WPI-CS-TR-09-04 April 2009

Multiple Query Optimization for Density-Based Clustering Queries over Streaming Windows

by

Di Yang
Elke A. Rundensteiner
Matthew O. Ward

Computer Science Technical Report Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

Multiple Query Optimization for Density-Based Clustering Queries over Streaming Windows *

Di Yang,Elke A. Rundensteiner,Matthew O. Ward
Worcester Polytechnic Institute
100 Institute Road
Worcester, MA, USA
diyang, rundenst, matt@cs.wpi.edu

ABSTRACT

In diverse applications ranging from stock trading to traffic monitoring, popular data streams are typically monitored by multiple analysts for patterns of interest. These analysts thus may submit similar pattern mining requests, such as mining for clusters or outliers, yet customized with different parameter settings. In this work, we present an efficient shared execution strategy for a large number of density-based cluster detection queries with arbitrary parameter settings. Given the high algorithmic complexity of the clustering process and the real-time responsiveness required by streaming applications, serving multiple such queries in a single system is extremely resource intensive. The naive method of detecting and maintaining clusters for different queries independently is often infeasible in practice, as its demands on system resources increase dramatically with the cardinality of the query workload. To overcome this, we analyze the interrelations between the cluster sets identified by queries with different parameters settings, considering both pattern-specific and window-specific parameters. We characterize the conditions under which a proposed *growth property* holds among these cluster sets. By exploiting this *growth property* we propose a uniform solution, called *Chandi*, which represents these identified cluster sets as one single compact structure and performs integrated maintenance on them – resulting in significant sharing of computational and memory resources. Our comprehensive experimental study, using real data streams from domains of stock trades and moving object monitoring, demonstrates that *Chandi* is on average four times faster than the best alternative methods, while using 85% less memory space in our test cases. It also shows that *Chandi* scales in handling large numbers of queries, on the order of hundreds or even thousands, under high input data rates.

1. INTRODUCTION

Motivation. 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. Although previous research efforts have developed efficient algorithms for streaming pattern detection [1, 2, 3], they focused on processing one single specific data mining task. Little effort has been made towards simultaneous execution of multiple pattern mining queries against the same input stream. In this work, we provide a framework to efficiently execute a large number (on the order of hundreds or even thousands) of pattern mining queries with different parameter settings, while still achieving real-time responsiveness required by stream applications.

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. For example, a query asking for the stocks that drop or rise significantly in most recent transactions can be considered as a parameterized query. It needs analysts to specify parameters that define their notion of "significance" in price fluctuation and the meaning of "most recent transactions" based on their application semantics. Other examples of parameterized queries include density-based clustering [4] that require a range and count threshold as input, and K-means style clustering [5] that requires K as input.

Given the prevalence of parameterized queries, a stream processing system often needs to handle multiple such queries. Multiple analysts monitoring the same input stream may submit the same pattern search but using different parameter settings. Using the earlier example, multiple analysts may have different interpretations of the "significance" in price fluctuation (say from $10 - 80\%$ of the original price). Even a single analyst may submit multiple queries with different parameter settings, because determining the best input parameters is difficult when faced with an unknown input stream. In static environments, this problem can be tackled by conducting pre-analysis of the static datasets or repeatedly trying different parameter settings until satisfactory results have been obtained. However, 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 at least a portion of the stream. Thus, an ideal stream processing system should be able to accommodate multiple queries covering many, if not all, major parameter settings of a parameterized query, and thus capture all potentially valuable patterns in the stream.

Challenges. In this work, we tackle multiple query optimization for handling multiple density-based clustering queries over sliding windows. Each query of this type has four input parameters: two pattern parameters, a range threshold θ^{range} and a count threshold θ^{cnt}, and two window parameters: window size win and slide size $slide$. Any such query can be expressed by the template given in Figure 1.

9*This work is supported under NSF grant IIS-0414380. We thank our collaborators at MITRE Corporation, in particular, Jennifer Casper and Peter Leveille, for providing us the GMTI data stream generator

VLDB '09, August 24-28, 2009, Lyon, France

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

Figure 1: Templated density-based cluster detection query for sliding windows over data steam

Realizing parameterization for this query type is important not only because its input parameters, such as θ^{range} and θ^{cnt}, are difficult to determine without pre-analysis of the stream data, but also because even a slight difference in any of these parameters may cause totally different cluster structures to be identified. Figure 8 shows different clusters identified in a subpart of the GMTI data stream [6] by density-based clustering queries with different pattern parameter settings.

Given the high algorithmic complexity of density-based clustering, serving a large number of such clustering queries in a single system is highly resource intensive. The naive method of maintaining progressive clusters (clusters identified in the previous window) for multiple queries independently has prohibitively high demands on both computational and memory resources.

Thus, the key problem we need to solve is to design a cluster maintenance mechanism that achieves effective sharing of system resources for multiple queries. This is a challenging problem, because the meta-information required to be maintained by this query type is more complex than those for SQL query operators, such as selection, join or aggregation. More specifically, we need to maintain the identified cluster structures, which are defined by the tuples and the global topological relationships among tuples. While SQL operators usually maintain pair-wise relations between two tuples (independent from the rest of the tuples) or simply numbers (aggregation results). The techniques [7, 8] regarding sharing among SQL queries thus cannot be used to solve our problem.

Proposed Solution. Our proposed solution allows arbitrary parameter settings for queries on all four input parameters, including both the pattern-specific and the window-specific ones. We first discover that given the same window parameters, if a query Q_i's pattern parameters are "more restricted" than those of another query Q_j, a *growth property* (Section 4) holds between the cluster sets identified by Q_i and Q_j. This *growth property* allows us to incrementally organize the clusters identified by multiple queries into one single integrated structure, called *IntView*. As a highly compact structure, *IntView* saves the memory space needed for storing the clusters identified by multiple queries. More importantly, *IntView* also enables integrated maintenance for the progressive clusters of multiple workload queries, and thus effectively saves the computational resources otherwise needed when maintaining them independently.

We also propose a "*meta query* strategy", which uses a single meta query to represent all workload queries for which their pattern parameters are the same, while their window parameters may differ. 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 window snapshots that need to be maintained in the system. We show in Section 6 that our meta query technique successfully transforms the problem of maintaining multiple queries into the execution of a single query.

Finally, we combine the *IntView* technique and meta query strategy to form one integrated solution. We call it *Chandi* [1], which stands for Clustering high speed streaming data for multiple queries using integrated maintenance. *Chandi* integrates the progressive clusters detected by all workload queries into a single structure, and thus realizes incremental storage and maintenance for this meta information across the queries. Computation-wise, for each window, *Chandi* only requires a single pass through the new data points, each running only one range query search and communicating 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 *Chandi* for a group of shared queries is independent of the number of queries in the group (see Section 7). In short, *Chandi* is a full sharing algorithm for arbitrary density-based clustering queries over windowed streams.

Our experimental study (in Section 8) shows that the system using our proposed algorithm *Chandi* comfortably handles 100 arbitrary workload queries under a 1K tuple per second stream 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 independent execution strategies, *IncDBSCAN* [9] and *Extra-N* [2], can only handle less than 1.7 and 12 percent of the same 1K query workload, respectively.

Contributions. The contributions of this work include: 1) We characterize the *growth property* that holds among density-based cluster sets as a general concept enabling multiple clustering query sharing in both dynamic and static environments. 2) We develop a technique called *IntView* that realizes integrated representation for density-based cluster sets identified by multiple queries in the same dataset. 3) We develop a *meta query* strategy as general technique to efficiently execute multiple sliding window queries with varying window parameters. 4) We develop the first algorithm that realizes full sharing for multiple density-based clustering queries over streaming windows. 5) Our comprehensive experiments on several real streaming datasets confirm the effectiveness of our proposed algorithms and also its superiority over all state-of-art alternatives in both CPU time and memory utilization.

2. PROBLEM DEFINITION

Density-Based Clustering in Sliding Windows. We first define the concept of density-based clusters [4]. We use the term *data point* to refer to a multi-dimensional tuple in the data stream. Density-based cluster detection uses a range threshold $\theta^{range} \geq 0$ to define the neighbor relationship (*neighborship*) between any two data points. For two data points p_i and p_j, if the distance between them is no larger than θ^{range}, p_i and p_j are said to be neighbors. We use the function $NumNei(p_i, \theta^{range})$ to denote the number of neighbors a data point p_i has, given the θ^{range} threshold.

