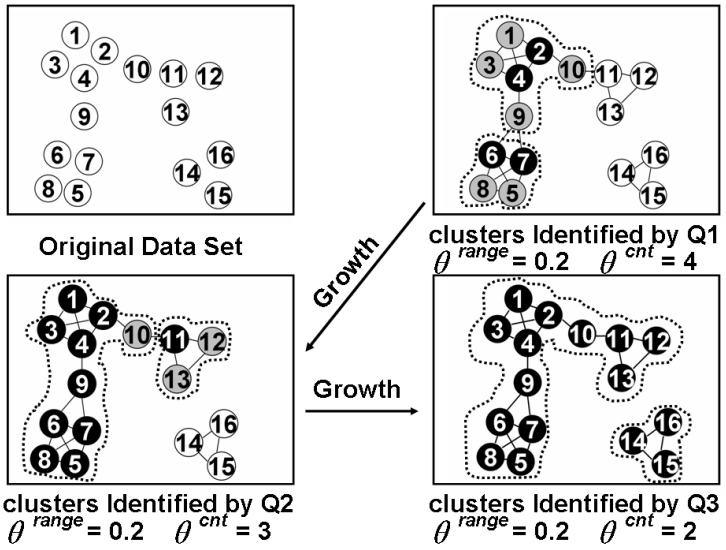

Fig. 10. Cluster sets identified by three different queries

Integrated Representation of Predicted Views across Multiple Queries with Arbitrary θ^{cnt}. As we discussed earlier in Section 4.1, once the "growth" property holds among the cluster sets, we can build the hierarchical cluster structure for them. We thus build an integrated hierarchical structure to represent multiple predicted views identified by different queries for the same corresponding predicted win-

dow. We refer to such Integrated Representation of Predicted Views across Queries with arbitrary θ^{cnt} by $IntView\_\theta^{cnt}$. For each predicted window, $IntView\_\theta^{cnt}$ starts from the predicted view with the most "restricted clusters". In this context, this corresponds to the predicted view maintained by Q_i with the largest θ^{cnt} among QG. Then, it incrementally stores the cluster "growth information", namely the "merge" of existing cluster memberships and the new cluster memberships, from one query to the next in the decreasing order of θ^{cnt}. Figure 11 gives an example of an $IntView\_\theta^{cnt}$, which represents the predicted views (shown in Figure 10) identified by three different queries.

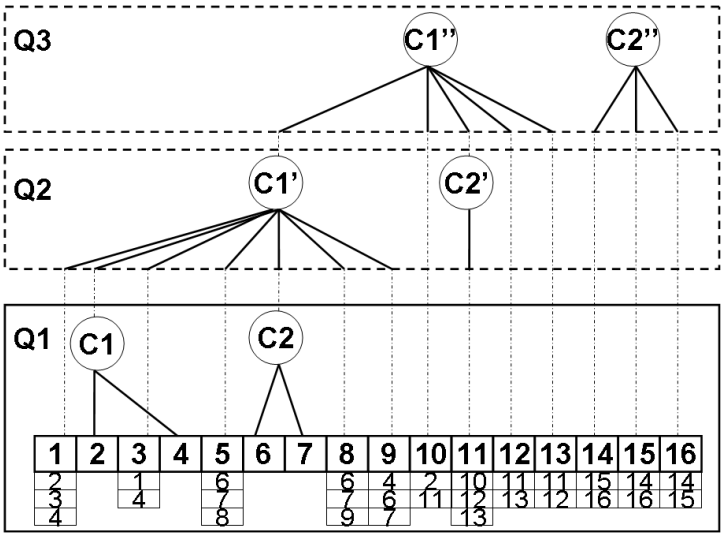

Fig. 11. $IntView\_\theta^{cnt}$: Integrated Representation for density-based clusters identified by three different queries

$IntView\_\theta^{cnt}$ successfully integrates the representations of multiple "predicted views" into a single structure, thus saving the memory space otherwise needed to store them independently.

Lemma 4.3**.** *Given the maximum window size allowed, the upper bound of the memory space needed by $IntView\_\theta^{cnt}$ is independent of $|QG|$, the cardinality of the query group.*

Proof: First, there are two types of meta-information that need to be stored by $IntView\_\theta^{range}$, namely the cluster memberships and the exact neighbors of the data points. Since $IntView\_\theta^{range}$ uses the hierarchical structure described in Section 4.1 to store the cluster memberships for the data points, the upper bound of the memory space used for storing cluster memberships is independent from $|QG|$ (as proven in Lemma 4.1). Second, $IntView\_\theta^{cnt}$ only stores the exact neighbors for "non-core" data points, and the maximum number of exact neighbors a "non-core" point can have is a constant (namely, $max(Q_i.\theta^{cnt}) - 1$). Thus, the upper bound of the memory space used for storing exact neighbors is again independent from $|QG|$. This proves Lemma 4.3. ■

Without using $IntView\_\theta^{cnt}$, the memory space needed for independently storing the cluster memberships identified by all member queries in QG will increase linearly with $|QG|$. Our method now makes it independent from $|QG|$ (as proven in Lemma 4.3).

Maintenance of $IntView\_\theta^{cnt}$. Besides the memory savings, we can also incrementally update multiple predicted views represented by a $IntView\_\theta^{cnt}$, thus saving computational resources. In particular, for each new data point p_{new}, we start the update process from the bottom level of $IntView\_\theta^{cnt}$, namely the predicted view identified by the query with largest θ^{cnt}. Then we incrementally propagate the effect of inserting this new data point to the next higher level of predicted views. Using the example utilized earlier in Figure 11, a new data point identified to have 3 neighbors in the window is a "none-core" in the bottom (most restricted) level predicted view, where $\theta^{cnt} = 4$. So, at the bottom level, we simply add all its neighbors to its neighbor list. However, its effect to upper level predicted views may differ, as this data point may be identified as a "core point" by a more "relaxed" query, say when $\theta^{cnt} = 3$. Then, we need to generate a cluster membership for it at that predicted view and merge it with those cluster memberships (if any) belonging to its neighbors.

The pseudo-code for the maintenance algorithm of $IntView\_\theta^{cnt}$ can be found in Figure 16, which is a special case of our final solution *Chandi*. In this special case, besides the exact same predicted windows built for all queries, the neighbor sets of a new data point identified by all queries are exactly same. We emphasize that the maintenance process is efficient for the following two reasons: 1) No extra range query search is needed when a data point is found to be a "core point" in an upper level predicted view and thus needs to communicate with its neighbors. This is because as a "none core point" in the lower level predicted views, it would already have stored the links to all its exact neighborships and thus would have direct access to them. 2) As the "growth" of cluster sets identified in predicted views is incremental, less and less maintenance effort will be needed as we handle the higher level predicted views.

We also found that the "growth property" holds between two queries with the **same θ^{cnt} but different θ^{range}s**.

Lemma 4.4. *Given two queries Q_i and Q_j specified on the same data set with $Q_i.\theta^{cnt} = Q_j.\theta^{cnt}$ and $Q_i.\theta^{range} \geq Q_j.\theta^{range}$, the cluster set identified by Q_i is a "growth" of that identified by Q_j.*

The shared execution strategy for this case is very similar to the previous case in which all queries have the same θ^{range}s. Thus, the detailed discussion for this case is omitted here.

4.2.2. Arbitrary θ^{range}, Arbitrary θ^{cnt} Case. Now we discuss the shared processing for a query group QG with queries having totally arbitrary pattern parameters, namely arbitrary θ^{range} and arbitrary θ^{cnt} values. Although the "growth property" holds between the cluster sets identified by two queries Q_i and Q_j, if Q_i and Q_j share at least one query parameter, it does not necessarily hold if both query parameters of Q_i and Q_j differ. To again take advantage of the compact structure of the Integrated Representation of Predicted Views, we need to explore when the "growth property" holds between two queries in the most general cases.

Lemma 4.5. *Given two queries Q_i and Q_j specified on the same dataset, with $Q_i.\theta^{cnt} \leq Q_j.\theta^{cnt}$ and $Q_i.\theta^{range} \geq Q_j.\theta^{range}$, the cluster set identified by Q_i is a "growth" of that identified by Q_j.*

Proof: Lemma 4.5 can be proven by the transitivity of the "growth property". Given a query Q_k with $Q_i.\theta^{cnt} \leq Q_k.\theta^{cnt} \leq Q_j.\theta^{cnt}$ and $Q_k.\theta^{range} = Q_j.\theta^{range}$, the cluster set identified by Q_k is a "growth" of that identified by Q_j (by Lemma 4.2). This means that for any cluster C_a identified by Q_j there exists a cluster C_b identified by Q_j such that $C_a \subseteq C_b$. Also, the cluster set identified by Q_i is a "growth" of that identified by Q_k (by Lemma 4.4). This means that for any cluster C_b identified by Q_j there exists

a cluster C_c identified by Q_i that $C_b \subseteq C_c$. So for any cluster C_a identified by Q_j there exist a cluster C_c identified by Q_i such that $C_a \subseteq C_c$. Thus, the cluster set identified by Q_i is a "growth" of that identified by Q_j (by Definition 4.2). ■

To more intuitively describe the relationship between any two queries in a query group, we give the following definition.

Definition 4.3. Given two queries Q_i and Q_j specified on the same dataset, if $Q_i.\theta^{cnt} \leq Q_j.\theta^{cnt}$ and $Q_i.\theta^{range} \geq Q_j.\theta^{range}$. we say Q_j is a "more restricted" query than Q_i, and Q_i is a "more relaxed" query than Q_j.

Integrated Representation of Predicted Views across Multiple Queries with Arbitrary Pattern Parameters. We aim to build a single structure which represents the "predicted views" identified by all member queries of QG in the same window. However, given the "growth property" only holds between two queries if one is more restricted than the other, we can no longer expect to put all member queries into a single hierarchy.

Our solution is to build a "**Predicted View Tree**", which integrates multiple linear predicted view hierarchies into a single tree structure. In this tree structure, each predicted view (except the root) only needs to store and maintain the incremental information (cluster "growth") from its parent, much like the predicted views in $IntView\_\theta^{range}$ and $IntView\_\theta^{cnt}$. In particular, such a "Predicted View Tree" starts from the predicted view that represents "the most restricted query" among QG. "The most restricted query" here corresponds to the member query that has both the smallest θ^{cnt} and the largest θ^{range} among QG. If such a "most restricted query" does not naturally exist in QG, we build a "virtual" one by generating a query with the smallest θ^{cnt} and the largest θ^{range} among QG. The predicted view representing this "most restricted query" will be the "root" of our "Predicted View Tree". If the most restricted query is a virtual query, its predicted view will be used for "Predicted View Tree" maintenance but it will never generate any output. Then the predicted views representing more relaxed queries will be iteratively put on the next higher level (farther from the root) of the tree. More specifically, after picking "the most restricted query" as the root of the tree, we iteratively pick (and remove) "the most restricted queries" remaining in QG and put their predicted views as the next level of the tree. Here, a member query Q_j is one of "the most restricted queries" remaining in QG, if there does not exist any other member query Q_i in QG, which is "more restricted" than Q_j.

For example, given $QG = \{Q_1(\theta^{range} = 0.5, \theta^{cnt} = 5), Q_2(\theta^{range} = 0.4, \theta^{cnt} = 7), Q_3(\theta^{range} = 0.2, \theta^{cnt} = 10), Q_4(\theta^{range} = 0.3, \theta^{cnt} = 7), Q_5(\theta^{range} = 0.4, \theta^{cnt} = 8)\}$. The root of the "Predicted View Tree" is the predicted view representing "the most restricted query", namely Q_3 in this case. Then, the second level "most restricted queries" are Q_4 and Q_5, which are more relaxed than Q_3 but more restricted than Q_1 and Q_2. Finally, the third level "most restricted queries" are Q_1 and Q_2. This process of figuring out "the most restricted queries" at each level is equal to the problem of iteratively calculating the "skyline" [Yuan et al. 2005; Zhang et al. 2009; Soliman et al. 2007] in the two dimensional space of θ^{range} and θ^{cnt}. Since this process of building "Predicted View Tree" can be conducted offline during query compilation, any existing skyline algorithm can be plugged into our system to solve this problem.

