A Shared Execution Strategy for Multiple Pattern Mining Requests over Streaming Data *

Di Yang
Worcester Polytechnic Institute
Computer Science Department
diyang@wpi.edu

Elke A. Rundensteiner
Worcester Polytechnic Institute
Computer Science Department
rundenst@cs.wpi.edu

Matthew O. Ward
Worcester Polytechnic Institute
Computer Science Department
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 may submit similar pattern mining requests, such as cluster detection queries, yet customized with different parameter settings. In this work, we present an efficient shared execution strategy for processing 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, including both pattern-specific and window-specific parameters. We introduce the notion of the *growth property* among the cluster sets identified by different queries, and characterize the conditions under which it holds. By exploiting this *growth property* we propose a uniform solution, called *Chandi*, which represents 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

*This work is supported under NSF grants CCF-0811510, IIS-0119276 and IIS-00414380. We thank our collaborators at MITRE Corporation, Jennifer Casper and Peter Leveille, for the GMTI data stream generator

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

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. Previous research effort that have developed efficient algorithms for streaming pattern detection focused on single query processing [5, 6, 18]. Little effort has been made towards simultaneous execution of multiple pattern mining queries against the same input stream. In this work, we provide a strategy 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 dropped or rose significantly in the most recent transactions can be considered as a parameterized query. Here, analysts need 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 [9] that require a range and count threshold as input, and K-means style clustering [11] 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, different 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 a priori the most effective input parameters is difficult when faced with an unknown input stream. 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.

In this work, we tackle multiple query optimization for density-based clustering queries over sliding windows. Such query type is of relevance to many important applications, such as monitoring congestions (clusters) in traffic from the streams reporting vehicle positions, and learning "transaction-intensive areas" (clusters) from the most recent stock transactions. In these applications, sliding window semantics need to be applied to force the clusters to form based on the most recent portion of the input streams only. The

out-of-date information, such as the positions of vehicles reported a long time ago, should no longer contribute to the recent clustering results and thus has to be purged from the window.

Challenges. Each density-based clustering query over sliding windows 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.

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 a 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. For instance, figure 8 shows different clusters identified in a subpart of the GMTI [7] data stream 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. 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, such as join or aggregation operators, usually maintain pair-wise relations between two tuples (independent from the rest of the tuples) or simply numbers (aggregation results). The techniques [10, 13] regarding sharing among SQL queries are not adequate to solve our problem.

Proposed Solution. Our proposed solution allows arbitrary parameter settings for queries on all four input parameters. 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, the *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 an 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 from maintaining them independently.

We also propose a "*meta query* strategy", which uses a single meta query to represent all the workload queries, when their pattern parameters are the same. The proposed meta query strategy adopts a flexible window management mechanism to efficiently organize the query windows that need to be maintained by multiple queries. By leveraging the overlap among query windows, it minimizes the number of windows that 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). Thus, *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 data rate. If the number of workload queries increases to 1K, the system still works stably with a 300 tuple per second input rate. On the same experimental platform, given the 300 tuples per second input rate, the independent execution strategies from the literature, such as *IncDBSCAN* [8] and *Extra-N* [18], 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 the density-based cluster sets as a general concept enabling multiple clustering query sharing in both dynamic and static environments. 2) We introduce a technique called *IntView* that realizes integrated maintenance for the 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 propose 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 [9, 8]. 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*

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

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

We focus on periodic sliding window semantics as proposed by CQL [2] and widely used in the literature [18, 3]. 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. Queries over the same input stream can have arbitrary settings on all four 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 focus on the generation of complete clustering results. In particular, we output the members of each cluster, each associated with the cluster id of the cluster they belong to. Given the precondition that all the member queries are accurately answered, our goal is to minimize both the average processing time for each data point and the memory space needed by the system.

3. EXISTING SINGLE QUERY EXECUTION AND BASIC SHARING STRATEGY

3.1 Extra-N Algorithm

Alternative methods for processing a single density-based clustering query over sliding windows are discussed in [18]. Both analytical and experimental studies conducted in [18] show that *Extra-N* is the best existing approach for executing a single query of this type. *Extra-N* realizes efficient evaluation 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 neighborships among data points, 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.