Definition 2.1. ***Density-Based Cluster:*** *Given θ^{range} and a count threshold θ^{cnt}, a data point p_i with $NumNei(p_i, \theta^{range}) \geq \theta^{cnt}$ is defined as a core point. Otherwise, if p_i is a neighbor of any core point, p_i is an edge point. p_i is a noise point if it is neither a core point nor an edge point. Two core points c_0 and c_n are connected if they are neighbors of each other, or there exists a sequence of core points $c_0, c_1, ...c_{n-1}, c_n$, where for any i with $0 \leq i \leq n-1$, a pair of core points c_i and c_{i+1} are neighbors of each other. Finally, a density-based cluster is defined as a group of connected core points and the edge points attached to them. Any pair of core points in a cluster are connected with each other.*

Figure 2 shows an example of a density-based cluster composed of

[1] name of a powerful god with multiple hands in hindu theology

11 *core points* (black) and 2 *edge points* (grey) in W_0.

We focus on periodic sliding window semantics as proposed by CQL [10] and widely used in the literature [11, 2]. Such semantics can be either time-based or count-based. For both cases, each query Q has a of 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 template of this query type has been shown in Section 1.

Optimization for Multiple Queries We support multiple density-based clustering queries that are specified to 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. 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. we focus on the generation of complete pattern detection results. In particular, we output the members of each cluster, each associated with a cluster id of the clusters they belong to. Other output formats, such as incremental output, indicating the evolution of the clusters over successive windows, can also be supported by our techniques as discussed in [12].

Our goal is to minimize the overall CPU- and memory- resource consumption for executing all the member queries registered to our system. In particular, given a certain QG and the precondition that all the member queries in QG are accurately answered, we want to minimize both the CPU time for processing each data point and the memory space needed for maintaining the meta information. The techniques regarding dynamic member query adding or removing are not in the scope of this paper, but will be an important part of our future work.

3. EXISTING SINGLE QUERY EXECUTION AND BASIC SHARING STRATEGY

3.1 Extra-N Algorithm

We first briefly describe the best existing approach for single density-based clustering query, called *Extra-N* [2]. *Extra-N* realizes efficient evaluation for single query by incrementally maintaining the cluster structures identified in the query window. Technically, *Extra-N* is based on two main ideas, namely the hybrid neighborship abstraction and the notion of predicted views.

Hybrid (Exact+Abstracted) Neighborship Abstraction. Since density-based cluster structures are defined based on the data points and the neighborships among them, efficiently maintaining the neighborships identified in the windows is naturally the core task for cluster structure maintenance. For each non-core point in the window, *Extra-N* maintains the exact neighborships (a neighbor list containing links to each of its neighbors). For each core point, *Extra-N* maintains the abstracted neighborships for it, including its neighbor count and cluster membership. Such hybrid neighborship abstraction achieves linear memory consumption to the number of data points in the window.

General Notion of Predicted Views. Another problem that needs to be solved for incremental maintenance of density-based clusters 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 clusters. Detecting and handling these changes caused by expirations, especially splitting, may require large amount of computation, which could be as expensive as recomputing the clusters from scratch.

To address this problem, *Extra-N* exploits 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 [13, 2] to process sliding window queries, and is called *Predicted View* technique. Figure 2 shows an example of the predicted views for three future windows. 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

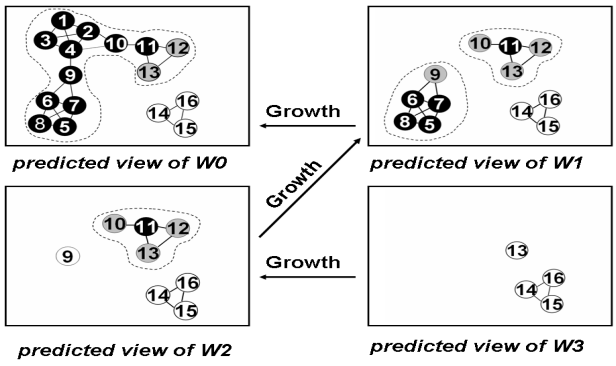

Figure 2: Predicted views of four consecutive windows at W_0

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 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 clusters in the predicted views and form the updated clusters in each window. Figures 2 and 3 respectively demonstrate examples of the "pre-generated" clusters in future windows and the updated clusters after the window slides.

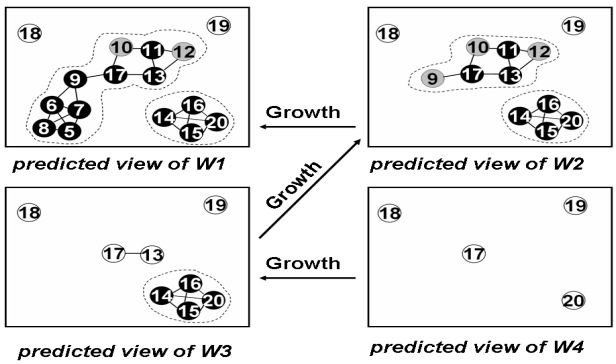

Figure 3: Updated predicted views of four windows at W_1

Discussion Generally, at each window slide, *Extra-N* runs one range query search for each new data point to update the progressive clusters, which are represented by the predicted views. As the best existing algorithm for single query execution, *Extra-N* achieves minimum number of range query searches needed at each window, while keeping the memory consumption linear in the number of data points in the current window. However, executing *Extra-N* 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 optimized processing mechanism for multiple queries to handle a large query group against high speed data streams.

3.2 Sharing Range Query Searches

The basic strategy to share the computations among multiple density-based clustering queries is to share the range query searches. Generally, to execute a query group QG with $|QG| = N$, we can execute N *Extra-N* algorithms, each for a member query, independently (each query maintains its own progressive clusters independently), but share the computations needed by the range query searches. Specifically, at each new query window, *Extra-N* requires every new data point p_{new} to run a range query search to identify its neighbors, and communicate with them to update progressive clusters in the window (as discussed earlier this in section). However, we can always run one instead of $|QG|$ range query searches 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 other queries with smaller and smaller θ^{range}. This is easy to understand, because 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. As an initial step, sharing range query searches can be beneficial, considering the expensiveness of range query searches, especially when the window size is large.

3.3 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 needs to store its progressive clusters independently, the memory space needed by executing a query group QG grows linearly with $|QG|$. 2) Because of the independent cluster storage, the cluster maintenance effort of different queries cannot be shared. To solve these two problem, we need to further analyze and exploit the commonalities of the member queries. Our goal is thus to design an integrated cluster maintenance mechanism, which effectively shares both the storage and computational resources needed by cluster maintenance for multiple queries.

4. "GROWTH PROPERTY" AND HIERARCHICAL CLUSTER STRUCTURES

Growth Property. Now we first introduce an important property of density-based cluster structures which will later be exploited to form the basis for efficient multiple query sharing. In particular, we call this the "Growth Property" of density-based clusters.

Here we first define the concept of "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 each includes a finite number of non-overlapping density-based clusters (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 4.3. *Given Clu_Set1 and Clu_Set2 that Clu_Set2 is a "**growth**" of Clu_Set1, any cluster C_j in Clu_Set2 must be either a **New** cluster (for any $p \in C_j$, $p \notin$ any C_i in Clu_Set1), an **Expansion** of a single cluster in Clu_Set1 (there exists exactly one C_i in Clu_Set1 that $C_i \subset C_j$), or a **Union** of multiple clusters in Clu_Set1 (there exist C_i, C_{i+1},...$C_{i+n} (n > 0)$ in Clu_Set1 with C_j, C_{j+1},...$C_{j+n} \subset C_j$.*

Figure 4 and 5 give an example of two cluster sets between which "growth property" holds.