The predicted views on the lower level of the tree always represent more restricted queries than those on the higher levels. Then, the "growth information", namely the evolution of cluster memberships and the "additional exact neighbors", will be stored from one predicted view to each of its "children" on the higher level. Such building process guarantees an important property of "Predicted View Tree" as described in the following lemma.

Lemma 4.6. *Given a cluster set Clu_Set_m identified by a query Q_m on the i^{th} level of the "Predicted View Tree", and a cluster set Clu_Set_n identified by a query Q_n on the $(i-1)^{th}$ level, the "growth information" between Clu_Set_m and Clu_Set_n is no more than that between Clu_Set_m and any cluster set Clu_Set_o identified by a query Q_o on the $(i-j)^{th}(i>j>1)$ level.*

Proof: Since the queries on the $(i-j)^{th}$ level are always more restricted than those on the $(i)^{th}$ level, we know that Clu_Set_n is a growth of Clu_Set_o, Clu_Set_m is a growth of Clu_Set_n and Clu_Set_m is also a growth of Clu_Set_o. This means the "growth information" from Clu_Set_o to Clu_Set_m can actually be divided into two parts, namely the "growth information" from Clu_Set_o to Clu_Set_n and that from Clu_Set_n to Clu_Set_m. This proves that the "growth information" from Clu_Set_n to Clu_Set_m is no more than that from Clu_Set_o to Clu_Set_m. ■

This property assures that each predicted view in the "Predicted View Tree" maintains the smallest increments and represent multiple predicted views as compact as possible.

To finalize the tree structure, for each query Q_n on the i^{th} level of the tree, we need to determine its "parent" on the $(i-1)^{th}$ level. We aim to find such a "parent" query Q_m that is most similar to Q_n, indicating that there exists the least "growth information" from the cluster set identified by itself to that identified by Q_n. Since the queries with similar $\theta^{range}s$ tend to identify similar neighborships in the window, this indicates that the difference on $\theta^{range}s$ has a larger influence on cluster changes compared with $\theta^{cnt}s$. So, when we determine the parent predicted view, although we consider the similarity between both pattern parameters, more "weight" is given to that between $\theta^{range}s$. To unify the names of the hierarchical structures representing multiple predicted views, we henceforth call this the "Predicted View Tree" $IntView\_\theta$.

Although $IntView\_\theta$ is a tree structure, instead of a linear sequence like $IntView\_\theta^{cnt}$ and $IntView\_\theta^{range}$, they share the core essence that each predicted view is incrementally built based on the predicted view most similar with it, and the "growth property" holds between them. We call the member queries on each path of $IntView\_\theta$ a group of ***shared queries***.

Lemma 4.7. *The upper bound of the memory space needed by $IntView\_\theta$ for any group of shared queries is independent from the number of queries in this group.*

Since all these queries are on the same path of $IntView\_\theta$ structure, indicating that the growth property transitively holds among the cluster set identified by them. The independency between the upper bound of the memory space and the number of queries can be proven using the same method as we used for proving Lemma 4.3.

The maintenance process of $IntView\_\theta$ is also similar with that for $IntView\_\theta^{cnt}$ and $IntView\_\theta^{range}$. For each new data point, we always start the maintenance from the root of the $IntView\_\theta$, namely the predicted view representing the most restricted query. Then we incrementally maintain the predicted views on the next higher level of $IntView\_\theta$. Again, this maintenance process is a special case of our final uniform solution *Chandi* (pseudo code shown in Figure 16). In this special case, the predicted windows built for all queries are exactly same.

Now we conclude with the contribution of $IntView\_\theta$.

Theorem 4.3. *For a given density-based clustering query group QG with member queries having arbitrary pattern parameters, $IntView\_\theta$ achieves full sharing of both memory space and query computation.*

Proof: First, the storage mechanism of $IntView\_\theta$ is completely incremental. In particular, since each predicted view in $IntView\_\theta$ only stores the increments from

its "parent", no duplicate information is ever stored among any two predicted views. This proves that $IntView\_\theta$ achieves full sharing of memory space. Second, since the maintenance process of $IntView\_\theta$ is incremental as well, indicating that each new data point only communicates with each of its neighbors once on each path of the tree structure, no matter how many different predicted views their neighborship appears in. This proves that $IntView\_\theta$ achieves full sharing of the computation of multiple queries. ■

4.3. Integrated Maintenance for Multiple Distance-Based Outlier Detection Queries

For distance-based outlier detection over sliding windows, the solution for processing a single request is discussed in the literature [Angiulli and Fassetti 2007; Yang et al. 2009a]. Using the most up-to-date technique, namely the Abstract-C algorithm [Yang et al. 2009a], the meta-information that needs to maintained to update the outliers is simple. In particular, for each data point in the window, its neighbor count will be sufficient to determine whether it is an outlier or not. Thus, for an individual query, the meta-information it maintains for each predicted view is the potential outlier set in the corresponding predicted window, namely the data points that are known to have less than $win * \theta^{fra}$ neighbors in that window, if no new data points were to join its neighborhood. Also, a predicted neighbor count will be maintained for each potential outlier in each predicted window. Such neighbor counts will be updated when the new data points come in. This then helps us to decide whether the data points should still be kept in the potential outlier set. More specifically, each new data point updates the predicted neighbor count of its own and of all its neighbors in each future window. The data points with too many neighbors (neighbor count larger or equal than $win * \theta^{fra}$) will be removed from the potential outlier set of a particular window, since they can no longer be identified as outliers.

Figure 12 shows the predicted views built for two queries Q_1 and Q_2 on a 2D dataset. That is $Q_1.\theta^{range} = 0.1, \theta^{fra} = 25\%, win = 8, slide = 2$ (for Q_1, a data point with 2 or less neighbors will be identified as an outlier), and Q_2 with $Q_2.\theta^{range} = 0.15, \theta^{fra} = 15\%, win = 8, slide = 2$ (for Q_2, a data point with 1 or less neighbors will be identified as an outlier) . Figure 13 (top 2 lines) shows the corresponding meta-information Abstract-C maintains for these two queries independently. More details of Abstract-C can be found in [Yang et al. 2009a].

By analyzing the semantics of distance-based outliers, we observe that the containment relationship holds between the outlier sets identified by different queries under certain conditions. First, let us fix the distance threshold θ^{range}, while varying the fraction threshold θ^{fra} among the member queries in a query group QG. We then note that the outliers identified by a query Q_i with the largest θ^{fra} among QG will always cover (be a superset of) all the outliers that are identified by other member queries. For any two queries Q_i and Q_j, if $Q_i.\theta^{range} \leq Q_j.\theta^{range}$, the outlier set identified by Q_j is a subset of that identified by Q_i. In other words, the identified outlier set "grows" monotonically as θ^{fra} increases. So, in this case, for each predicted window, we can store and maintain the potential outlier sets identified by different queries incrementally for all member queries. As shown in Figure 13, we can just store a single copy of the largest potential outlier set and use a flag (depicted with different levels of darkness) to distinguish by which queries each outlier is identified. Here a single integer flag will be sufficient to distinguish among all the possible combinations, because the "strictness" of the queries are exactly ranked. In other words, a data point can only be identified as outlier by Q_i, if it can be identified by all queries that have $Q_j.\theta^{fra} > Q_i.\theta^{fra}$. Also, since the neighbor count of each data point identified by all queries will always be same, we can simply maintain a single predicted neighbor count for each data point

and use it to answer all queries. Thus, in this case both the storage and computation of the meta-information maintained by all queries are fully shared.

Second, if we fix the fraction threshold θ^{fra}, while varying the distance threshold θ^{range} among a query group QG, the outliers identified by a query Q_i with smallest θ^{range} among QG will always cover all the outliers that should be identified by the other member queries. For any two queries Q_i and Q_j, if $Q_i.\theta^{range} \geq Q_j.\theta^{range}$, then the outlier set identified by Q_j is a subset of that identified by Q_i. In other words, the identified outlier set "grows" monotonically as θ^{fra} decreases. This case is very similar to the previous case just discussed in the last paragraph. Thus, the same incremental maintenance mechanism of potential outlier sets is also applicable here. The only difference between this case and the previous case is now the neighbors identified by different queries for the same data point may be different. However, as the neighbor counts identified for any data point monotonically increase with the θ^{range} parameter, we simply maintain the increments on the neighbor counts for each query, if there are any. See data point 4 in Figure 13 for an example.

Finally, we allow arbitrary settings on both distance and fraction thresholds. In this more general case, we can observe that the outlier set identified by a "stricter" query Q_i is a subset of that identified by a more "relaxed" query Q_j, if $Q_i.\theta^{range} \leq Q_j.\theta^{range}$ and $Q_i.\theta^{fra} \leq Q_j.\theta^{fra}$. Thus we can group the member queries into several non-overlapping subgroups to ensure that such containment relationship transitively holds among the queries in each of the subgroups. Then we can use the same techniques discussed in the previous cases to maintain the outliers integrally for all queries in each subgroup.

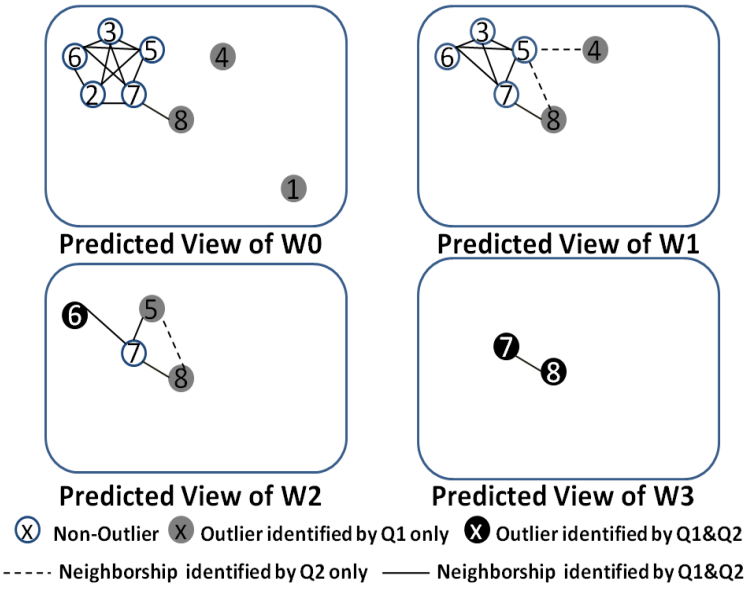

Fig. 12. Distance-Based Outliers Identified by Q_1 and Q_2

4.4. Integrated Maintenance for Multiple kNN Queries

This pattern type takes only one pattern-specific parameter, namely the input k. Thus, a group of such queries with different input k settings are all querying the nearest

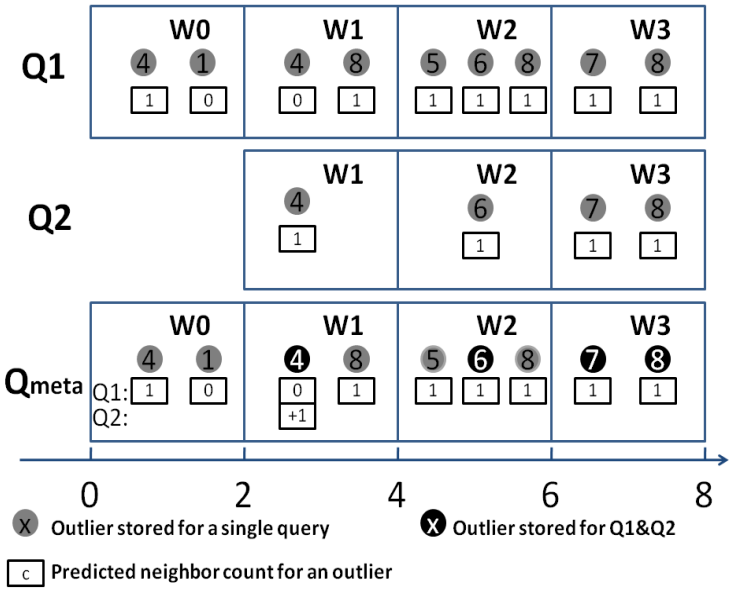

Fig. 13. Independent vs. Integrated Representation for Distance-Based Outliers Identified by Q_1 and Q_2

neighbors of the same given object but are asking for different numbers of such nearest neighbors. In this case, using the incremental pattern maintenance mechanism is quite straightforward. Basically, in any predicted window, the k nearest neighbors (kNN) identified by a query Q_i with the largest k among the query group will cover all the kNN that should be identified by other queries. For any two queries Q_i and Q_j, if $Q_i.k \geq Q_j.k$, the kNN identified by Q_j is a subset of that identified by Q_i. So, we can again incrementally store and maintain the k nearest neighbors identified by different queries in an integrated manner.