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, such as splitting of clusters. Detecting and handling these changes 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 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 (refered as "predicted windows") based on these known-to-participate data points and thus form the "predicted views" for them. Figure 2 shows an example of the predicted views for three future windows. In this example, the window size and the slide size of the query are 16 tuples and 4 tuples respectively. 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

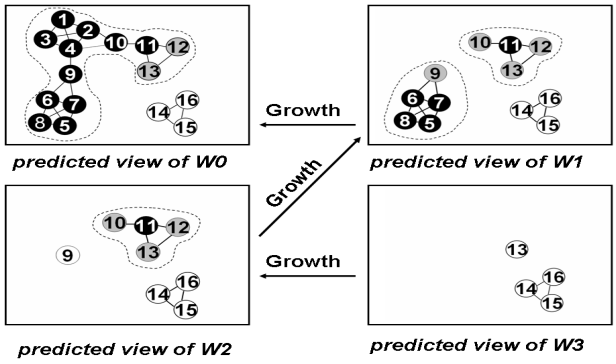

Figure 2: Predicted views of four consecutive windows at W_0

view technique, we can avoid the computational effort needed for discounting the effect of such expired data points from the detected clusters. The idea is to pre-generate the partial clusters for the future windows based on the data points that are in the current 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. Figures 2 and 3 respectively demonstrate examples of the "pre-generated" clusters in future windows and the updated clusters after the window slides.

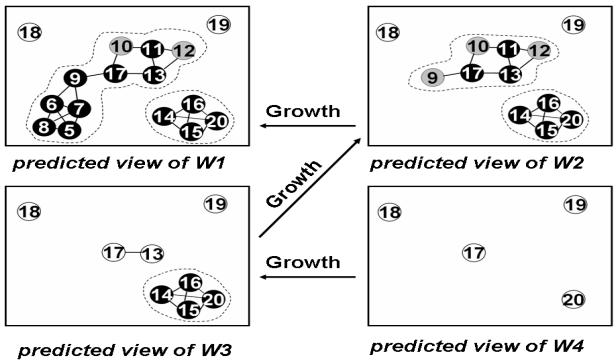

Figure 3: Updated predicted views of four windows at W_1

Discussion. 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. Although *Extra-N* is an effective solution for single query execution, executing *Extra-N* algorithm for each member query independently is not scalable for handling a QG with large $|QG|$. We thus aim to design a shared processing strategy 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, one for each member query, independently (each maintaining its own progressive clusters), but share the range query searches. Specifically, for each new data point p_{new}, we run one instead of $|QG|$ range query searches, with $Q_i.\theta^{range}$ the largest θ^{range} in QG. Using the result set of this "broadest" range query search, we then gradually filter out the results for the queries with smaller and smaller θ^{range}. Since the range query search with the largest θ^{range} is in any case needed by at least one query, no extra computation is introduced by this process. Considering the expensiveness of range query searches, sharing them can be beneficial, especially for large windows.

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 needs to store its progressive clusters independently, the memory space for executing a QG grows linearly with $|QG|$. 2) Because of the independent cluster storage, the cluster maintenance effort for different queries cannot be shared. To solve these problems, we propose below 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 introduce an important property of density-based cluster structures that will later be exploited to form the basis for efficient multiple query sharing. We call this the "*growth property*" of 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, $p \in C_i$ implies $p \in C_j$, we say that C_i is **contained** by C_j, denoted by $C_i \subset C_j$.*

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

We now characterize the possible relationships between the two cluster sets between which the *growth property* holds (see Figures 4 and 5 for an example).

Observation 4.3. *Given Clu_Set1 and Clu_Set2 with Clu_Set2 a **growth** of Clu_Set1, then any cluster C_j in Clu_Set2 must either be a **New** cluster (for any $p \in C_j$, $p \notin C_i$, if C_i is in Clu_Set1), an **Expansion** of a single cluster in Clu_Set1 (there exists exactly one C_i in Clu_Set1 such that $C_i \subset C_j$), or a **Merge** of multiple clusters in Clu_Set1 (there exist C_i, C_{i+1},...$C_{i+n} (n > 0)$ in Clu_Set1 with C_i, C_{i+1},...$C_{i+n} \subset C_j$.*

The black circles in Figures 4 and 5 represent the data points belonging to both cluster sets, while the gray ones represent those belonging to Clu_Set2 only. The cluster C_4 in Clu_Set2 is a merge of cluster C_1 and C_2 in Clu_Set1, while the clusters C_5 and C_6 in Clu_Set2 are an expansion of cluster C_2 in Clu_Set1 and a new cluster respectively. Generally, if Clu_Set2 is a "growth" of Clu_Set1, any two data points belonging to the same cluster in Clu_Set1 will also belong to the same cluster in Clu_Set2.