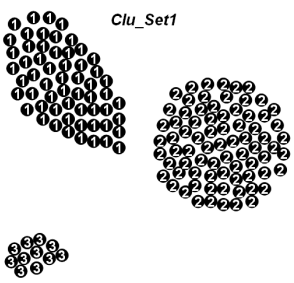

Figure 4: Cluster Set 1 containing 3 clusters

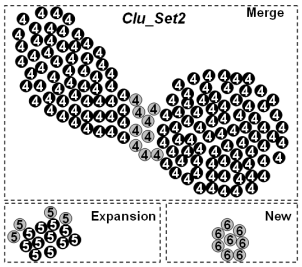

Figure 5: Cluster Set 2 containing 3 clusters, which is a growth of Cluster Set1

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 cluster 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 a same cluster in Clu_Set1 will also belong to a same cluster in Clu_Set2.

Hierarchical Cluster Structure. 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 6 and 7 respectively give examples of independent and hierarchial cluster membership structures built for the two cluster sets shown in Figures 4 and 5.

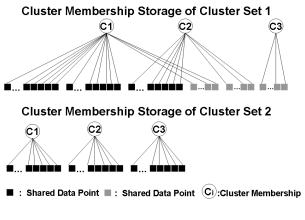

Figure 6: Independent Cluster Membership Storage for Cluster Sets 1 and 2

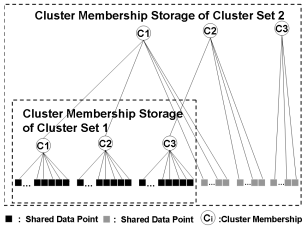

Figure 7: Hierarchial Cluster Membership Storage for Cluster Sets 1 and 2

As shown in Figure 6, 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 7, 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 the 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 with "growth property" transitively holding, the hierarchical cluster structure can largely save the memory space needed for storing them. In particular:

Lemma 4.1. *Given a query group QG having 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.*

The proof of this lemma is straightforward. This is equal to the relationship between the total size of a binary heap and the number of leaf nodes of this heap.

Besides the benefit of huge memory savings, such hierarchical cluster structure can also help us to realize integrated maintenance for multiple cluster sets identified by different queries, and thus save the 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.

5. SHARING FOR QUERIES WITH ARBITRARY PATTERN PARAMETERS

In this section, we discuss the shared processing of multiple queries with arbitrary pattern parameters, namely arbitrary $\theta^{range}s$ and $\theta^{cnt}s$. We first assume that all these queries have the same window parameters, namely same window size win and same slide size $slide$. This assumption will later be relaxed in Section 7 to allow completely arbitrary parameters. In our design for multiple query optimization, we will reeuse the predicted views and hybrid neighborship abstraction techniques (discussed in Section 3) for maintenance of cluster structures. While the key for such design is to exploit the growth property (introduced in Section 4) and thus integrate the clusters identified by multiple queries.

5.1 Same θ^{range}, Arbitrary θ^{cnt} Case.

We first look at the case that all the queries have the same θ^{range} but arbitrary θ^{cnt}. Here, we make a straightforward observation.

Observation 5.1. *Given all the queries in a query group having the same θ^{range}, the neighbors of each data point identified by these queries are same.*

This observation indicates that for all our member queries, the neighborships identified in same the 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 Section 3) requires each none-core point stores 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.

Lemma 5.1. *Given two queries Q_i and Q_j specified to a same dataset, with $Q_i.\theta^{range} = Q_j.\theta^{range}$ and $Q_i.\theta^{cnt} \leq Q_j.\theta^{cnt}$ and querying on the same dataset, the cluster set identified by Q_i is a "growth" of the cluster set identified by Q_j.*

Proof: First, since $Q_i.\theta^{cnt} \leq Q_j.\theta^{cnt}$, the "core point" set identified 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 same, all the "connections" in any cluster structure identified by Q_j will also hold for Q_j. This indicates that the cluster structure identified by Q_j will also be identified by Q_i (although 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 extends 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 4.3). This proves Lemma . ■

Figure 8 demonstrate san example of the cluster sets identified by three queries having a same θ^{range} but different $\theta^{cnt}s$.

Integrated Representation of Predicted Views across Multiple Queries with Arbitrary θ^{cnt}. Since the window parameters of all member queries are same in this case, the predicted windows that need to be maintained by all member queries are the same. In particular, all member queries need to maintain a same number of predicted windows, say from W_0 to W_n. Also, for all member queries, any predicted window W_i (i¡n) has exactly same data points fallen into it. This indicates that, in any predicted window W_i, the cluster sets identified by the member queries will have the growth hold among them (by Lemma 5.1).

As we discussed earlier in Section 4, once the "growth" property holds among the cluster sets, we can build the hierarchical cluster structure for them. Now we build an integrated hierarchical

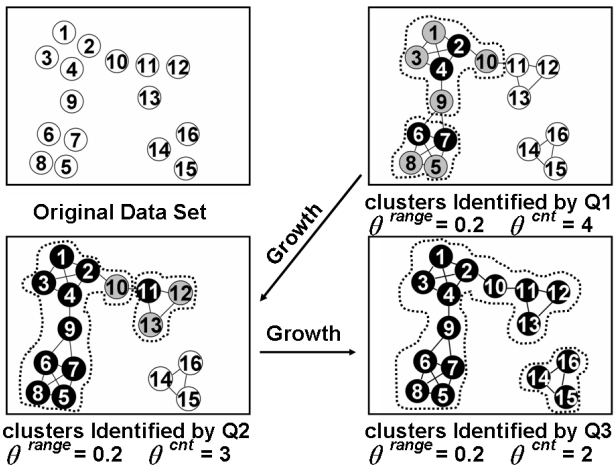

Figure 8: Cluster sets identified by three different queries

structure to represent multiple predicted views identified by different queries for a same corresponding predicted window. We denote 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 9 gives an example of a $IntView\_\theta^{cnt}$, which represents the predicted views (shown in Figure 8) identified by three different queries .

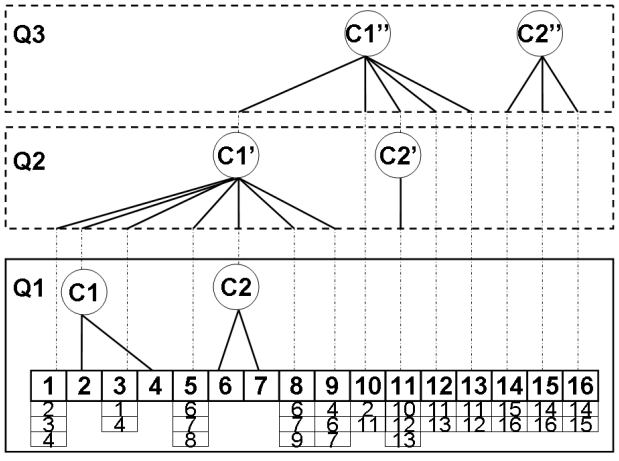

Figure 9: $IntView\_\theta^{cnt}$: Integrated Representation for Predicted Views identified by three different queries

$IntView\_\theta^{cnt}$ successfully integrates the representations of multiple "predicted views" into a single structure, thus saving the memory space from storing them independently. In particular,

Lemma 5.2. *Given the maximum window size allowed, the upper bound of the memory space needed by $IntView\_\theta^{cnt}$ is independent from $|QG|$, the cardinality of the query group represented by it.*

Proof: First of all, 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 to store the cluster memberships for the data points, the upper bound of the memory space used for storing cluster memberships is independent from $|QG|$ (proved in Lemma 4.1. Since $IntView\_\theta^{cnt}$ 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 ($max(Q_i.\theta^{cnt}) - 1$), the upper bound of the memory space used for storing exact neighbors is again independent from $|QG|$. This proves Lemma 5.2. ■

Obviously, 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, however, now makes it independent from $|QG|$ (as proved in Lemma 4.1).

Maintenance of $IntView\_\theta^{cnt}$. Besides the memory saving, we can also update multiple predicted views represented by a $IntView\_\theta^{cnt}$ incrementally and thus save computational resource. In particular, for each new data point p_{new}, we start the updating 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 this new data point to the next higher level predicted views. Using the example that we utilized earlier in Figure 9, a new data point identified to have 3 neighhors in the window, is a "none-core" in the bottum (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, it's 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.

We omit the exact maintenance algorithm of $IntView\_\theta^{cnt}$ here to save space, but emphasize that 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, 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.