In particular, in any predicted window, for a group of kNN queries, rather than maintaining one kNN set for each query, we only maintain a single "KNN set" of the query object, namely its K Nearest Neighbors, with K the largest k setting among all queries in the query group. This single KNN set will represent the kNN of all queries in the query group. This KNN set can be simply implemented as a sorted list based on their distance to the query object.

When a new data point p_{new} comes into the system, we only compare the distance between p_{new} and the query object with the distance between the query object and its Kth nearest neighbor in KNN. If the p_{new} is closer to the query object compared to its Kth nearest neighbor, it qualifies for the KNN set, indicating that it will be in kNN set for at least one query in the group. Otherwise p_{new} is discarded for this predicted window, as it has no chance to make kNN for any query in this predicted window.

If p_{new} qualified the KNN set, we use p_{new} to update the KNN set, We only need two operations to finish the update. First, we put p_{new} into the KNN set and then remove the previous Kth nearest neighbor (the farthest one) from KNN. During the output, we simply scan the kNN set from the 1st to kth nearest neighbors. The nearest ones will be reported as kNN for all queries, while the farther ones will be reported to less and less queries depending on the k settings of the particular queries. Clearly, the cost of our integrated maintenance strategy for multiple kNN queries is almost equal to the cost of executing the single kNN query with the largest k setting.

5. SHARED EXECUTION FOR QUERIES WITH DIFFERENT WINDOW PARAMETERS

In this section, we discuss memory and CPU sharing strategy among multiple neighbor-based pattern queries with different window parameters, namely variations in the window size win and the slide size $slide$. During this discussion, we assume that all these queries have the same pattern parameters. The techniques proposed in this section are general, and can be equally applied to all the neighbor-based pattern mining queries discussed in this work and even other query types, such as graph mining. This is because the optimizations introduced here are at the window level, namely, regarding to planning and organization of predicted windows but independent from the specific pattern maintenance within each window. To explain the proposed ideas in details, we pick density-based clusters as the specific pattern type in our running examples.

5.1. Same win, Arbitrary $slide$ Case.

In this case, all member queries have the same window size win, while their slide sizes may vary. First, we assume that all queries start simultaneously. The equality of window sizes implies that all queries always query on the same portion of the data stream. More specifically, at any given time the data points falling into the windows of different queries are the same. Then, the only difference among different queries is that they need to generate output at different moments, as they have different slide sizes. For example, given three queries Q_1, Q_2 and Q_3, with $Q_1.win = Q_2.win = Q_3.win = 10(s)$, $Q_1.slide = 2(s)$, $Q_2.slide = 3(s)$ and $Q_3.slide = 6(s)$, the query windows of them cover exactly the same portion of the data stream at any given time, while they are required to output the clusters at every 2, 3 and 6 seconds respectively. So, to serve the different output time points, they need to build predicted windows starting at different times, each serving a future output time point. In this example, assuming all three queries start at wall clock time 00:00:00, they all need to build a predicted window starting at 00:00:00 for generating the output at 00:00:10, which is their first and shared output time point. Then Q_1 needs to build predicted windows starting at 00:00:02, 00:00:04, etc, to serve the output time points at 00:00:12, 00:00:14, while Q_2 and Q_3 need to build predicted windows starting at 00:00:03, 00:00:06, etc, and 00:00:06, 00:00:012, etc respectively.

To solve this problem, for a given group QG, we build a single meta query Q_{meta} which integrates all the member queries of QG. In particular, this meta query Q_{meta} has the same window size with all member queries in QG, while its slide size is no longer fixed but adaptive during the execution. More specifically, the slide size of Q_{meta} at a particular moment is decided by the nearest moment which at least one member query of QG needs to be answered. The specific formula to determine the next output moment is:

$$T_{nextoutput} = Min(\lceil \frac{T - win}{Q_i.slide} \rceil + 1) * Q_i.slide + win)$$

With T the current wall-clock time and win the common window size among all queries. Using the earlier example, for the query group having three member queries, we build a meta query Q_{meta} for it with $win = 10s$. So, at wall-clock time 00:00:10, the slide size of Q_{meta} should be 2s, as 00:00:12 will be the nearest time at which a member query (Q_1) needs to be answered. Then its slide size is adapted to 1s, 1s and 2s at 00:00:02, 00:00:03 and 00:00:04 respectively for the same reason.

Such adaptive slide size strategy is compatible with the "view prediction" technique. This is because, although the slide size of Q_{meta} may keep changing, these changes are still predictable and periodic. In particular, given the slide size of all the member queries, we always know at which moments which member queries need to be an-

swered. The interval between any two successive output moments is actually changing periodically. So, we can construct an output schedule (with a lookahead of finite number of output time points) for Q_{meta}, which predetermines the slide size of Q_{meta} at any given moment.

Knowing the slide sizes of Q_{meta}, we can just build predicted windows for Q_{meta} based on the output time points. Still using the earlier example, at wall-clock time 00:00:10, we would have built eight "predicted windows" for Q_{meta} , which start from 00:00:00, 00:00:02, 00:00:03, 00:00:04, 00:00:06, 00:00:08, 00:00:09 and 00:00:10 respectively, as each of them corresponds to an output time point for at least one member query. Among these eight "predicted windows", many of them are actually serving multiple queries. For example, the "predicted windows" starting at 00:00:00 and 00:00:06 will be used to answer Q_1, Q_2 and Q_3 as they correspond to the output time points that are shared by all three queries. This also means that if we were to maintain the predicted windows for these queries independently, four more predicted windows would need to be maintained at this given moment. In particular, Q_2 and Q_3 would have needed to maintain their own predicted windows starting at 00:00:00 and 00:00:06 separately, although they are exactly the same as those maintained by Q_1. In this example, 33 percent of the "predicted windows" are saved from the independent maintenance mechanism. This means that 33 percent of storage space and computational resources are saved in this case.

In conclusion, by building a meta query representing all member queries in a query group, we can save both the memory space and CPU processing time for answering the query group for the following reasons: 1) No overhead, in particular, no extra predicted views will be introduced, as a predicted window is built only if at least one member query needs output at that moment. In other words, all the predicted windows built in our integrated solution need to be maintained by individual member queries anyways. 2) Many predicted views can be shared as several member queries may require output at the same time. The specific amount of sharing depends on the percentage of overlaps of member queries' output time points.

5.2. Same $slide$, Arbitrary win Case

In this case, although the window size may vary among the member queries, we hold the slide size steady, indicating that their output schedules are identical. Here we first work with a common assumption that all the window sizes of the member queries are multiples of their common slide size. We observed that, given a query group with member queries having the same slide size but different window sizes, all the member queries require output at exactly the same moments. Based on this observation an important characteristic can be discovered for such query groups.

Lemma 5.1**.** *Given a query group QG with member queries having the same slide size $slide$ but arbitrary window sizes (multiples of $slide$), the "predicted windows" maintained for Q_i, with $Q_i.win$ larger or equal to any other $Q_j.win$ in QG, will be sufficient to answer all member queries in QG.*

Proof: This is because the "predicted windows" maintained for Q_i will cover all the "predicted windows" that need to be maintained for all the other queries. More specifically, at any given moment, say wall-clock time T, the "predicted windows" that need to be maintained for a member query Q_n include all those starting at $T - n * slide$ ($1 \leq n \leq \frac{Q_n.win}{slide}$). As $Q_i.win$ is larger or equal than any $Q_j.win$, the "predicted windows" maintained for Q_i cover all those needed by other queries. At time T, any member query Q_j can be answered by the "predicted window" starting from $T - Q_j.win$. ■

For example, given three queries Q_1, Q_2 and Q_3, with $Q_1.slide = Q_2.slide = Q_3.slide = 5s$, $Q_1.win = 10$, $Q_2.slide = 15s$ and $Q_3.slide = 20s$, at wall clock time 00:00:20, the "predicted windows" built by Q_3 start from 00:00:00, 00:00:05, 00:00:10 and 00:00:15 respectively, while those need to be maintained by Q_1 and Q_2 start from 00:00:10, 00:00:15 and 00:00:05, 00:00:10, 00:00:15 respectively. The later all overlap with those built by Q_3. At this moment, the "predicted window" starting from 00:00:00 can be used to answer Q_3, while the predicted windows starting from 00:00:10 and 00:00:05 can be used to answer Q_1 and Q_2 respectively.

In summary, we only need to maintain the predicted windows for a single member query, namely the query with the largest window size, and then can answer all the member queries in the query group with different predicted windows it maintains. Clearly, full sharing is achieved. Here, we also note that although we made the common assumption in Lemma 5.1 that the window sizes are multiples of $slide$, to make the problem easier to understand, it is not crucial for our solution. Our solution can easily be relaxed to handle the cases where window sizes of member queries are completely arbitrary.

5.3. Arbitrary $slide$, Arbitrary win Case

We now give the solution for the more general case that both window parameters, namely win and $slide$, are arbitrary. Generally, the solution for this case is a straightforward combination of the two techniques introduced in the last two subsections. In particular, we simply build one single meta query that has the largest window size among all the member queries and uses an adaptive slide size. These two techniques are fully compatible, because they were both designed to make sure correct predicted windows (start and end as required by query semantics) are created to answer the member queries.

Here we use an example to demonstrate our solution. Given three queries Q_1, Q_2 and Q_3, with $Q_1.win = 10$, $Q_1.slide = 4$, $Q_2.win = 9$, $Q_2.slide = 5$, $Q_3.win = 6$ and $Q_3.slide = 2$, and all starting at wall clock time 00:00:00, we build a meta query Q_{meta} with $Q_{meta}.win = max(Q_i.win)_{(1 \leq i \leq 3)} = 10$. Then we adaptively change its slide size based on the next nearest output time point required by (at least) one of these three queries. For instance, at wall clock time 00:00:10, six predicted windows would have been built, which start from 00:00:00 (serving Q_3 for output at 00:00:10), 00:00:01 (serving Q_2 for output at 00:00:10), 00:00:04 (serving Q_1 for output at 00:00:12 and Q_3 for output at 00:00:10), 00:00:06 (serving Q_2 for output at 00:00:13 and Q_3 for output at 00:00:12), 00:00:08 (serving Q_1 for output at 00:00:18 and Q_3 for output at 00:00:14) respectively. Figure 14 shows the predicted views that need to be maintained by each of these three queries independently, versus those would instead be maintained by the meta query at wall clock time 00:00:10.