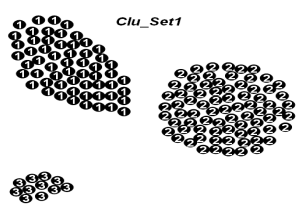

Figure 4: Cluster Set 1 contains 3 clusters

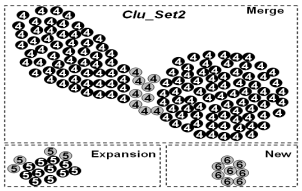

Figure 5: Cluster Set 2 contains 3 clusters. Cluster Set 2 is a growth of Cluster Set 1

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 them independently, we incrementally store the cluster growth information from one cluster set to another. Figures 6 and 7 respectively give examples of independent and hierarchical cluster structures built for the two cluster sets shown in Figures 4 and 5.

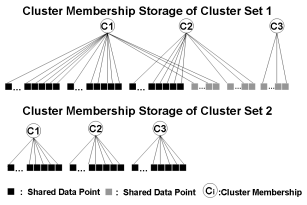

Figure 6: Independent Cluster Membership Storage for Cluster Sets 1 and 2

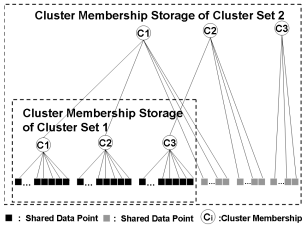

Figure 7: Hierarchial Cluster Membership Storage for Cluster Sets 1 and 2

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 the cluster in Clu_Set2 that contains it. Such growth information is now based on the **granularity of complete clusters** rather than **individual cluster members** - thus saving memory space for storing cluster memberships.

Lemma 4.1. *Given a query group QG with the growth property transitively holding 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 intuition here is that this is equal to the relationship between the total size of a binary heap and the number of leaf nodes of this heap. The formal proof can be found in our technical report [17].

Besides the benefit of huge memory savings, this hierarchical cluster structure also realizes integrated maintenance for cluster sets identified by multiple queries, and thus saves the computational

resources needed to maintain them independently. This general principle forms the foundation for our multiple query optimization.

5. SHARING AMONG QUERIES WITH ARBITRARY PATTERN PARAMETERS

In this section, we discuss the shared processing of multiple queries with arbitrary pattern parameters, namely arbitrary θ^{range} and θ^{cnt}. Here we assume that all 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.

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

We first look at the case that all queries have the same θ^{range} but arbitrary θ^{cnt}. Given the same θ^{range}, the neighbors of each data point identified by all queries are the same. This indicates that for all member queries, the neighborships identified in the same window are exactly the same. However, this does not imply that the cluster structures identified by all queries are identical, because the different $\theta^{cnt}s$ of the 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 being a non-core point for Q_2 having query $Q_2.\theta^{cnt} = 10$. Recall the hybrid neighborship abstraction (in Section 3) requires each non-core point to store the links to its exact neighbors, while the core points store the cluster membership only. Thus, a data point may need to store different types of neighborship abstractions depending on its roles assigned by different queries. To solve this problem, we turn to the *growth property* of density-based cluster structures discussed in Section 4.

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

Lemma 5.1 holds, because the more relaxed count threshold $Q_i.\theta^{cnt}$ can only cause "extra core points" to be identified by Q_i. Thus it brings a new cluster or an expansion or merge of existing clusters identified by Q_j. Formal proof of this lemma can be found in [17].

Figure 8 demonstrates an example of the cluster sets identified by three queries having the same θ^{range} but different $\theta^{cnt}s$ in a subpart of GMTI data [7].