5.2 Same θ^{cnt}, Arbitrary θ^{range} Case.

Now we discuss the case that all member queries have the same θ^{cnt} but arbitrary θ^{range}. This case is more complicated than the previous one, because different $\theta^{range}s$ will affect the neighborships identified by different queries. For example, a pair of data points p_i and p_j with distance equal to 0.2 will be considered to be neighbors in Q_1 with $\theta^{range} = 0.1$, while not in Q_2 with $\theta^{range} = 0.4$. As the neighborships identified by different queries are different, the clusters identified by them are likely to be different as well.

Based on our experience of designing $IntView\_\theta^{cnt}$, we explore whether the "growth property" holds between two queries with same θ^{cnt} but different θ^{range}s. Fortunately, the answer is positive as demonstrated below.

Lemma 5.3. *Given two queries Q_i and Q_j specified to a same*

data set, with $Q_i.\theta^{cnt} = Q_j.\theta^{cnt}$ and $Q_i.\theta^{range} \geq Q_j.\theta^{range}$ and querying against the same dataset, the cluster set identified by Q_i is a "growth" of that identified by Q_j.

Proof: First, since $Q_i.\theta^{range} \geq Q_j.\theta^{range}$, for each data point p_i, its neighbors identified by Q_j is a subset of those identified by Q_i. Then, since $Q_i.\theta^{cnt} = Q_j.\theta^{cnt}$, the "core point" set identified by Q_j is a subset of that identified by Q_i. Second, also because of $Q_i.\theta^{range} \geq Q_j.\theta^{range}$, the neighborships identified by Q_j in this data set is a subset of those identified by Q_i. which means that all the "connections" in any cluster structure identified by Q_j will also hold for Q_i. For these two reasons, the cluster structure identified by Q_j will also be identified by Q_i (although 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. Similarly, the "additional connections (neighborships)" among data points can only cause the expansion or merge of clusters identified by Q_j as well. Thus the cluster set identified by Q_i is a "growth" of that identified by Q_j. ■

Figure 10 demonstrates an example of the cluster sets identified by three queries having the same θ^{cnt} but different θ^{range} parameters.

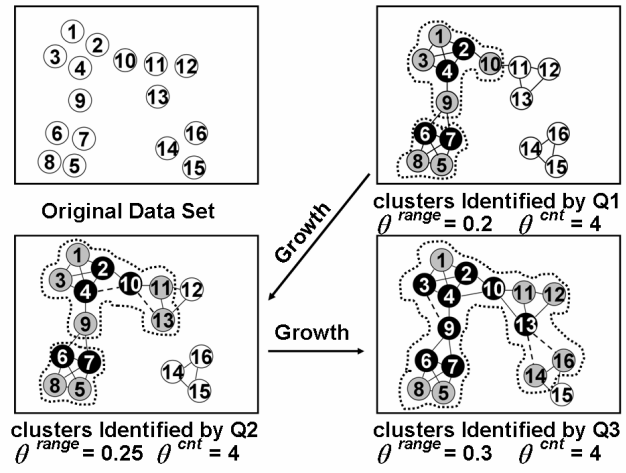

Figure 10: cluster sets identified by three different queries

Integrated Representation of Predicted Views across Multiple Queries with Arbitrary θ^{range}. Similarly as before, we now can build an integrated structure to represent multiple "predicted views" identified by different queries with arbitrary θ^{cnt}. We call it $IntView\_\theta^{range}$.

Similar with $IntView\_\theta^{cnt}$, $IntView\_\theta^{range}$ starts from the "predicted view" with the most "restricted clusters", namely the predicted view representing Q_i with the the smallest θ^{range} among QG in this case. Then, it incrementally stores the cluster "growth information" from one query to the next in the increasing order of θ^{range}. However, as now each data point may be identified to have more "neighbors" in the higher level predicted views, which represent queries with larger and larger θ^{range}, a new type of increment, namely the "additional exact neighbors" of each data point, need to be stored by these predicted views.

Figure 11 gives an example of $IntView\_\theta^{range}$ that integrates the predicted views (shown in Figure 10) identified by different queries. Again, we can prove that the upper bound of the memory space needed by a $IntView\_\theta^{range}$ is independent from $|QG|$. In particular,

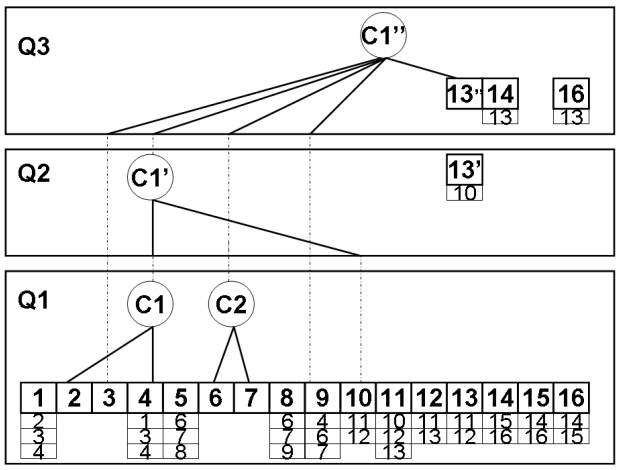

Figure 11: $IntView\_\theta^{cnt}$: Integrated Representation for Predicted Views identified by three different queries

Lemma 5.4. *Given the maximum window size allowed, the upper bound of the memory space needed by $IntView\_\theta^{range}$ is independent from $|QG|$, the cardinality of the query group represented by it.*

Proof: Similar with $IntView\_\theta^{cnt}$, $IntView\_\theta^{range}$ needs to store two types of meta-information, namely the cluster memberships and the exact neighbors of the data points. Since $IntView\_\theta^{range}$ uses the hierarchical structure described in Section **??** to store the cluster memberships for the data points, the upper bound of the memory space used for storing cluster memberships is independent from $|QG|$ (proved in Lemma 4.1. Although $IntView\_\theta^{range}$ needs to incrementally store "additional exact neighbors" for the data point in higher level predicted views, the maximum number of exact neighbors a "non-core" point can have is still a constant ($\theta^{cnt} - 1$). The upper bound of the memory space used for storing exact neighbors is thus independent from $|QG|$. This proves Lemma 5.4. ■

Since $IntView\_\theta^{range}$ is very similar in concept with $IntView\_\theta^{cnt}$, we omit the details of $IntView\_\theta^{range}$ maintenance.

5.3 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 are different. So, to again take the advantage of the compact structure of Integrated Representation of Predicted Views, we need to explore that when does "growth property" hold between two queries in the most general cases.

Lemma 5.5. *Given two queries Q_i and Q_j specified to a same dataset, with $Q_i.\theta^{cnt} \leq Q_j.\theta^{cnt}$ and $Q_i.\theta^{range} \geq Q_j.\theta^{range}$ and querying against the same dataset, the cluster set identified by Q_i is a "growth" of that identified by Q_j.*

Proof: Lemma 5.5 can be proved 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 5.1). This

means that for any cluster C_a identified by Q_j there exist a cluster C_b identified by Q_j that $C_a \in C_b$. Also, the cluster set identified by Q_i is a "growth" of that identified by Q_k (by Lemma 5.3). This means that for any cluster C_b identified by Q_j there exist a cluster C_c identified by Q_i that $C_b \in C_c$. So for any cluster C_a identified by Q_j there exist a cluster C_c identified by Q_i that $C_a \in C_c$. Thus, the cluster set identified 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 follow definition.

Definition 5.2. *Given two queries Q_i and Q_j specified to a 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 again aim to build a single structure which represents the "predicted views" identified by all member queries of QG in a 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 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 indicates 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 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" remained 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 (neither of them is more restricted than the other). 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 calculating the "skyline" 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 (before the real-time execution), 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 the 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 demonstrated below.

Lemma 5.6. *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_n 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 by 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 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 least "growth information" from the cluster set identified by itself to that identified by Q_n. Based on our analysis, for two member queries, their difference on "neighborships" identified is more likely to cause the difference on the cluster sets identified by them, compared to their difference on the requirement for "core points". 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 larger influence to 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 "weights" are given to that between $\theta^{range}s$. The specific algorithm is omitted here. To unify the names of the hierarchical structures representing multiple predicted views, we henceforth call the "Predicted View Tree" Int_θ.

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 most similar predicted view 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 5.7. *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 proved using the same method as we used for proving Lemmas 5.2 and 5.4.

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 higher level of $IntView\_\theta$.