6. PUTTING IT ALL TOGETHER: THE GENERAL CASE

Finally, we now discuss the case that the pattern and window parameters are both arbitrary for the queries in a query group. Although sharing among a group of totally arbitrary queries is a hard problem if we had to solve it from scratch, we now can easily handle it by combining the two techniques introduced in last two sections, namely the incremental pattern representation technique and the meta query technique. These two techniques are orthogonal to each other, and can thus be easily combined. In particular, the integrated pattern representation technique (introduced in Section 4) is designed to share among a group of queries that are specified on the same dataset, which in our case is each predicted window. So, we can consider this here as an "**intra-predicted-windows**" sharing technique. On the other hand, the meta query technique (introduced in Section 5) is designed to make sure that the predicted windows,

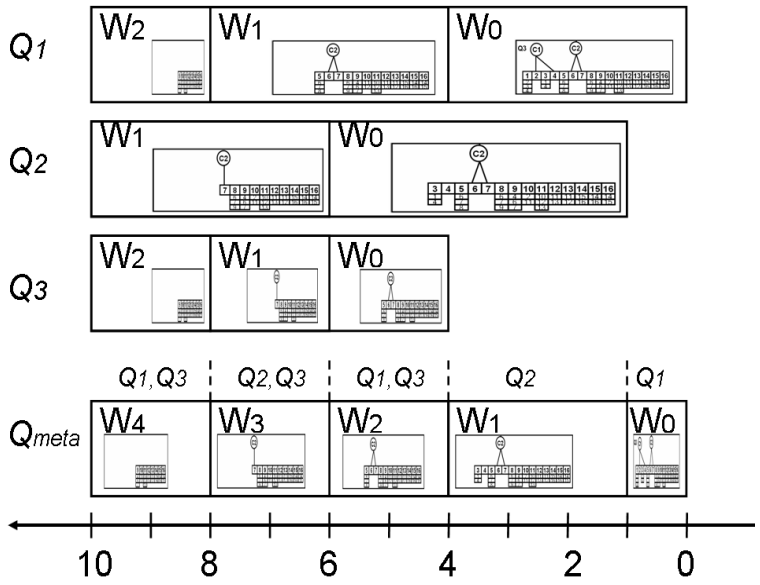

Fig. 14. Predicted Views Maintained By Three Queries Q1, Q2 and Q3 Independently versus Those Maintained By a Single Meta Query

which needs to be maintained by different queries, start and end properly and share across the different predicted windows. So, it is an "**inter-predicted-windows**" sharing technique. Thus, these two orthogonal techniques can be easily applied together to realize the full potential of sharing of the member queries on both the inner- and inter-predicted window level.

Here we use an example to demonstrate such a combination. Given three queries Q_1, Q_2 and Q_3 starting at 00:00:00, with $Q_1(win = 10, slide = 4, \theta^{range} = 0.2, \theta^{cnt} = 5)$; $Q_2(win = 9, slide = 5, \theta^{range} = 0.3, \theta^{cnt} = 4)$ and $Q_3(win = 6, slide = 2, \theta^{range} = 0.2, \theta^{cnt} = 3)$, we first use the meta query technique to build the predicted windows they need to maintain. At wall clock time 00:00:10, the required predicted windows are the same as those shown in Figure 14. Then, for each predicted window built, we apply the $IntView\_\theta$ technique to build an "Predicted View Tree" to integrate the predicted views (of different queries) in this window. For the predicted window starting from 00:00:04, which is serving Q_1 and Q_3, we build a "Predicted View Tree" representing both Q_1 and Q_3. Now the "Predicted View Tree" structures built for different windows may no longer be all the same as those in the example we demonstrated in Figure 15. This is because the predicted view of a particular query will appear on a "Predicted View Trees" only if this predicted window needs to be maintained by this query, indicating this predicted window corresponds to an output time point for it. Using the same example, Q_2 has no predicted view in W_4, as W_4 is not a predicted window that need to be maintained by it.

We call this ultimate hierarchical structure $IntView$. Figure 15 depicts the final $IntView$ built for the three queries mentioned in the earlier example.

In particular, $IntView$ is a tree structure that starts from the predicted view acting as the root (r_{newest}) of the "Predicted View Tree" in the newest predicted window (with the largest window number). Thus, each root predicted view in an older predicted window is now incrementally built based on that in the next window. This indicates that, as subtrees for $IntView$, each "Predicted View Trees" in an older window is now

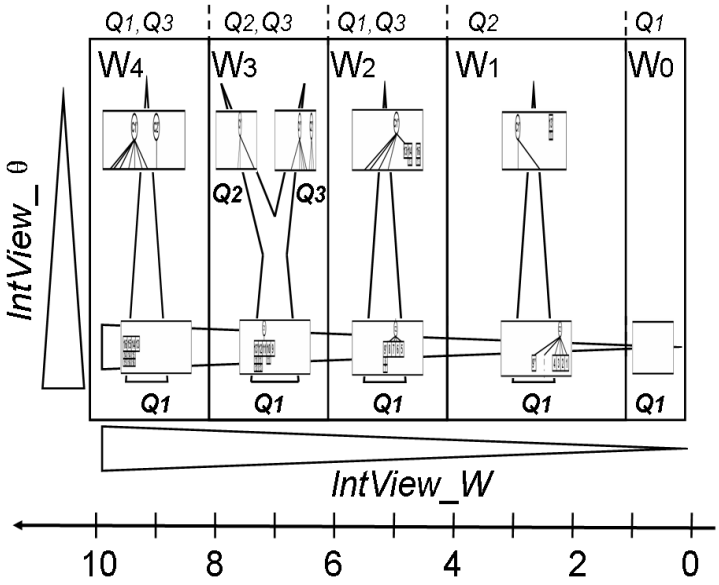

Fig. 15. $IntView$: Integrated Representation for Predicted Views Identified by 3 Queries in 5 Predicted Windows

built based on the incremental information from the next (the newer) window (as its root itself now is incremental). We call the final solutions for density-based clustering, distance-based outlier detection and kNN queries *Chandi*, "*SDOD* and *SkNN* respectively. We give the pseudo-code of them in Figures 16, 17 and 18.

As shown in our pseudo-code in Figures 16 and 17, when a new data point arrives at the system, the *Chandi* and *SDOD* algorithms, which handles density-based clusters and distance-based outliers respectively, first run a range query search using the largest θ^{range} among the query group to collect all its potential neighbors. Then it distributes each of them to the first predicted view on each path of $IntView$, in which their "neighborship" truely exists. Then, it starts the $IntView$ maintenance process from the root of $IntView$, namely the root predicted view of the newest predicted window in $IntView$, and then incrementally maintains those at higher levels of $IntView$. During the maintenance of each predicted view, it only needs to communicate with the neighbors assigned to that particular view. The computation needed by *SkNN* is simpler. For each predicted window, it first decides whether the new object is qualified for the KNN set in that window. If the answer is "yes", then it updates the KNN in that window using the new object, otherwise the new object will not affect the KNN in that window.

Computation-wise, all three proposed algorithms, namely *Chandi*, *SDOD* and *SkNN* only require a single pass through the new data points at each window slide. In particular, *Chandi* and *SDOD* only require one range query search for each new object, and each new object only communicates with its neighbors once for all shared queries. *SkNN* only requires each new object to update the KNN in each window at most once.

Memory-wise, for all three proposed algorithm, as they all maintain the pattern sets identified by the multiple queries in a single $IntView$ structure, the upper bound of the memory consumption of them for a group of shared queries on the same path is independent from the "length" of this path, namely the number of shared queries in this group This can be proven using the same methods as we used for proving Lemma

p_i: a data point. p_{new}:a new data point. $p_i.T$:p_i's time stamp.
clu_mem:cluster membership. W_i : predicted window. $W_{oldest/newest}$: oldest/newest W on $IntView$. $W.T_{end}$: ending time of W. $W_i.root$: the root predicted view of in W_i. $IntView$: the overall IntView structure. PV : a predicted view.
Q_i : a member query. $Q_i.PV$: a predicted view built for Q_i.

Chandi **(**QG**)**
1 For each new data point p_{new}
2 **If** $p_{new}.T > W_{oldest}.T_{end}$
3 Purge(W_{oldest}); //purge the oldest predicted window // **purge**
4 load p_{new} into index // **load**
5 $neighbors$:=RangeQuerySearch($p_{new}, max(Q_i.\theta^{range})$)
6 UpdateIntView $(p_{new}, neighbors)$ // **IntView Maintenance**
7 **If** $p_{new}.T == T_{output}$
8 Output(); // **output**
9 add new window W_{newest} to $IntView$

Purge(W_i**)**
1 purge any p_i from index If $p_i.T < W_i.T_{end}$
2 remove W_i from $IntView$

UpdateIntView $(p, neighbors)$
1 For i:=1 to $neighbors.size()$
2 DistributeNeighbor($p, neighbors[i], W_{newest}.root$);
3 UpdatePredictedView($p, W_{newest}.root$);

DistributeNeighbor(p_{new}, p_i, PV**)**
1 If $dist(p_{new}, p_j) \leq Qi.\theta^{range}$
2 add p_i to PV.neighbors (neighbors distributed to PV)
3 Else For each $Q_j.PV$ at higher level
4 DistributeNeighbor (p,neighbor, $Q_j.PV$);

UpdatePredictView (p, PV)
1 $p.neighborcount = PV.neighborsinthisview.size()$;
2 For i:=1 to $PV.neighbor.size()$
3 $PV.neighbors[i].neighborcount++$;
4 **If** $PV.neighbors[i]$ becomes a new core
5 HandleNewCore($PV.neighbors[i]$);
6 If $p.neighborcount \geq Q_i.\theta^{cnt}$
7 HandleNewCore(p, PV);
8 For each $Q_j.PV$ at higher level
9 UpdatePredictView $(p, Q_j.PV)$;

HandleNewCore(p, PV)
1 $p.type = core$;
2 $p.clu\_mem$=**new** clu_mem (cluster membership);
3 For i:=1 to $PV.neighbors.size()$
4 **If** $PV.neighbors.type == core$
5 Merge $PV.neighbors[i].clu\_mem$ and $p.clu\_mem$;
6 **If** $PV.neighbors[i].type == noise$
7 $PV.neighbors[i].type := edge$;
8 $PV.neighbors[i].clu\_mem := p.clu\_mem$;
9 For each $Q_j.PV$ at higher level
10 PropagateNewCore($p, Q_j.PV$);

Fig. 16. *Chandi*: Proposed Algorithm for Multiple Density-Based Clustering Queries

p_i: a data point. p_{new}:a new data point. $p_i.T$:p_i's time stamp.
W_i : predicted window. $W_{oldest}/_{newest}$: oldest/newest W on $IntView$.
$W.T_{end}$: ending time of W. $PV.p\_outlier$: the potential outliers in PV.
$IntView^{outlier}$: the overall IntView structure. $W_i.PV$: predicted view built for W_i.
Q_i : a member query. $Q_i.PV$: a predicted view built for Q_i.