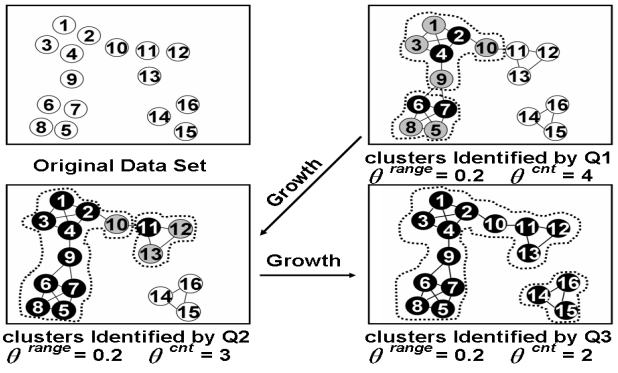

Figure 8: Clusters identified by queries with different θ^{cnt}

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

As discussed in Section 4, once the *growth property* holds among cluster sets, we can build the hierarchical cluster structure for them. Thus for each predicted window, we can represent multiple predicted views identified by different queries in an integrated structure. We denote such integrated representation of predicted views across queries with arbitrary θ^{cnt} by *IntView_*θ^{cnt}. For each W_i, *IntView_*θ^{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 decreasing order of θ^{cnt}. Figure 9 gives an example of a *IntView_*θ^{cnt}, which represents the predicted views from Figure 8 identified by three queries.

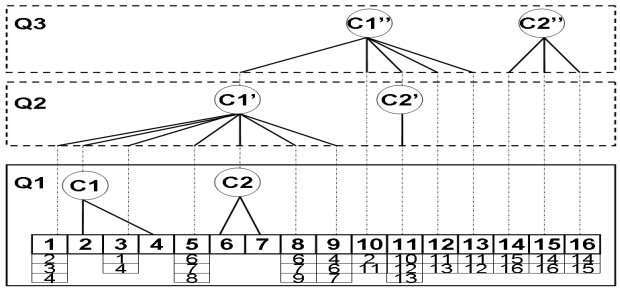

Figure 9: *IntView_*θ^{cnt}: Integrated representation for predicted views identified by three different queries

*IntView_*θ^{cnt} successfully integrates the representations of multiple predicted views into a single structure, thus saving the memory space from storing them independently.

Lemma 5.2. *Given the maximum window size allowed for the member queries in $|QG|$, the upper bound of the memory space needed by IntView_θ^{cnt} is independent from $|QG|$, the cardinality of the query group represented by it.*

Lemma 5.2 holds because both types of meta-information that need to be maintained by *IntView_*θ^{cnt}, namely the cluster memberships and the exact neighbors of data points, have upper bounds independent from $|QG|$. The formal proof of Lemma 5.2 can be found in [17]. Obviously, without using *IntView_*θ^{cnt}, the memory space needed for independently storing the cluster memberships identified by all queries in QG will increase linearly with $|QG|$. Our method now makes it independent from $|QG|$.

Maintenance of *IntView_*θ^{cnt}. Besides the memory savings, we can also update multiple predicted views represented by *IntView_*θ^{cnt} incrementally and thus save computational resources. For each new data point p_{new}, we start the update process from the bottom level of *IntView_*θ^{cnt}, namely the predicted view identified by the query with the largest θ^{cnt}. Then we incrementally propagate the effect of this new data point to the next higher level predicted views. Using the example shown in Figure 9, a new data point identified to have 3 neighbors in the window is a non-core in the bottom (most restricted) level predicted view, where $\theta^{cnt} = 4$. So, at the bottom level, we simply add all its neighbors to its neighbor list. However, its effect on the 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 detailed maintenance algorithm of *IntView_*θ^{cnt} here to save space, but emphasize the efficiency of the maintenance process. First, 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. The reason is that as a non-core point in the lower level predicted views, it would already have stored the links to its neighbors and thus would have direct access to them. Second, as the "growth" of cluster sets identified in predicted views is incremental, less and less maintenance effort will be needed as we handle predicted views at higher levels.

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, two data points p_i and p_j with distance equal to 0.2 will be considered to be neighbors in Q_1 with $\theta^{range} = 0.1$, but not in Q_2 with $\theta^{range} = 0.4$. As the neighborships identified by queries are different, the clusters identified by them are likely to be different as well.

Based on our experience of designing *IntView_*θ^{cnt}, we now 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 Q_i and Q_j specified on the same data set, with $Q_i.\theta^{cnt} = Q_j.\theta^{cnt}$ and $Q_i.\theta^{range} \geq Q_j.\theta^{range}$, the cluster set identified by Q_i is a "growth" of that identified by Q_j.*

This lemma holds because the additional neighborships identified by the more relaxed range threshold $Q_i.\theta^{range}$ will only either bring new clusters or the expansion or merge of existing clusters identified by Q_j. Formal proof can be found in [17]. Figure 10 demonstrates the cluster sets identified by three queries having the same θ^{cnt} but different θ^{range} settings. It uses dashed lines between data points to represent the extra neighborships identified.