Now we conclude the contribution of $IntView\_\theta$ as demonstrated below.

Theorem 5.3. *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 on Int_θ only store the increments from its "parent", no duplicate information is ever stored among any two predicted views. This proves that $IntView\_\theta$ achieves full sharing on 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 tree structure, no matter how many different predicted views their neighborship appears in. This proves that $IntView\_\theta$ achieves full sharing on computation of multiple queries. ■

5.4 $IntView\_\theta$ In Multiple Predicted Windows

Given the assumption that all the member queries in QG share the window parameters, namely the same win and $slide$, we have discussed earlier in this section that the predicted windows that need to be maintained by all member queries are the same. So, a most straightforward way to serve a query group that need to maintain N predicted windows, is to use N $IntView_\theta$s to represent N predicted windows independently. Using this method, the N $IntView_\theta$s presenting different predicted windows will have the same tree structure, as they are representing the same queries in each predicted window. Figure 12 gives an example of using four N $IntView_\theta$s to represent four predicted windows for a query group with 5 member queries.

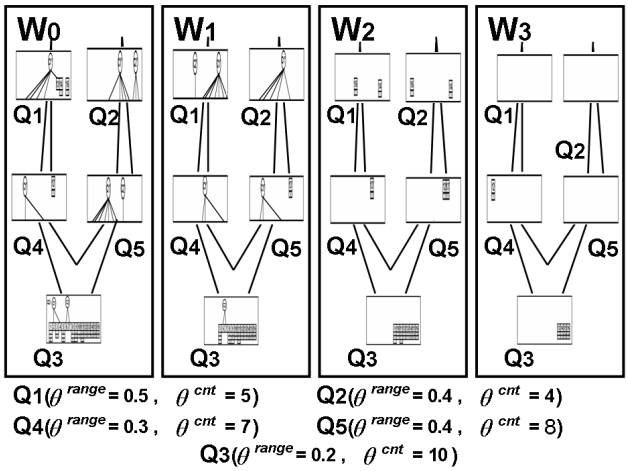

Figure 12: $IntView\_\theta$: Integrated Representation for Predicted Views identified by five different queries with arbitrary pattern parameters

As conclusion, for the general case of handling a query group with arbitrary pattern parameters but the same window parameters, we employ N ($N > 0$) $IntView_\theta$s to represent the predicted views identified by multiple queries in multiple windows, and maintain them independently at arrival of each new data point. We note that such straightforward application of $IntView\_\theta$ realizes the full sharing in each predicted window (Lemma 5.3), but no sharing is yet achieved across the different predicted windows, as they still need to be maintained independently. A more sophisticated hierarchical structure integrating multiple $IntView_\theta$s will later be introduced next to further realize full sharing across multiple predicted windows.

6. SHARING FOR QUERIES WITH DIFFERENT WINDOW PARAMETERS

In this section, we discuss memory and CPU sharing among multiple queries with different window parameters, namely variations in the window size win and the slide size $slide$. During the discussion, we assume that all these queries have the same pattern parameters.

6.1 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. So that 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 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 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 time, each serving a future a 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 of 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 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 does not require any substantial change to "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 that at which moments the member queries need to be answered. Also, the "distance" between

any two successive output moment is changing periodically. So, we can always make an output schedule (with a finite 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 correspond 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 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 need to maintain their own predicted windows starting at 00:00:00 and 00:00:06 separately, although they are exactly same with those maintained by Q_1. In this example, 33 percent of "predicted windows" are saved from the independent maintenance mechanism, which means 33 percent of storage space and computational resources are saved. Such "predicted windows" for a meta query are no different from those needed for any single query. So, a straightforward way to maintain them is to use the maintenance method introduced in *Extra-N* [] to update them independently at the arrival of each new data point.

In conclusion, by building a meta query representing all member queries in a query group, we can save both the memory space and CPU 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 (all the predicted windows built in our integrated solution need to be maintained by individual member queries any ways). 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.

6.2 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 use 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 characteristics can be discovered for such query groups.

Lemma 6.1. *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, which all overlap with those built by Q_3. At this moment, the "predicted window" starting from 00:00:00 can 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 6.1 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.

6.3 Arbitrary $slide$, Arbitrary win Case

We now give the solution for the cases 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 properly as the queries required) 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 13 shows the predicted views that need to be maintained by each of these three queries independently, versus those by the meta query at wall clock time 00:00:10.

Figure here:

Integrated Representation of Predicted Views across Multiple Windows. Although *Extra-N* [2] algorithm can be applied to maintain the predicted views of the meta query and thus answer the whole query group, this algorithm achieves no sharing across the multiple predicted windows, because it requires the predicted view of each predicted window to be stored and maintained independently (we mentioned this at the end of Section 5 as well). Now we introduce a further optimization for the maintenance for the predicted views of multiple predicted windows based on the "growth property". We first make the following observation.

Lemma 6.2. *For a single query Q_i, at any given time T with W_n being the current window, the cluster set identified in the "pre-*

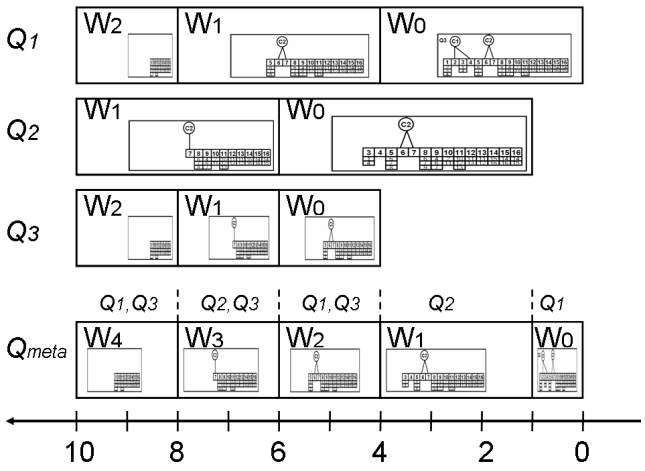

Figure 13: Predicted views maintained by three queries independently versus those maintained by a single meta query

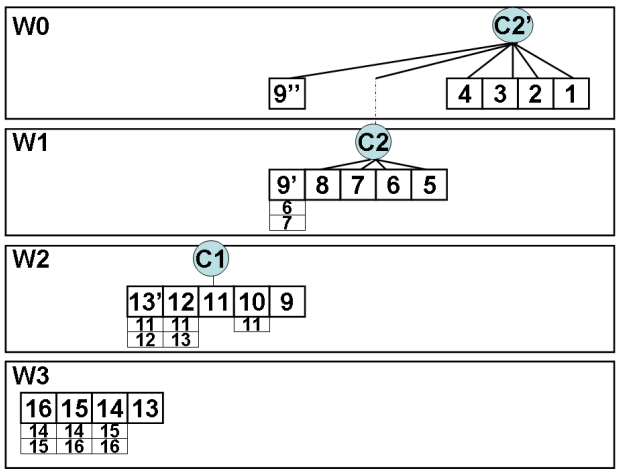

Figure 14: $IntView\_W$: Integrated Representation for Predicted Views Identified by a Single Query in 4 Predicted Windows

*dicted view" of predicted window W_{n+i} is always a **growth** of that identified in the "predicted view" of W_{n+i+1}.*

Proof: At any given time T with W_n being the current window, it is known that more and more data points in W_n will expire as the window number increases. So the data points falling into W_n that appear in the "predicted view" of W_{n+i+1} are a subset of those appear in "predicted view" of W_{n+i}. This is equal to saying that the "predicted view" of W_{n+i} is composed by adding data points to the "predicted view" of W_{n+i+1}. As we mentioned earlier in Section 3 and demonstrated in [2], birth, expansion and union are the only possible pattern changes caused by addition of data points to a window. So, based on observation 4.3, the cluster set identified in W_{n+i} is a "growth" of cluster set identified in W_{n+i+1} This proves Lemma 6.3. ■

Note that Lemma abstracts the change of clusters from one window to another in a reversed direction with the real sequence of the windows. Cluster sets shown in Figure 2 can be taken as an example where such "growth property" holds. In particular, the "growth property" transitively holds from the cluster sets identified in predicted window W_3 to W_0. Again, the "growth property" allows us to build an integrated structure to incrementally store and maintain the predicted views across multiple windows. We call such integrated representation of predicted views across multiple windows $IntView_W$.