SDOD **(**QG**)**
1 For each new data point p_{new}
2 **If** $p_{new}.T > W_{oldest}.T_{end}$
3 Purge(W_{oldest}); //purge the oldest predicted window // **purge**
4 load p_{new} into index // **load**
5 $neighbors$:=RangeQuerySearch($p_{new}, max(Q_i.\theta^{range})$)
6 UpdateIntView $(p_{new}, neighbors)$ // **IntView Maintenance**
7 **If** $p_{new}.T == T_{output}$
8 Output(); // **output**
9 add new window W_{newest} to $IntView$

Purge(W_i**)**
1 purge any p_i from index If $p_i.T < W_i.T_{end}$
2 remove W_i from $IntView$

UpdateIntView $(p, neighbors)$
1 For each W_i on $IntView^{outlier}$
2 UpdatePredictedView($p, W_i.PV$,$neighbors$);

UpdatePredictView $(p, W_i.PV, neighbors)$
1 For each Q_i that maintains W_i (in the ascending order of "strictness")
2 $p.neighborcount := 0$
3 **For** i:=1 to $neighbor.size()$
4 **If** $Distance(p, neighbor[i]) < Q_i.\theta^{range}$
5 $p.neighborcount++$;
6 $neighbors[i].neighborcount++$;
7 **If** $neighbors[i] \in W_i.PV.p\_outliers$ and
$neighbors[i].neighborcount \geq Q_i.\theta^{fra} \times win$
8 mark $neighbors[i]$ as safe non-outlier for Q_i;
9 **If** $neighbors[i]$ has been marked as safe non-outlier for all queries;
10 remove $neighbors[i]$ from $W_i.PV.p\_outliers$;
11 **If** $p \notin W_i.PV.p\_outliers$ $p.neighborcount < Q_i.\theta^{fra} \times win$;
12 put p in $W_i.PV.p\_outliers$;

Fig. 17. *SDOD*: : Proposed Algorithm for Multiple Distance-Based Outlier Detection Queries

4.7. In conclusion, our proposed algorithms, namely *Chandi*, *SDOD*, *SkNN* achieve full sharing for multiple density-based clustering, distance-based outlier queries and kNN queries over the same input stream in terms of both CPU and memory resources.

7. EXPERIMENTAL STUDY

All our experiments are conducted on a HP Pavilion dv4000 laptop with Intel Centrino 1.6GHz processor and 1GB memory, which runs Windows XP operating system. We implemented all algorithms with VC++ 7.0.

Real Streaming Dataset. We used two real streaming data sets. The first data set, GMTI (Ground Moving Target Indicator) data [Entzminger et al. 1999], records the

p_i: a data point. p_{new}:a new data point. $p_i.T$:p_i's time stamp. W_i : predicted window.
$W_{oldest}/_{newest}$: oldest/newest W on $IntView$. $W.T_{end}$: ending time of W.
p^Q: the query object. $PV.KNN$: the KNN (see Section 4.4) in PV.
$PV.p^Q.K^{th}\_NN$: the K^{th} nearest neighbor of p^Q in PV.
$IntView^{kNN}$: the overall IntView structure for kNN queries.
$W_i.PV$: predicted view built for W_i. Q_i : a member query.
$Q_i.PV$: a predicted view built for Q_i.

SDOD **(**QG**)**
1 For each new data point p_{new}
2 If $p_{new}.T > W_{oldest}.T_{end}$
3 Purge(W_{oldest}); //purge the oldest predicted window // **purge**
4 load p_{new} into index // **load**
5 UpdateIntView (p_{new}) // **IntView Maintenance**
6 If $p_{new}.T == T_{output}$
7 Output(); // **output**
8 add new window W_{newest} to $IntView$

Purge(W_i**)**
1 purge any p_i from index If $p_i.T < W_i.T_{end}$
2 remove W_i from $IntView$

UpdateIntView (p)
1 For each W_i on $IntView^{kNN}$
2 UpdatePredictedView($p, W_i.PV$);

UpdatePredictView ($p, W_i.PV$)
1 If $W_i.PV.KNN.size() < K$
2 insert p into $W_i.PV.KNN$;
3 Else
4 If $Dist(p^Q, p) < Dist(p^Q, W_i.PV.p^Q.K^{th}\_NN)$
5 insert p into $W_i.PV.KNN$;
6 remove $W_i.PV.p^Q.K^{th}\_NN$ from $W_i.PV.KNN$;

Fig. 18. *SkNN*: : Proposed Algorithm for Multiple kNN Queries

real-time information of moving objects gathered by 24 different data ground stations or aircrafts in 6 hours from JointSTARS. It has around 100,000 records regarding the information of vehicles and helicopters (speed ranging from 0-200 mph) moving in a certain geographic region. In our experiment, we used all 14 dimensions of GMTI while detecting clusters based on the targets' latitude and longitude. The second dataset is the Stock Trading Traces data (STT) from [INETATS], which has one million transaction records throughout the trading hours of a day.

Alternative Algorithms. As we discussed earlier in Section 4, density-based cluster has one of the most complex pattern structure within the neighbor-based pattern family. Also, given the same window size and input rate, each individual density-based clustering query is also much more system-resource consuming compared with individual distance-based outlier or kNN query. Therefore our experimental evaluation will concentrate on thoroughly evaluate the performance of our proposed algorithm *Chandi* which handles multiple density-based clustering queries. This will include evaluations on its performance under a broad range of parameter settings, how it compares with method using straightforward sharing method, and test to its scalability on the num-

ber of queries that can be handled. Beyond that, theoretical analysis to the performance of other two pattern types will be given later in Sections 7.1 and 7.1.

To evaluate our proposed *Chandi* algorithm, for any input QG, we compare *Chandi*'s performance of two major alternative methods, executing QG with four alternative methods, namely executing one *Extra-N* algorithm [Yang et al. 2009a] for each member query with and without sharing of range query searches (henceforth referred as *Extra-N with rqs* and *Extra-N*), and executing one *IncDBSCAN* algorithm [Ester et al. 1998] for each member query with and without sharing of range query searches (referred as *IncDBSCAN with rqs* and *IncDBSCAN*). The reasons why we choose them are: 1) *Extra-N* algorithm is the only algorithm we are aware of in the literature solving density-based clustering over sliding windows; 2) *IncDBSCAN* algorithm is the most well known method for incremental density-based clustering (but not designed for sliding window semantics).

Experimental Methodologies. We measure two common metrics for stream processing algorithms, namely average processing time for each tuple (CPU time) and memory footprint, indicating the peak memory space required by an algorithm.

As we know, each density-based clustering query using sliding window semantics has four input parameters, namely two pattern parameters: θ^{cnt}, θ^{range}, and two window parameters: win and $slide$. In many cases, the domain knowledge or specific requirements of the analysis tasks may restrict some of them to particular values. For example, a moving object monitoring task may require the θ^{range} to be the maximum distance that two objects can keep wireless communication, and the window size to be the time interval between two successive reports of a single object. Thus the queries submitted by different analysts may only differ on a subsets of these parameters. In our experiments, we first evaluate the four test cases, each has only one of the four parameters different among the member queries.

Evaluation for One-Arbitrary-Parameter Cases. For each test case, we prepare a query group QG with $|QG| = 20$ by randomly generating one input parameter (in a certain range) for each member query, while using common parameter settings on the other three parameters. The parameter settings in our experiment are learned from a pre-analysis of the datasets. In particular, we pick parameter ranges that allow member queries to identify all the different major cluster structures that could be identified in the datasets. In all our test cases, the largest number of clusters identified by a member query is at least five times the smallest number of clusters identified by the other, indicating that the cluster structures identified by different queries vary significantly. In each test case, we use different subsets of QG (sized from 5 to 20) to execute against GMTI data.

Arbitrary θ^{cnt} ***case.*** We use $\theta^{range} = 0.01$, $win = 5000$ and $slide = 1000$, while varying θ^{cnt} from 2 to 20. In this test case, at most 16 clusters are identified by the most restricted query with $\theta^{cnt} = 20$, while at least 3 clusters are identified by the most relaxed one with with $\theta^{cnt} = 3$. As shown in Figures 19 and 20, both the average processing time and the memory space used by all five alternatives increases as the number of member queries increases. This is because more meta-information needs to be computed and stored by all of them. However, the utilization of CPU resources by *Chandi* is significantly lower than those consumed by other alternatives, especially when the number of the member queries increases, and its memory consumption is almost equal to *IncDBSCAN* and much lower than *Extra-N*. This matches with our analysis in Section 4, because in this test case, the predicted windows need to be maintained by *Chandi* for different queries are completely overlapped. Also, since there is no "extra neighborships" existing in any window, the cluster growth information need to be maintained by *Chandi* among the queries are relatively simple. Thus, the system resource consumption of *Chandi* increases very modestly when the num-

ber of member queries increases. While since other alternative methods maintain the progressive clusters independently for different queries, their consumption to system resources increases dramatically when the number of member queries increases.

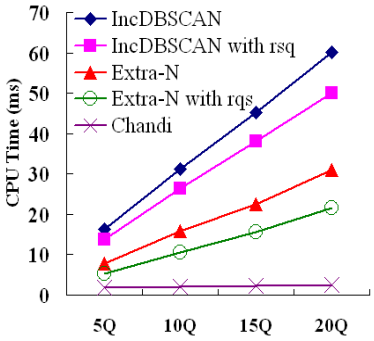

Fig. 19. CPU time used by five competitors in arbitrary θ^{cnt} cases

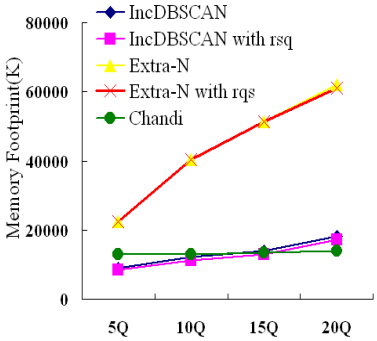

Fig. 20. Memory space used by five competitors in arbitrary θ^{cnt} cases

Arbitrary θ^{range} ***case.*** In this case, we use $\theta^{cnt} = 10$, $win = 5000$ and $slide = 1000$, while varying θ^{range} from 0.01 to 0.1. In this test case, at most 10 clusters are identified by the most restricted query with $\theta^{range} = 0.1$, while at least 2 clusters are identified by the most relaxed one with $\theta^{range} = 0.1$. As shown in the Figures 21 and 22, similar situations can be observed that *Chandi* uses significantly less CPU and memory resources than other alternatives. In this test case, the system resource consumption of *Chandi* increases more as the number of queries increases compared with the previous test cases. This is for of two main reasons. 1) Since the θ^{range} parameters vary among the queries, the range query search cost increases along with the increase of the number of queries even with the range query sharing (each data point needs to figure out its neighbors defined by different queries). 2) As the neighborships identified by different queries differ, such "extra-neighborships" are more likely to cause cluster structure changes and thus requires *Chandi* to maintain more meta-information in $IntView$. The performance of other competitors, especially for *IncDBSCAN*, is affected by the increasing cost of range query searches as well. This is because the performance of *IncDBSCAN* (with rqs or not), which consumes large numbers of range query searches during the purging process, largely relies on the cost of range query searches.

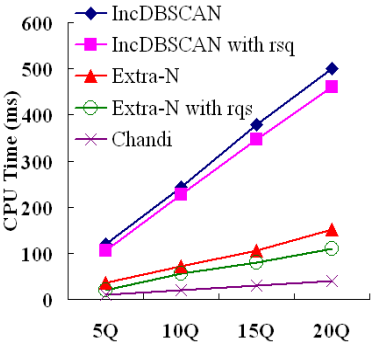

Fig. 21. CPU time used by five competitors in arbitrary θ^{range} cases

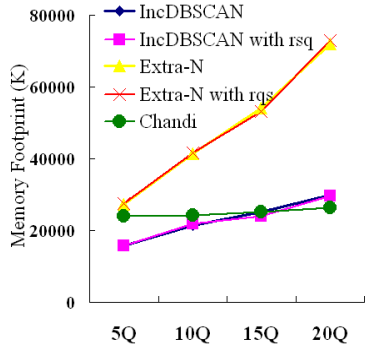

Fig. 22. Memory space used by five competitors in arbitrary θ^{range} cases

Arbitrary win ***case.*** In this case, we use $\theta^{cnt} = 10$, $\theta^{range} = 0.01$, $slide = 500$, while varying win from 1000 to 5000 (we use 500 as granularity for any window parameter). As shown in Figures 29 and 30, we can observe that the performance of *Chandi* is even better compared with the previous test cases. In particular, its resource utilizations for both CPU and memory are almost unchanged as the number of queries increases. This is expected, because in this case *Chandi* only maintains the meta-information for a single query, which is sufficient to answer all the member queries. Thus, the cost of *Chandi* in this case only depends on the query with the largest win, which is independent of the number of queries in the query group.