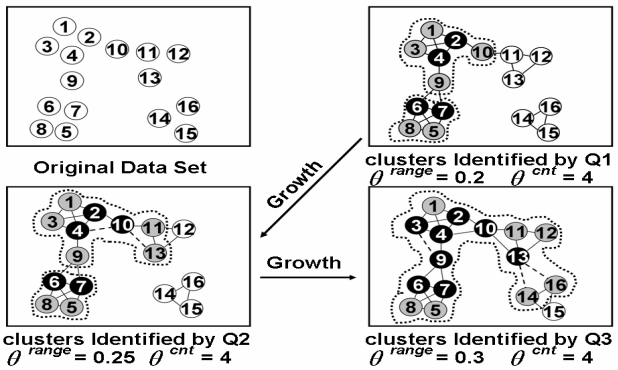

Figure 10: Clusters identified by queries with different θ^{range}

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 queries with arbitrary θ^{cnt}. We call it *IntView_*θ^{range}. Similar with *IntView_*θ^{cnt}, *IntView_*θ^{range} starts from the predicted view with the most restricted clusters, namely the predicted view representing Q_i with the smallest θ^{range} among QG. 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, needs to be stored.

Again, we can prove that the upper bound of the memory space needed by *IntView_*θ^{range} is independent from $|QG|$. The proof can be found in [17]. Since *IntView_*θ^{range} is very similar in concept to *IntView_*θ^{cnt}, we omit the details of its maintenance.

5.3 Arbitrary θ^{range}, Arbitrary θ^{cnt} Case.

Now we discuss the case where QG has queries with totally arbitrary pattern parameters, namely arbitrary θ^{range} and arbitrary θ^{cnt} values. Although the *growth property* has been shown to hold 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, we observe that it does not necessarily hold if both query parameters of them differ. To take advantage of the compact structure of integrated representation of the predicted views, we again study in what situations when the *growth property* would hold.

Lemma 5.4. *Given Q_i and Q_j specified on the same dataset, with $Q_i.\theta^{cnt} \leq Q_j.\theta^{cnt}$ and $Q_i.\theta^{range} \geq Q_j.\theta^{range}$, the cluster set identified by Q_i is a "growth" of that identified by Q_j.*

This lemma can easily be proven by the transitivity of the *growth property*. The formal proof is omitted here but can be found in [17].

To more intuitively describe the relationship between two such queries Q_i and Q_j, with the *growth property* holding among the cluster sets identified by them, 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. For each predicted window, we aim to build a single structure to represent the predicted views for all queries. 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 given predicted views into a single linear hierarchy.

Our solution is to build a **Predicted View Tree**, which integrates multiple predicted view hierarchies as branches into a single tree structure. In this tree structure, each predicted view (except the root) only needs to maintain the incremental information (cluster "growth") from its parent, much like the predicted views in *IntView_*θ^{range} and in *IntView_*θ^{cnt}. In particular, such a predicted view tree starts from the predicted view that represents the most restricted query among QG as its root. This would be 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 exist in QG, we build a virtual one by generating a query with the smallest θ^{cnt} and the largest θ^{range} among QG. Its predicted view will be used for predicted view tree maintenance but it will never generate any output.

Starting from the root, we iteratively pick (and remove) the most restricted queries remaining in QG and put their predicted views as the next level of the tree. Here, a member query Q_j is one of the most restricted queries remaining in QG if there does not exist any other query Q_i in QG, which is "more restricted" than Q_j. This process of figuring out "the most restricted queries" at each level is equal to the problem of calculating the Skyline [19] in the two dimensional space of θ^{range} and θ^{cnt}. Since this process can be conducted offline during query compilation, any existing skyline algorithm can be employed to solve this problem.