Figure 14 gives an example of the $IntView_W$ built for the predicted views showing Figure 2

In particular, since the data points covered by different predicted windows incrementally increase as the window number decrease, another new type of incremental information, namely the "addition data points", need to be maintained at each higher level predicted views. As a hierarchical structure that are very similar with $IntView_\theta^{cnt}$ and $IntView_\theta^{range}$, the predicted views in $IntView_W$ can also be incrementally maintained at the arrival of each new data point, by a very similar maintenance method with them. We omit the details of the maintenance method here.

As conclusion, now we have an even more efficient way to handle a query group QG with same pattern parameters but arbitrary window parameters. For such query group QG, we can always build a meta query for it which carries all the predicted windows needed by its member queries, and then we can use $IntView_W$ to integrate them into a single $IntView\_W$ structure.

Figure 15 shows the improved predicted view storage mechanism for the meta query by using $IntView\_W$.

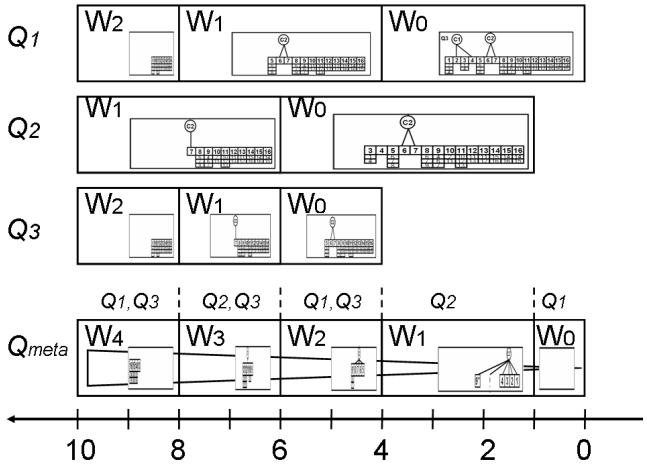

Figure 15: $IntView\_W$: Integrated Representation for Predicted Views Identified by a Meta Query in 5 Predicted Windows

7. PUTTING 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 have to solve it from scratch, we now can easily handle it by combine the two techniques introduced in last two sections, namely the the $IntView\_\theta$ technique and the meta query technique. These two techniques are orthogonal to each other, and can thus be easily combined. In particular, the $IntView_\theta cnt$ technique (introduced in Section 5) is design to share among a group of queries that are specified to a same dataset, which in our case is each predicted windows. So, we can consider this here as an

"**inner-predicted-windows**" sharing technique. On the other hand, the meta query technique (introduced in Section 6) is designed to make sure the predicted windows, which need 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, such two orthogonal techniques can be easily applied together to realize the full sharing of the member queries on both 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 same with those shown in Figure 15. Then, for each predicted window built, we apply $IntView\_\theta$ 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 are serving Q_1 and Q_3, we build a "Predicted View Tree" for it 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 16. 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 is corresponding to an output time point for it. Using the same example, Q_2 has predicted views in W_4, as W_4 is not a predicted window that need to be maintained by it.

To apply the $IntView\_W$ technique (introduced in Section 6), which allows us to share across multiple predicted windows, we use "the most restricted query" of the whole query group to act as the root of all the "Predicted View Trees" built in different windows. Using the same example in the last paragraph, the roots of all "Predicted View Trees" will be the predicted view for Q_1 having $\theta^{cnt} = 0.2$ and $\theta^{cnt} = 5$. By doing so, the "Predicted View Trees" in different predicted now start from the predicted view representing a same query. Thus, we can further integrate these roots in different predicted windows into a $IntView\_W$ structure. This final move "connects" all the "Predicted View Trees", forming a single hierarchical structure that realizes completely incremental storage and maintenance for all member queries across multiple predicted windows. We call such ultimate hierarchical structure $IntView$. Figure 16 demonstrates the $IntView$ 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 largest window number). As the "backbone" of $IntView$, an $IntView_W$ structure connects the roots of all "Predicted View Trees" ($IntView\_\theta$) in different predicted windows. 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 built based on the incremental information from the next (the newer) window (as its root itself now is incremental). We call this final solution Integrated maintenance for density-based clustering "*Chandi*". We give the psuedo-code of *Chandi* below.

As shown our psuedo-code (in Figure 17), when a new data point arrives at the system, it first runs a range query search using the largest θ^{range} among the query group to collect all its potential neighbors, and then it distributes each of them to the first predicted view on each path of $IntView$, in which their "neighborship" truely exist. Then, it starts the $IntView$ maintenance process from the root of $IntView$, namely the root predicted view of the newest predicted window on $IntView$, and then incrementally maintain those at higher level of $IntView$. During the maintenance for of each predicted view, it only needed communicates with the neighbors distributed to that particular view. Thus, computation-wise, *Chandi* only requires a single pass through the new data points, each running one range query search and communicating with its neighbors once for all shared queries (on each path). Memory-wise, as the growth property holds among the cluster sets identified by the queries on each path of $IntView$, the upper bound of the memory consumption of *Chandi* 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 proved using the same methods as we used for proving Lemma 5.7). In conclusion, *Chandi* achieves full sharing for multiple density-based cluster queries over the same input stream in terms of both CPU and memory resources.

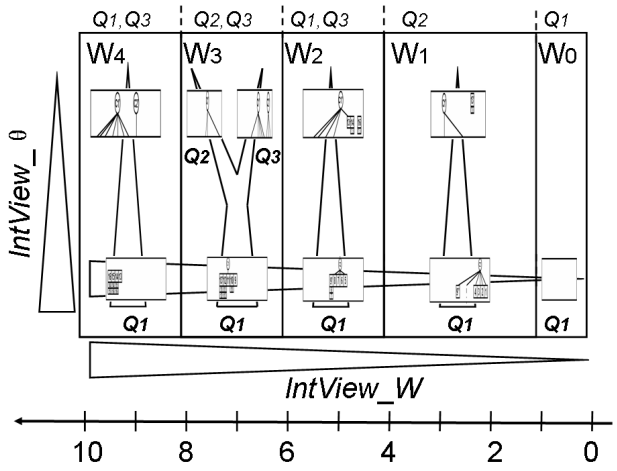

Figure 16: $IntView$: Integrated Representation for Predicted Views Identified by 3 Queries in 5 Predicted Windows

8. EXPERIMENTAL STUDY

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

We used two real streaming data sets. The first data set, GMTI (Ground Moving Target Indicator) data [6], records the 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 [14], which has one millions transaction records throughout the trading hours of a day.

Competitor Algorithms and Experimental Methodologies. To evaluate our proposed algorithm *Chandi*, for any input QG with $|QG| = N$, we compare *Chandi*'s performance of executing QG with four alternative methods. First, *Extra-N* algorithm with no range query sharing (henthforth referred as *Extra-N*), indicating that we run N *Extra-N* algorithms, each for a member query in

p_i: a data point. p_{new}: a new data point.
clu_mem : cluster membership.
W_i : a predicted window.
$IntView$: the overall IntView structure.
$W_{oldest/newest}$: oldest/newest W on $IntView$.
W_{oldest} : the newest predicted window on $IntView$.
$W_i.root$: the root predicted view of in W_i.
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}
// purge
2 **if** $p_{new}.T > W_{oldest}.T_{end}$
3 Purge(W_{oldest}); //purge the oldest predicted window
// load
4 load p_{new} into index
// IntView Maintenance
5 $neighbors$:=RangeQuerySearch($p_{new}, max(Q_i.\theta^{range})$)
6 UpdateIntView $(p_{new}, neighbors)$
// output
7 **if** $p_{new}.T = T_{output}$
8 Output();
9 add new window W_newest to $IntView$

Purge(W_i**)**
1 purge any p_i from index if
$W_i.T_{start} \leq p_i.T < W_{i+1}.T_{start}$
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($p, W_{newest}.root$);

To put a neighbor of the new data point to the predicted views where it needs to be processed. It guarantees that each neighbor only appears once on each path of $IntView$.
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($p, Q_j.PV$);

Figure 17: ***Chandi*** **Algorithm**

$|QG|$, independently. Second, *Extra-N* algorithm with range query sharing (referred as *Extra-N with rqs*), indicating that we run N *Extra-N* algorithms independently but share the computation consumed by range query searches among them. Third, Incremental DBSCAN (referred as *IncDBSCAN*), indicating that we run N Incremental DBSCAN algorithms independently. Fourth, Incremental DBSCAN with range query sharing (referred as *IncDBSCAN with rqs*), indicating that we run N *Extra-N* algorithms independently but share the range query searches among. We note that, for different queries, during the purging process, the data points that are required to run range query searches by Inc DBSCAN are different. Sharing range query searches for Inc DBSCAN during purging is not a trivial problem and not discussed in literature. Thus here we share the range query searches for the new data point in each window only.