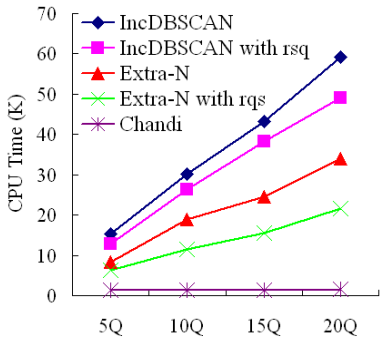

Fig. 23. CPU time used by five competitors in arbitrary win cases

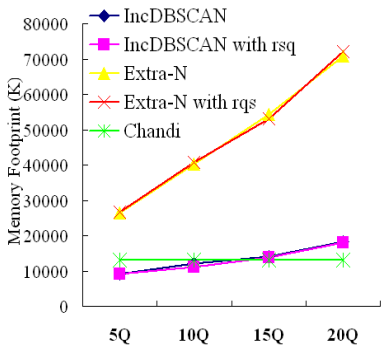

Fig. 24. Memory space used by five competitors in arbitrary win cases

Arbitrary $slide$ ***case.*** In this case, we use $\theta^{cnt} = 10$, $\theta^{range} = 0.01$, $window = 5000$, while varying $slide$ from 500 to 5000. As shown in Figures 25 and 26, the performance of *Chandi* is similar with that in the arbitrary win case. This is because the cost of *Chandi* in this case depends on the number of predicted windows that need to be maintained, which is decided by the query with smallest slide size but does not necessarily increase with the number of queries in the query group.

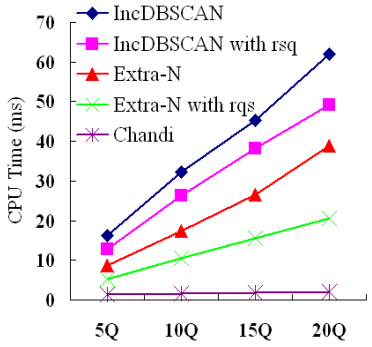

Fig. 25. CPU time used by five competitors in arbitrary $slide$ cases

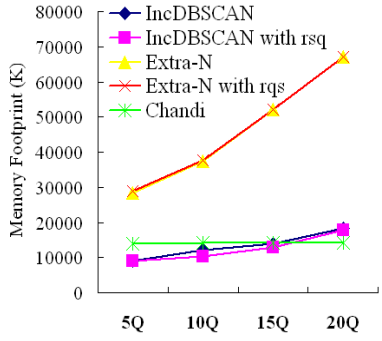

Fig. 26. Memory space used by five competitors in arbitrary $slide$ cases

Evaluation for Two-Arbitrary-Parameter Cases We evaluate two test cases, each has two of the four parameters different among the member queries. In the first test case, member queries have arbitrary pattern parameters but common window parameters, indicating that they may have different definition to the clusters but always

have the same query window. In the second test case, member queries have arbitrary window parameters but common pattern parameters, indicating they may have different query windows but have the same definition to the clusters.

Arbitrary Pattern Parameters. In this case, we use $win = 5000$, $slide = 1000$, while vary θ^{cnt} from 2 to 20 and θ^{range} from 0.01 to 0.1. As shown in Figures 27 and 28, *Chandi* still consumes significantly less CPU time compared with the other alternatives, although the increase of CPU consumption caused by the increase of member queries is more obvious. This is because totally arbitrary pattern parameters lead to an even larger difference in the clusters identified by different queries, and thus increase the maintenance costs of *Chandi*. In particular, in this test case, the largest number of clusters identified by the number query with $\theta^{range} = 0.01$ and $\theta^{cnt} = 14$ reaches 35, while the smallest number of clusters identified by the query (with $\theta^{range} = 0.1$ and $\theta^{cnt} = 3$) is only 2. The memory space used by *Chandi* in this case is much less than *Extra-N* while being only slightly higher than *IncDBSCAN*. Again, this is caused by the more incremental information existing among the predicted views maintained by *Chandi*. However, as the CPU performance of *IncDBSCAN* is much worse than *Chandi*, the overall performance of *Chandi* is still much better.

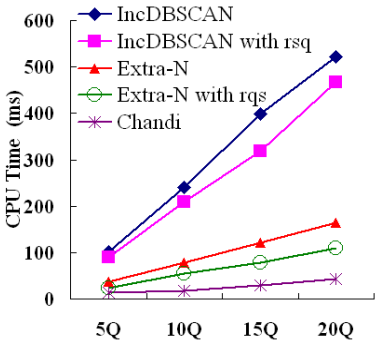

Fig. 27. CPU time used by five competitors in arbitrary pattern parameter cases

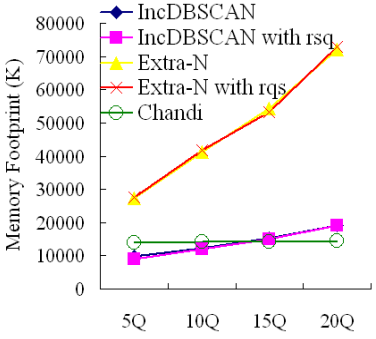

Fig. 28. Memory space used by five competitors in arbitrary pattern parameter cases

Arbitrary Window Parameters. In this case, we use $\theta^{cnt} = 10$ and $\theta^{range} = 0.01$, while varying win from 1000 to 5000 and $slide$ from 500 to 5000 (for any query Q_i, $Q_i.slide < Q_i.win$). As shown in Figures 29 and 30, the performance of *Chandi* is similar with that observed in the arbitrary win or $slide$ case. This is because, although the queries now have arbitrary settings on both parameters, such fact does not affect the principle of how the "meta query" strategy works. In particular, the cost of answering a query group still only depends on the largest win in the query group and the number of predicted views that need to be maintained. Both do not necessarily increase along with the number of queries.

General Case: Four Arbitrary Parameters. Finally, we evaluate the general case, with all four parameters being arbitrary. We divide this experiment into three cases, each measuring the performance of the algorithms when executing different numbers of queries. In particular, for each test case, we generate 30 query groups each with N member queries (N equals to 20, 40 and 60 for three cases respectively). Each query group is independently generated, and the member queries in each group are randomly generated with parameter settings: $\theta^{cnt} = 2$ to 20, $\theta^{range} = 0.01$ to 0.1, $win =$ 1000 to 5000, and $slide = 500$ to 5000. For each test case, we measure the average cost of each algorithm for executing all 30 query groups. Beyond that, we zoom into the

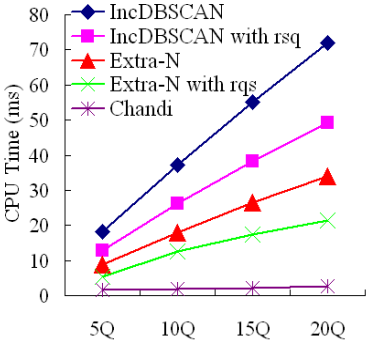

Fig. 29. CPU time used by five competitors in arbitrary window parameter cases

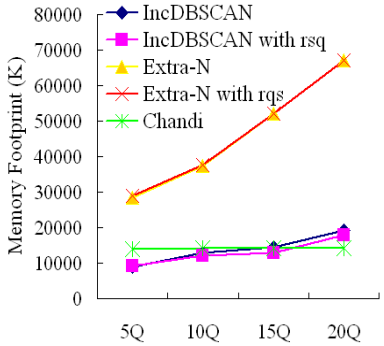

Fig. 30. Memory space used by five competitors arbitrary window parameter cases

overall average cost of each algorithm, and measure the cost caused by each specific subtask. In particular, the CPU measurement is divided into two parts, namely the CPU time used by range query searches and that used by cluster maintenance. For the memory space consumed, we distinguish between the memory used by raw data (for storing actual tuples) and the memory used for meta-data.

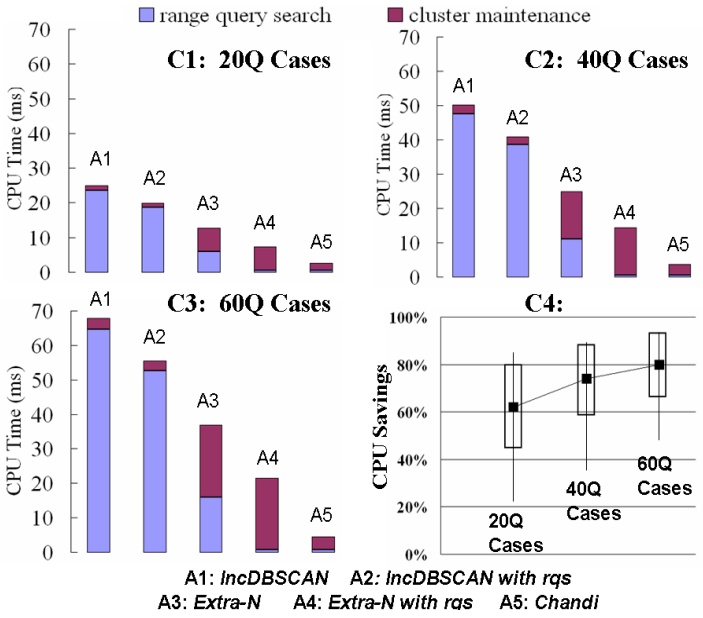

Fig. 31. Detailed comparison on CPU time consumption of five algorithms

As shown in charts C1, C2 and C3 in Figure 31, we observe that the average CPU time used by *Chandi* is 70, 76, and 85 percent lower then the best alternative method, *Extra-N with rqs*, in the three cases respectively. In particular, the CPU time used by *Chandi* to conduct range query searches is always less than 10% compared with that needed by *IncDBSCAN* with rqs. This is because *Chandi* only requires each new data point to run one range query search when it arrives at the system, while *IncDBSCAN* relies on repeated range query searches to determine the cluster changes. The

CPU time used by *Chandi* to maintain meta-information is at least 62% less than that used by *Extra-N with rqs*. This is because *Chandi* updates the meta-information for different queries integrally, while *Extra-N* maintains them independently.

Besides the comparison of the average system resource consumption, we also measure the savings of *Chandi* for each individual query group in all three test cases. In particular, for each query group, we measure the difference in resource utilization between *Extra-N with rsq* and *Chandi*, which corresponds to the difference between executing them using the best existing technique and our proposed strategy. More specifically, for each group, we first calculate the difference on CPU (or memory) utilization between two *Chandi* and *Extra-N*. Then, we use the difference to divide that used by *Extra-N with rqs* to get the saving percentage achieved by *Chandi*. As shown in C4 of Figures 31, *Chandi* never performs worse than *Extra-N with rqs* for any query group. For the first test case (each query group has 20 queries), the average savings achieved by *Chandi* in terms of CPU time are 62%. Although the minimum savings in this case among the 30 groups is 23%, the maximum savings reach 84% , and the standard deviation is only 19% . As the number of queries in each group increases, the savings achieved by *Chandi* are even higher in the other two test cases. In particular, the average savings achieved by *Chandi* of CPU time increases to 80% when the number of queries in each group increases to 60. The minimum and maximum savings on CPU time increases to 45% and 92% respectively in this case, and the standard deviation of the savings decreases to 12%. This shows the promise of *Chandi* that, for a query group with 60 queries, it can achieve savings between 73% to 92% of CPU time in most of the cases. Among the 30 queries in this query group, 23 of them fall into this range. The average savings achieved by *Chandi* on memory space in this 60-query cases is 89%.