To connect queries at adjacent levels, 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 a parent query Q_m that is most similar to Q_n, indicating that there exists least growth between the cluster set identified by them, compared with that between the cluster set identified by Q_n and any other Q_o on the $(i-1)^{th}$ level. Based on our analysis, for two member queries, their difference on neighborships identified (decided by their difference on θ^{range} settings) is more likely to cause the variations on the cluster sets identified, compared to their difference on the requirement for core points (decided by their difference on θ^{range} settings). So, when we determine the parent predicted view, although we consider the similarity between both pattern parameters, more weight is given to that between $\theta^{range}s$. We can prove that, in our proposed predicted view tree structure, each predicted view maintains the smallest increment possible. The detailed algorithm of building the predicted view tree and the proof to its properties can be found in [17]. To unify the names of the hierarchical structures representing multiple predicted views, we call the predicted view tree *IntView_θ*.

Although *IntView_θ* is a tree structure, instead of the linear sequences like *IntView_*θ^{cnt} and *IntView_*θ^{range}, they all three share the essence that each predicted view is incrementally built based on the predicted view most similar to it. We call the queries on each path of *IntView_θ* a group of ***shared queries***.

Lemma 5.5. *The upper bound of the memory space needed by IntView_θ for any group of shared queries is independent from the number of queries in this group.*

Since the *growth property* transitively holds among the cluster sets identified by all queries on the same path of *IntView_θ*, the independency between the upper bound of the memory space and the number of queries can be proven as in Lemma 5.2.

The maintenance process of *IntView_θ* is also similar to that for *IntView_*$\theta^{cnt/range}$. For each new data point, we start the maintenance from the root of *IntView_θ*, and then incrementally maintain the predicted views on the next higher level of *IntView_θ*.

Theorem 5.1. *For a given QG with member queries having arbitrary pattern parameters, IntView_θ achieves full sharing of both memory space and query computation.*

Lemma 5.1 is ture because *IntView_θ* achieves completely incremental storage and computation for the predicted views maintained by multiple queries. The formal proof can be found in [17].

5.4 *IntView_θ* In Multiple Predicted Windows

Given the assumption that all the queries in QG share the same window parameters, the predicted windows that need to be maintained by them are the same. A straightforward way to serve a query group that needs to maintain N predicted windows, is to use N *IntView_θ*s to represent these N predicted windows independently, and maintain them independently at arrival of each new data point. Figure 11 gives an example of using four *IntView_θ*s to represent four predicted windows for a query group with 5 queries.

This strategy realizes the full sharing in each predicted window (Lemma 5.1). However, no sharing is yet achieved across the different predicted windows. In Section 6, we will introduce a more sophisticated methods that succeed to integrate multiple *IntView_θ* representing different windows into a single structure.

6. SHARING FOR QUERIES WITH ARBITRARY WINDOW PARAMETERS

Next, we study the case that the window parameters can vary, while the pattern parameters are common among the queries.

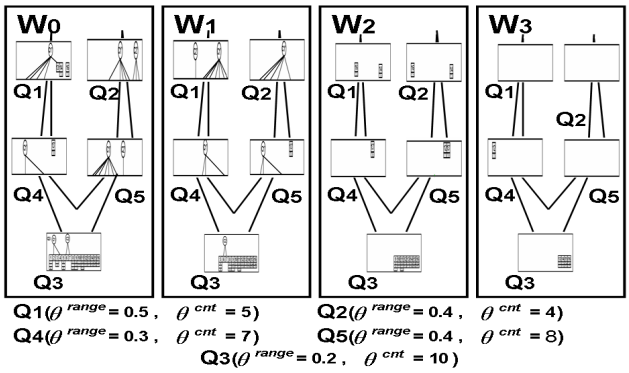

Figure 11: *IntView_θ*: integrated representation for predicted views for five queries with arbitrary pattern parameters

6.1 Same win, Arbitrary $slide$ Case

In this case, all 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 the same portion of the data stream. More specifically, at any given time the data points falling into the windows of different queries are the same. Then, the only difference among queries is that they need to generate output at different moments due to different slide sizes. For example, given three queries Q_1, Q_2 and Q_3, with $Q_1.win = Q_2.win = Q_3.win = 10(s)$, $Q_1.slide = 2(s)$, $Q_2.slide = 3(s)$ and $Q_3.slide = 6(s)$, the query windows of them cover exactly the same portion of the data stream at any given time, while they are required to output the clusters at every 2, 3 and 6 seconds respectively. So, to serve the different output time points, they need to build predicted windows starting at different times. These predicted windows should each end at a future output time point, and thus cover the data points on which the clusters need to be formed for that output time point.