Our goal is to evaluate the performance of these five algorithms when executing a query group specified to a same input stream. We measure two common metrics for stream processing algorithms, namely average processing time (for each tuple) and memory footprint. We run all the experiments using real data to the end of the datasets. The average processing time is averaged over all the tuples in each experiment. The memory footprint, which indicates the maximum memory space required by an algorithm, is recorded over all the windows.

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 other three parameters for all of them. The parameter settings in our experiment are learned from the pre-analysis to 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 to 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 vary 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 the Figures 18 and 19, both the average processing time and the memory space used by all four 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 ultilization 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 consump-

tion is almost equal to *IncDBSCAN* and much lower than *Extra-N*. This matches with our analysis in Section 5, 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" exsiting 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 number 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.

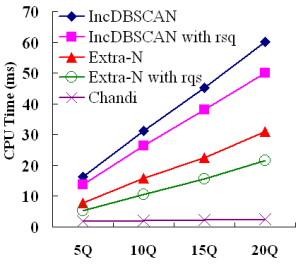

Figure 18: CPU time used by five competitors in arbitrary θ^{cnt} cases

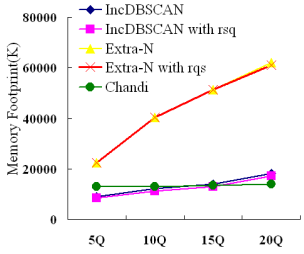

Figure 19: Memory space used by five competitors in arbitrary θ^{cnt} cases

Arbitrary θ^{range} ***case.*** In this case, we use $\theta^{cnt} = 10$, $win = 5000$ and $slide = 1000$, while vary 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 20 and 21, 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 because of two main reasons. 1) Since the θ^{range} parameters vary among the queries, the range query search cost increases along with 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*, are 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 rely on the cost of range query searches.

Arbitrary win ***case.*** In this case, we use $\theta^{cnt} = 10$, $\theta^{range} = 0.01$, $slide = 500$, while vary win from 1000 to 5000 (we use 500 as granularity for any window parameter). As shown in Figures 28 and 29, 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 from the number of queries in the query group.

Arbitrary $slide$ ***case.*** In this case, we use $\theta^{cnt} = 10$, $\theta^{range} = 0.01$, $window = 5000$, while vary $slide$ from 500 to 5000. As shown in Figures 24 and 25, 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 needs 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.

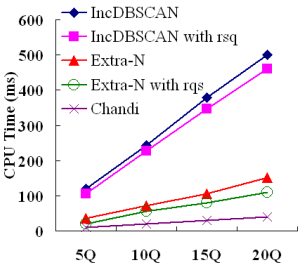

Figure 20: CPU time used by five competitors in arbitrary θ^{range} cases

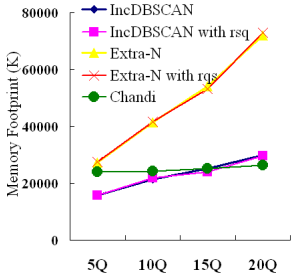

Figure 21: Memory space used by five competitors in arbitrary θ^{range} cases

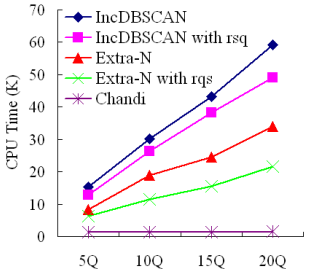

Figure 22: CPU time used by five competitors in arbitrary win cases

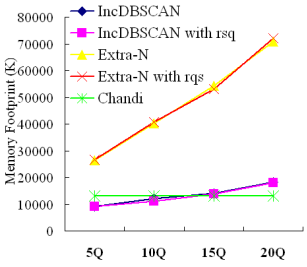

Figure 23: Memory space used by five competitors in arbitrary win cases

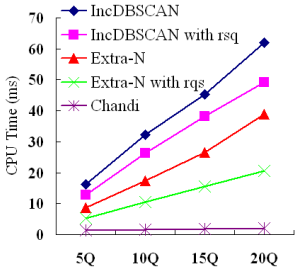

Figure 24: CPU time used by five competitors in arbitrary $slide$ cases

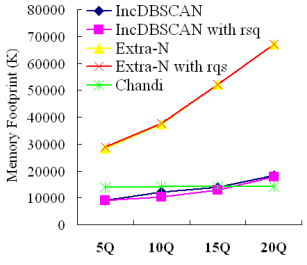

Figure 25: Memory space used by five competitors in arbitrary $slide$ cases

Evaluation for Two-Arbitrary-Parameter Cases We evaluate the 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 a 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 Fig-

ures 26 and 27, *Chandi* still consumes significantly less CPU time compared with other alternatives, although the increase of CPU consumption corresponding to the increase of member queries in more obvious. This is because totally arbitrary pattern parameters leads to even larger difference on the cluster identified by different queries, and thus increase the maintenance cost 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 the number 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 slightly higher than *IncDBSCAN*. Again, this is the caused by 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.

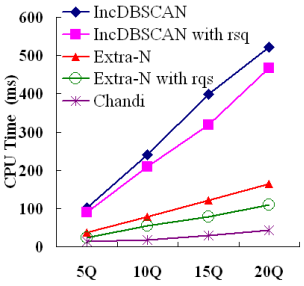

Figure 26: CPU time used by five competitors in arbitrary pattern parameter cases

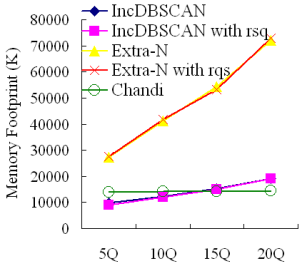

Figure 27: Memory space used by five competitors in arbitrary pattern parameter cases

Arbitrary Window Parameters. In this case, we use $\theta^{cnt} = 10$ and $\theta^{range} = 0.01$, while varying win from 1000 to 5000 $slide$ from 500 to 5000 (for any particular query Q_i, $Q_i.slide < Q_i.win$). As shown in Figures 28 and 29, the performance of *Chandi* is similar with that is observed from the arbitrary win or $slide$ case. This is because, although the member 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 an query group still only depends on the largest win in the query group and the number of predicted views that need to be maintained, which both do not necessarily increase along with the number of queries.

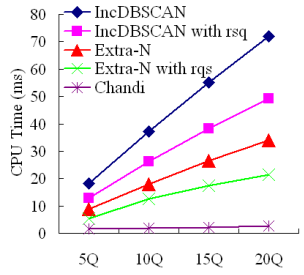

Figure 28: CPU time used by five competitors in arbitrary window parameter cases

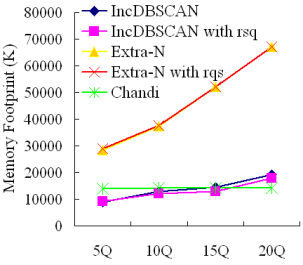

Figure 29: Memory space used by five competitors arbitrary window parameter cases

General Case: Four Arbitrary Parameters. Finally, we evaluate the general case, with all four parameters arbitrary. We divide this experiment into three cases, each measuring the performance of the algorithms when executing different number 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 $slide = 500$ 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 overall average cost of each algorithm, and measure the cost caused by each specific subtask. In particular, for the CPU measurement is divided it 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 the memory used by raw data (for storing actual tuples) and the memory used for meta-data.