Evaluation for Scalability. Now we evaluate the scalability of the algorithms in terms of the number of queries they can handle under a certain data rate. In this experiment, we use *Extra-N*, *Extra-N with rqs* and *Chandi* to execute query groups sized from 10 to 1000 against GMTI data. Similar with the earlier experiment, the member queries in the query group are randomly generated with the arbitrary parameter settings in certain ranges. In particular, the parameters settings in this experiment are $\theta^{cnt} = 2$ to 30, $\theta^{range} = 0.001$ to 0.01, $win = 1000$ to 5000, and $slide = 500$ to 5000.

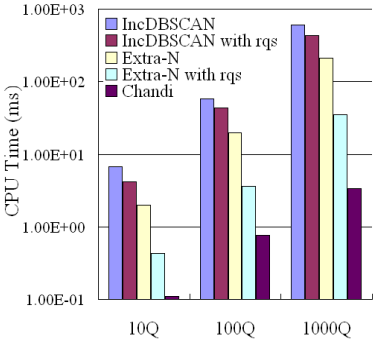

Fig. 32. CPU time used by five competitors in logarithmic scale

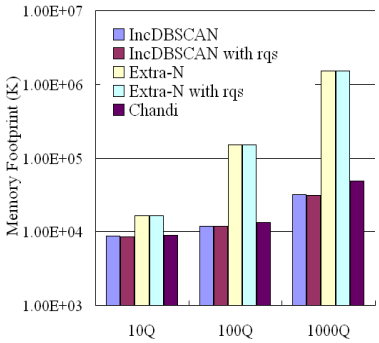

Fig. 33. Memory space used by five competitors in logarithmic scale

As shown in Figures 32 and 33, both the CPU time and the memory space used by *Chandi* increase modestly as the number of member queries increases. In particular, the CPU time consumed by *Chandi* increases around 6 times when the number of

queries grows from 10 to 100 (increased 9 times), and then it increases less than 4 times when number of queries grows from 100 to 1000. Thus totally the CPU time consumed by *Chandi* increases 33 times when the number of queries increased from 10 to 1000, which is 100 times. Such increase for *Extra-N* and *Extra-N with rqs* are 105 times and 89 times respectively. More specifically, in our test cases, the average processing time (CPU) for each tuple used by *Chandi* to execute the 100-query and 1000-query query groups are 0.76ms and 3.3ms respectively. This indicates that our system can comfortably handle 100 queries under a 1000 tuples per second data rate, and handle 1000 queries under a 300 tuples per second data rate. For the memory space used, *Chandi* has even better performance as its utilization of memory space only increases 5 times when the number of queries increases from 10 to 1000, while such increase for *Extra-N* and *Extra-N with rqs* are both 98 times.

Conclusion for Experimental Study. Generally, *Chandi* is more efficient than other alternative methods in terms of both CPU and memory utilization when executing multiple density-based clustering queries specified on the same input stream. *Chandi* achieves most sharing when only one of the four parameters differ among the member queries. Among the four one-arbitrary-parameter cases, *Chandi* achieves most sharing in the arbitrary win case, while least is achieved in the arbitrary θ^{range} case. For the two-arbitrary-parameter cases, *Chandi* performs better when the member queries have arbitrary window parameters rather than arbitrary pattern parameters. For the general cases, where the member queries have arbitrary parameter settings on all four parameters, *Chandi* still clearly outperforms the other alternative methods by achieving on average 60 percent savings for CPU time and 84 percent savings in memory space. Lastly, *Chandi* shows a good scalability in terms of handling a large number (hundreds or even thousands) of queries under a high data rate.

7.1. Discussion

Now we discuss the potential performance of our proposed methods for other neighbor-based pattern types. The savings expected for our proposed methods for the other neighbor-based pattern types are similar to those observed from our above comprehensive case study with the clustering-pattern type. Thus we only briefly review these expected savings below.

Performance Analysis for Distance-Based Outlier Queries. For individual **distance-based outlier queries**, the CPU processing resources for query execution is composed by two major parts, namely running range query searches for each new object arriving at the system and updating the neighbor-counts for each data point whose neighborhood is affected by those new objects. This is really identical to the situation observed for density-based clustering queries. In our experiments for density-based clusters, for a single query execution, the cost for neighbor searches constitutes around 40 percent of the overall CPU processing costs, while and the remaining costs is primarily consumed by updating the cluster structures. For distance-based outlier queries, this percentage of CPU utilization for conducting neighbor searches will surely be even much higher, as the pattern structures to be updated for maintaining outliers are simpler and thus clearly need much less computational for processing updates on them.

Same as clustering queries, the cost of the neighbor searches can be completely shared among all queries using our method. In particular, we simply need to run one single range query search for each new data point (using the largest range, of course) to collect all the point's neighbors for all participating queries. This indicates that, for executing a large group of queries, the costs for conducting neighbor searches can almost be completely saved. This is because the cost of running a single range query search for each data point is neglectable compared with the cost required for updating

the neighbor counts for for potentially large number of queries upon the arrival of each data point.

For the second part of the cost, namely the cost associated with the neighbor count maintenance, the savings depend on the overlap of the stream windows and on the number of queries that identify the same number of neighbors for each data point. Thus, the savings that can be achieved in this component may vary somewhat. However, as indicated above we can completely save the costs associated with neighbor searches (at least 40 percent). In addition, we can expect to have fairly significant amount of sharing among the pattern updating processes for the different patterns, especially for larger query groups (where similar queries tend to be more likely). Thus one can safely expect that savings achieved by our method for distance-based outlier queries will be en par or even outperform those for the cluster-based queries.

Memory-wise, the cost for individual distance-based outlier query execution is composed by both raw and meta data storage. In particular, each query needs to store all the valid data points in the window and the neighbor counts for each data point. For this pattern type, the major memory savings that could be achieved by our method should come from the storage for raw data, as now using incremental pattern representation, we only need to store one reference for each data point for all queries. The storage for meta information, namely the neighbor counts, may not be saved significantly. This is because each neighbor count maintained by a query is just an integer. Storing a new count number for a query or only storing the increments from a "stricter" query will not make any different in terms of memory usage.

Performance Analysis for kNN Queries. Computation-wise, the major cost for executing individual kNN queries comes from updating the k nearest neighbors when the new data points arrive. As we discussed in Section 4, the k (the largest k setting among all queries) nearest neighbors of the query object are incrementally stored, and thus the updating effort can be completely shared. In particular, if a new data point qualifies for the kNN of the query object, we only need two operations to first put the new data point into a single kNN set and remove the previous Kth nearest neighbor (the farthest one). This cost is much cheaper than placing the new data point into the kNN sets for all queries, and more importantly, will almost not affected by the number of queries in the query group. Therefore, we envision that the percentage of savings achieved by our method on CPU time will increase linearly the number of queries increases.

Memory-wise, as discussed in Section 4, using our shared execution method, the information (raw and meta) needs to be stored by a group of query is same with that needs to be stored by a single query (the query with largest k setting). Therefore the memory cost of our method is independent from the number of queries in the query group.

8. RELATED WORK

Data mining, as a general concept for extracting or "mining" knowledge from large amounts of data, covers a rather diverse range of mining tasks. Also, any data mining task may consist of an iterative sequence of the following steps: 1) data cleaning, 2) data integration, 3) data selection, 4) data transformation, 5) pattern extraction, 6) pattern evaluation and 7) knowledge representation [Han 2005]. In this work, we focus on the problem of shared execution strategies for multiple neighbor-based pattern mining queries in streaming windows. Such query execution techniques falls into the pattern extraction step of data mining, which is an essential process where intelligent methods are applied to extract patterns form well prepared data.

Next, we review a categorization of the common data mining tasks from the literature and show where our target query types lie in this categorization. Traditionally,

data mining techniques [Zhang et al. 1996; Ester et al. 1996; Ankerst et al. 1999; Knorr and Ng 1998; Breunig et al. 2000; Jagadish et al. 2005; Koudas et al. 2004; Nutanong et al. 2008] are designed for static environments with large volumes of stored data. More recently, as stream applications are becoming prevalent, the problem of mining streaming data is being tackled [Aggarwal et al. 2003; Yang et al. 2009a; Babcock et al. 2003; Subramaniam et al. 2006; Angiulli and Fassetti 2007; Mouratidis and Papadias 2007]. Based on the dynamics of the input data, we can first divide the data mining tasks into ***static data mining*** and ***stream data mining***. Clearly, our tasks of mining neighbor-based patterns in streaming windows falls into the ***stream data mining*** category.

Second, for both *static* and streaming data mining, Han and Kamber [Han 2005] divide the data mining tasks into two categories based on their purposes, namely ***descriptive*** and ***predictive*** mining. In particular, descriptive mining tasks characterizes the general properties of the data in the database. Predictive mining performs inferences on the current data in order to make predictions for future data. Typical *predictive* data mining tasks include *classification*, *prediction* and *trend mining* [Han 2005]. Since our task of mining neighbor-based patterns in the data streams aims to find specific patterns in the most recent portion of the stream, which does not necessarily predict anything about the future, our task falls into the ***descriptive data mining*** category.

Among the *descriptive streaming data mining* tasks, they can be further divided them into following categories based on their distinct key characteristics [Han 2005].
1) ***graph mining***: The one-to-one relationships, namely the (directed or undirected, weighted or unweighted) *edges* among objects and the topological *relationships* among objects are the key factors that defines the patterns which *graph mining* mines for.

2) ***association rule and correlation mining***: *Frequency* is the key factor for association rules and correlations. Namely, the frequency of a certain type of objects to appear together or the frequency of a certain relationship existing among certain attributes of the objects defines association rules and correlations.

3) ***text mining***: In *text mining*, the appearance of certain key workds and relationships among their linguistic meanings of these key words are the key factors that define the patterns in the text.

4) ***sequence mining***: In *sequence mining*, the *time sequences* in which certain events happen or the *time sequences* in which values of an attribute appears are the key factors for the *sequence mining* process.

5) ***clustering***: Cluster mining processes aim to divide the input objects into different *groups*, each having its own characteristics. Maximizing the *similarity* among the objects within the same groups and the *disimilarity* among objects within the different groups are the goals pursued by the clustering algorithms.

6) ***outlier mining***: The *abnormality* of some objects compared to the majorities is the knowledge that the *outlier mining* process mines for.

7) ***web page mining***: The *textual content* and the *link structures* among the web pages is explored by *web page mining*.

8) ***multimedia mining***: Mining on *images* and *voices* distinguishes *multimedia mining* from other mining tasks.

Given this categorization, if we analyze our neighbor-based pattern mining tasks from the perspective of their general **purpose**, they are clearly related to multiple categories, including cluster and outlier mining. However, as discussed earlier in Section 2, if we analyze the characteristics of the target **pattern structures** of neighbor-based pattern mining queries, namely how these **pattern strcutures** are defined, all the neighbor-based pattern mining queries can be viewed as subclass of the *graph mining* category. In conclusion, our target neighbor-based pattern mining queries over

streaming windows fall into the *graph mining* category in *descriptive streaming data mining*.

Within our target neigbor-based pattern types, density-based clustering was first proposed in [Ester et al. 1996] as DBSCAN algorithm for static data. Later an Incremental DBSCAN [Ester et al. 1998] algorithm was introduced to incrementally update density-based clusters in data warehouse environments. However, as both analytically and experimentally shown in [Yang et al. 2009a], since all optimizations in [Ester et al. 1998] were designed for single updates (a single deletion or insertion) to the data warehouse, it may fit well for the relatively stable data warehouse environment, but it is not scalable to highly dynamic streaming environments. Our experimental study conducted in Section 7 also demonstrates that executing multiple queries using [Ester et al. 1998] in streaming environments is prohibitively expensive in terms of CPU resource consumption.

Algorithms for density-based clustering queries over streaming data include [Yang et al. 2009a; Chen and Tu 2007; Cao et al. 2006]. Among these works, [Chen and Tu 2007] and [Cao et al. 2006] have goals different from ours, because they are neither designed to identify the individual members in the clusters nor enforce the sliding window semantics for the clustering process. Thus these two algorithms cannot be applied to solve the problem we tackle in this work. [Yang et al. 2009a] is the only algorithm we are aware of that detects density-based clusters in sliding windows. Our experimental study conducted in Section 7 shows that our shared execution strategy largely outperforms the strategy of using this algorithm independently for each query. [Yang et al. 2010] builds a visual system to allow analysts to interatively explore density-based clusters in streaming environments.

[Angiulli and Fassetti 2007] and [Mouratidis and Papadias 2007] discuss the problem of detecting distance-based outliers and top-k nearest neighbors in data streams respectively. Again, these works concentrate on single query execution only. We borrow the basic ideas of maintaining meta-information, such as potential outlier sets and k nearest neighbors from them. However, instead of maintaining such meta-information independently for each query, we developed the integrated maintenance strategies for shared execution among multiple queries, and thus achieve significant savings on both CPU and memory resources.

As a general query optimization problem, multiple query optimization has been widely studied for not only static but also streaming environments. Such techniques can be roughly divided into two different groups, namely “plan level” and “operator level” sharing. “Plan level” sharing techniques [Liu et al. 2008; Chen et al. 2000; Chen et al. 2002] aim to allow the different input queries to share the common operators across their query plans, and thus lower the overall costs for multiple query execution. Operator level sharing studies the sharing problem on a finer granularity, namely within the individual operators. In particular, they aim to share the operator state as well as the query processing computation within a single operator, when multiple queries have similar yet not identical operator specifications. For example, two queries may calculate aggregations for the same input stream but using different window sizes. The problem we solve in this paper falls into the operator level sharing category.

Previous research efforts discussing such operator level sharing techniques focus on simple operators, such as selection and join operators [Madden et al. 2002; Hammad et al. 2003; Krishnamurthy et al. 2004; Wang et al. 2006; Zhang et al. 2005], and aggregation operators [Krishnamurthy et al. 2006; Arasu and Widom 2004; Zhang et al. 2005]. To our best knowledge, none of them discuss the sharing for clustering operators. Some general principles used in these works, such as query containment [Hammad et al. 2003], can also be applied in our context (used in sharing range query

searches for our solution). However, the key problem we address in this work, namely the integrated maintenance of density-based cluster structures identified by multiple queries, is different from the optimization effort required by selection, join or aggregation sharing. In particular, the meta-information we need to maintain, namely the cluster structures defined by individual cluster member objects as well as their interrelationships, is much more complex than those for selection, join or aggregation operators, which are usually pair-wise relations or simply numbers (aggregation results). Efficient maintenance of such meta-information requires thorough analysis of the properties of density-based cluster structures, which is a key contribution of our work. This has not been studied in any of these works.

9. CONCLUSION

In this work, we present the first framework for the efficient shared processing of a large number of neighbor-based pattern mining requests over streaming windows. It is the first step of applying multiple query optimization principles from the field of databases to process large numbers of data mining requests in stream environments. We propose several general optimization principles that are applicable to different (at least three) neighbor-based pattern mining query types. Both our analytical and experimental studies show that these principles can bring significant system resource sharing among multiple queries. In particular, our proposed algorithms *Chandi*, *SDOD* and *SkNN*, which are based on these optimization principles, respectively achieve full sharing of both CPU and memory utilization when simultaneously executing multiple density-based clustering, distance-based outlier and kNN queries. Our experimental study shows that, our proposed solution *Chandi* that handles the density-based clustering queries, which has the most complex pattern structure within neighbor-based pattern family, is on average four times faster than the best alternative method while using 85% less memory space. More savings can be achieved if the queries have similar parameter settings. *Chandi* also exhibits excellent scalability in terms of being able to handle large numbers of queries under high speed input streams in our experiments. Our performance analysis for distance-based outlier and kNN queries shows that the similar performance can be expected from our proposed strategies for those two pattern types as well.

10. ACKNOWLEDGEMENTS

We thank Zaixian Xie, Zhengyu Guo, Venkatesh Raghavan, Abhishek Mukherji, Karen Works, Mo Liu, Mingzhu Wei, Di Wang, Xika Lin, Chuan Lei, Lei Cao, Medhabi Ray, Yingmei Qi, Kaiyu Zhao, Dazhi Zhang and other members of XMDV and DSRG group at WPI for providing input on this work.

REFERENCES

ACHTERT, E., BÖHM, C., KRÖGER, P., KUNATH, P., PRYAKHIN, A., AND RENZ, M. 2006. Efficient reverse k-nearest neighbor search in arbitrary metric spaces. In *ACM SIGMOD Conference*. 515–526.

ACHTERT, E., KRIEGEL, H.-P., KRÖGER, P., RENZ, M., AND ZÜFLE, A. 2009. Reverse k-nearest neighbor search in dynamic and general metric databases. In *EDBT Conference*. 886–897.

AGGARWAL, C. C., HAN, J., WANG, J., AND YU, P. S. 2003. A framework for clustering evolving data streams. In *VLDB Conference*. 81–92.

ANGIULLI, F. AND FASSETTI, F. 2007. Detecting distance-based outliers in streams of data. In *CIKM Conference*. 811–820.

ANKERST, M., BREUNIG, M. M., KRIEGEL, H.-P., AND SANDER, J. 1999. Optics: Ordering points to identify the clustering structure. In *SIGMOD Conference*. 49–60.

ARASU, A., BABU, S., AND WIDOM, J. 2006. The cql continuous query language: semantic foundations and query execution. *VLDB J. 15,* 2, 121–142.

ARASU, A. AND WIDOM, J. 2004. Resource sharing in continuous sliding-window aggregates. In *VLDB Conference*. 336–347.

BABCOCK, B., DATAR, M., MOTWANI, R., AND O'CALLAGHAN, L. 2003. Maintaining variance and k-medians over data stream windows. In *PODS*. 234–243.

BREUNIG, M. M., KRIEGEL, H.-P., NG, R. T., AND SANDER, J. 2000. Lof: identifying density-based local outliers. *SIGMOD Rec. 29,* 2, 93–104.

CAO, F., ESTER, M., QIAN, W., AND ZHOU, A. 2006. Density-based clustering over an evolving data stream with noise. In *SDM*.

CHEN, J., DEWITT, D. J., AND NAUGHTON, J. F. 2002. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *IEEE ICDE Conference*. 345–356.

CHEN, J., DEWITT, D. J., TIAN, F., AND WANG, Y. 2000. Niagaracq: A scalable continuous query system for internet databases. In *ACM SIGMOD Conference*. 379–390.

CHEN, Y. AND TU, L. 2007. Density-based clustering for real-time stream data. In *ACM KDD Conference*. 133–142.

ENTZMINGER, J. N., FOWLER, C. A., AND KENNEALLY, W. J. 1999. Jointstars and gmti: Past, present and future. *IEEE Transactions on Aerospace and Electronic Systems 35,* 2, 748–762.

ESTER, M., KRIEGEL, H.-P., SANDER, J., WIMMER, M., AND XU, X. 1998. Incremental clustering for mining in a data warehousing environment. In *VLDB Conference*. 323–333.

ESTER, M., KRIEGEL, H.-P., SANDER, J., AND XU, X. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *ACM KDD Conference*. 226–231.

GORAWSKI, M. AND MALCZOK, R. 2006. Aec algorithm: A heuristic approach to calculating density-based clustering *ps* parameter. In *ADVIS*. 90–99.

HAMMAD, M. A., FRANKLIN, M. J., AREF, W. G., AND ELMAGARMID, A. K. 2003. Scheduling for shared window joins over data streams. In *VLDB*. 297–308.

HAN, J. 2005. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

INETATS, I. Stock trade traces. http://www.inetats.com/.

JAGADISH, H. V., OOI, B. C., TAN, K.-L., YU, C., AND ZHANG, R. 2005. idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst. 30*, 364–397.

KNORR, E. M. AND NG, R. T. 1998. Algorithms for mining distance-based outliers in large datasets. In *VLDB Conference*. 392–403.

KOUDAS, N., OOI, B. C., TAN, K.-L., AND ZHANG, R. 2004. Approximate nn queries on streams with guaranteed error/performance bounds. In *VLDB Conference*. 804–815.

KRISHNAMURTHY, S., FRANKLIN, M. J., HELLERSTEIN, J. M., AND JACOBSON, G. 2004. The case for precision sharing. In *VLDB Conference*. 972–986.

KRISHNAMURTHY, S., WU, C., AND FRANKLIN, M. J. 2006. On-the-fly sharing for streamed aggregation. In *ACM SIGMOD Conference*. 623–634.

LI, J., MAIER, D., TUFTE, K., PAPADIMOS, V., AND TUCKER, P. A. 2005. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record 34,* 1, 39–44.

LIU, Z., PARTHASARATHY, S., RANGANATHAN, A., AND YANG, H. 2008. Near-optimal algorithms for shared filter evaluation in data stream systems. In *ACM SIGMOD Conference*. 133–146.

MADDEN, S., SHAH, M. A., HELLERSTEIN, J. M., AND RAMAN, V. 2002. Continuously adaptive continuous queries over streams. In *ACM SIGMOD Conference*. 49–60.

MOURATIDIS, K. AND PAPADIAS, D. 2007. Continuous nearest neighbor queries over sliding windows. *IEEE Trans. Knowl. Data Eng. 19,* 6, 789–803.

NUTANONG, S., ZHANG, R., TANIN, E., AND KULIK, L. 2008. The v*-diagram: a query-dependent approach to moving knn queries. *PVLDB 1,* 1, 1095–1106.

SOLIMAN, M. A., ILYAS, I. F., AND KOUDAS, N. 2007. Finding skyline and top-k bargaining solutions. In *IEEE ICDE Conference*. 1263–1267.

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

SUBRAMANIAM, S., PALPANAS, T., PAPADOPOULOS, D., KALOGERAKI, V., AND GUNOPULOS, D. 2006. Online outlier detection in sensor data using non-parametric models. In *VLDB Conference*. 187–198.

WANG, S., RUNDENSTEINER, E. A., GANGULY, S., AND BHATNAGAR, S. 2006. State-slice: New paradigm of multi-query optimization of window-based stream queries. In *VLDB Conference*. 619–630.

YANG, D., GUO, Z., XIE, Z., RUNDENSTEINER, E. A., AND WARD, M. O. 2010. Interactive visual exploration of neighbor-based patterns in data streams. In *ACM SIGMOD Conference*. 1151–1154.

YANG, D., RUNDENSTEINER, E. A., AND WARD, M. O. 2009a. Neighbor-based pattern detection for windows over streaming data. In *EDBT Conference*. 529–540.

YANG, D., RUNDENSTEINER, E. A., AND WARD, M. O. 2009b. A shared execution strategy for multiple pattern mining requests over streaming data. *PVLDB 2,* 1, 874–885.

YUAN, Y., LIN, X., LIU, Q., WANG, W., YU, J. X., AND ZHANG, Q. 2005. Efficient computation of the skyline cube. In *VLDB Conference*. 241–252.

ZHANG, R., KOUDAS, N., OOI, B. C., AND SRIVASTAVA, D. 2005. Multiple aggregations over data streams. In *ACM SIGMOD Conference*. 299–310.

ZHANG, S., MAMOULIS, N., AND CHEUNG, D. W. 2009. Scalable skyline computation using object-based space partitioning. In *ACM SIGMOD Conference*. 483–494.

ZHANG, T., RAMAKRISHNAN, R., AND LIVNY, M. 1996. Birch: An efficient data clustering method for very large databases. *SIGMOD Record, vol.25(2), p. 103-114*.