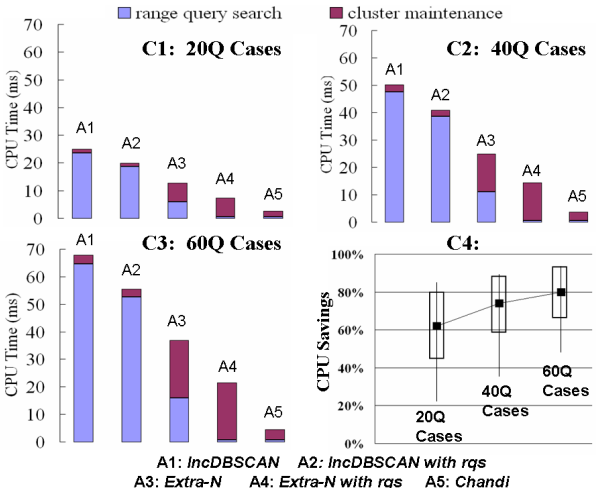

Figure 30: Detailed comparison on CPU time consumption of five algorithms

As shown in C1, C2 and C3 of Figure 30, 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 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) utilizations 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 30, *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* on CPU time are 62%. Although the minimum savings in this case among the 30 groups is 23%, the maximum savings reaches 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 saving achieved by *Chandi* on 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 query group in this test case, 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 $slide = 500$ to 5000.

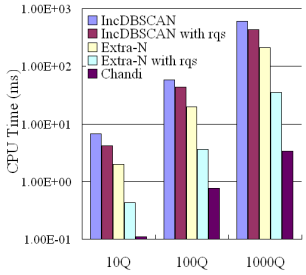

Figure 31: CPU time used by five competitors in logarithmic scale

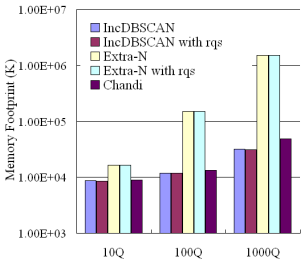

Figure 32: Memory space used by five competitors in logarithmic scale

As shown in Figures 31 and 32, both the CPU time and the memory space used by *Chandi* increases 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 0.76 and 3.3ms respectively, which indicates that our system can comfortably handle 100 queries under a 1000 tuple per second data rate, and handle 1000 queries under a 300 tuple 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* are more efficient than other alternative methods in terms of both CPU and memory utilization when executing multiple queries specified to a same input stream. *Chandi* achieves most sharing when only one of the four parameters are different 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 other alternative methods by achieving on average 60 percent saving for CPU time and 84 percent saving for memory space. Lastly, *Chandi* shows a good scalability in terms of handling a large number (hundreds or even thousands) of queries under high data rate.

9. RELATED WORK

Traditionally, pattern detection techniques, such as cluster [15, 4, 16] and outlier detection [17, 18], are designed for static environments with large volumes of stored data. More recently, as stream applications are becoming prevalent, the problem of efficient pattern detection in the streaming context is being tackled. Previous work for streaming data clustering include [19, 2, 20], and for detecting outliers include [21, 22].

In this work, our target pattern type is a well known pattern type, namely density-based clusters first proposed in [4] as DBSCAN algorithm for static data. Later an Incremental DBSCAN [9] algorithm was introduced to incrementally updates density-based clusters in data warehouse environments. However, as both analytically and experimentally shown in [2], since all optimizations in [9] 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 8, also demonstrates that executing multiple queries using [9] in streaming environment is prohibitively expensive in terms of CPU resource consumption.

Algorithms on density-based clustering queries over streaming data include [2, 1, 3]. Among these works, [1] and [3] 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. [2] is the only algorithm we are aware of that detects density-based clusters in sliding windows.

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 [23, 24, 25] 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 stated information as well as the query processing computation with a single operator, when multiple queries have similar yet not identical operator specifications For example, two queries may calculate aggregations for a same input stream but using different window sizes. The problem we solve this paper fall into the operator level sharing category.

Previous research effort discussing such operator level sharing techniques focus on operators, such as selection and join operators [26, 7, 8, 27], and aggregation operators [28, 11]. To our best knowledge, none of them discuss the sharing for clustering operators. Some general principles used in these works, such as query containment [7] 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 the 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 the interrelationships among them, 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 while has not been studied in any of these works.

10. CONCLUSION

In this work, we are the first to present a framework, called *Chandi*, for efficient shared processing of a large number of density-based clustering queries over streaming windows. For answering multiple such queries with arbitrary parameter settings, *Chandi* achieves full sharing on both CPU and memory utilizations. Our experimental study shows that, for the most general cases, *Chandi* is on average four times faster than the best alternative while using 85% less memory space. More savings can be achieved if the queries share more common parameter settings. *Chandi* also shows very good scalability in terms of handling large numbers of queries under high speed input streams in our experiments.

11. REFERENCES

[1] Y. Chen and L. Tu, "Density-based clustering for real-time stream data," in *KDD*, 2007, pp. 133–142.

[2] D. Yang, E. A.Rundensteiner, and M. O. Ward, "Neighbor-based pattern detection for windows over streaming data." *EDBT*, 2009, to be appear.

[3] F. Cao, M. Ester, W. Qian, and A. Zhou, "Density-based clustering over an evolving data stream with noise," in *SDM*, 2006.

[4] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise." in *KDD*, 1996, pp. 226–231.

[5] J. A. Hartigan and M. A. Wong, "A k-means clustering algorithm," *Applied Statistics*, vol. 28, no. 1.

[6] J. N. Entzminger, C. A. Fowler, and W. J. Kenneally, "Jointstars and gmti: Past, present and future," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 35, no. 2, pp. 748–762, april 1999.

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

[8] S. Krishnamurthy, M. J. Franklin, J. M. Hellerstein, and G. Jacobson, "The case for precision sharing," in *VLDB*, 2004, pp. 972–986.

[9] M. Ester, H.-P. Kriegel, J. Sander, M. Wimmer, and X. Xu, "Incremental clustering for mining in a data warehousing environment," in *VLDB*, 1998, pp. 323–333.

[10] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: semantic foundations and query execution," *VLDB J.*, vol. 15, no. 2, pp. 121–142, 2006.

[11] A. Arasu and J. Widom, "Resource sharing in continuous sliding-window aggregates," in *VLDB*, 2004, pp. 336–347.

[12] D. Yang, "Neighbor-based pattern detection for windows over streaming data." *WPI Technical Report*, 2008, $http://users.wpi.edu/\ diyang/str\_patt\_detect.pdf$.

[13] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "No pane, no gain: efficient evaluation of sliding-window aggregates over data streams," *SIGMOD Record*, vol. 34, no. 1, pp. 39–44, 2005.

[14] I. INETATS, "Stock trade traces. http://www.inetats.com/."

[15] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: an efficient data clustering method for very large databases," *SIGMOD Record, vol.25(2), p. 103-14*, 1996.

[16] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, "Optics: Ordering points to identify the clustering structure," in *SIGMOD*.

[17] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "Lof: identifying density-based local outliers," *SIGMOD Rec.*, vol. 29, no. 2, pp. 93–104, 2000.

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

[19] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, "A framework for clustering evolving data streams." in *VLDB*, 2003, pp. 81–92.

[20] B. Babcock, M. Datar, R. Motwani, and L. O'Callaghan, "Maintaining variance and k-medians over data stream windows," in *PODS*, 2003, pp. 234–243.

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

[22] F. Angiulli and F. Fassetti, "Detecting distance-based outliers in streams of data," in *CIKM*, 2007, pp. 811–820.

[23] Z. Liu, S. Parthasarathy, A. Ranganathan, and H. Yang, "Near-optimal algorithms for shared filter evaluation in data stream systems," in *SIGMOD Conference*, 2008, pp. 133–146.

[24] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, "Niagaracq: A scalable continuous query system for internet databases," in *SIGMOD Conference*, 2000, pp. 379–390.

[25] J. Chen, D. J. DeWitt, and J. F. Naughton, "Design and evaluation of alternative selection placement strategies in optimizing continuous queries," in *ICDE*, 2002, pp. 345–356.

[26] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman, "Continuously adaptive continuous queries over streams," in *SIGMOD Conference*, 2002, pp. 49–60.

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

[28] S. Krishnamurthy, C. Wu, and M. J. Franklin, "On-the-fly sharing for streamed aggregation," in *SIGMOD Conference*, 2006, pp. 623–634.