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. Q_{meta} has the same window size as all member queries in QG, while its slide size is no longer fixed but rather 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 three member queries, we build a meta query Q_{meta} with $win = 10s$. At wall-clock time 00:00:10, the slide size of Q_{meta} should be 2s, as 00:00:12 will be the nearest time at which a member query (Q_1) needs to be answered. Then its slide size is adapted to 1s, 1s and 2s at 00:00:12, 13 and 14 respectively for the same reason.

Knowing the slide sizes of Q_{meta}, we build predicted windows for Q_{meta} based on the output time. Using the earlier example, at wall-clock time 00:00:10, we would have built eight predicted windows for Q_{meta} , which start from 00:00:00, 00:00:02, 00:00:03, 00:00:04, 00:00:06, 00:00:08, 00:00:09 and 00:00:10 respectively, as each of them corresponds to one output time point for at least one member query. Among these eight "predicted windows", many of them are serving multiple queries. For example, a single "predicted window" starting at 00:00:06 will be used to answer Q_1, Q_2

and Q_3. Each would have maintained one predicted window starting at this time if executed independently. As the predicted views representing such predicted windows for a meta query are no different from those needed for any single query, a straightforward way to maintain them is to use the maintenance method introduced in *Extra-N* [18] to update them independently at the arrival of each new data point. We will discuss further optimizations for this in the later part of this section.

In conclusion, this meta query strategy saves the system resources for answering a 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 from it. 2) Predicted views are shared among member queries requiring output at the same time. The specific amount of sharing depends on the overlaps of queries' output times.

6.2 Same $slide$, Arbitrary win Case

In this case, although the window sizes may vary, we hold the slide size steady, indicating that their output schedules are identical. Here we first use the simplifying assumption that all the window sizes of the member queries are multiples of their common slide size. Since the queries have the same slide size, it is easy to observe that all queries require output at exactly the same moments. Based on this observation, an important characteristic can be discovered.

Lemma 6.1. *Given a query group QG with member queries having the same slide size but arbitrary window sizes (multiples of $slide$), the predicted windows maintained for Q_i with $Q_i.win$ the largest window size among QG will be sufficient to answer all member queries in QG.*

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. 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. 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. The formal proof of this can be found in [17].

In summary, we only need to maintain the predicted windows for a single member query with the largest window size, and then we can answer all the queries in the query group with different predicted windows maintained. Clearly, full sharing is achieved.

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 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.

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 $Q_{meta}.win = max(Q_i.win)_{(1 \le i \le 3)} = 10$. Then we adaptively change its slide size based on the next nearest output time point required by at least one query. 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 12 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.

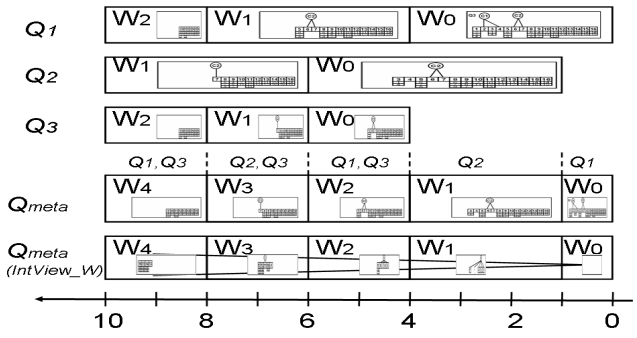

Figure 12: Predicted views maintained by three queries independently versus those maintained by a single meta query

Integrated Representation of Predicted Views across Multiple Windows. Although *Extra-N* [18] can be applied to maintain the predicted views of the meta query, it achieves no sharing across the multiple predicted windows (see Section 3). Now we introduce a further optimization for maintaining predicted views of multiple predicted windows. [16] found that even for a single query , the *growth property* holds among the cluster sets identified by it in different predicted windows. In particular, for a single query Q_i at any given time T with W_n being the current window, the cluster set identified by it in a predicted window W_{n+i} is always a **growth** of that identified by it in W_{n+i+1}. Figure 2 demonstrates an example of the predicted views representing four successive predicted windows of a single query. The formal proof of this can be found in [17]. This *growth property* allows us to build an integrated structure to incrementally store and maintain the predicted views across multiple windows, called *IntView_W*. *IntView_W* starts from the predicted view with the most restricted clusters, namely the one representing the newest predicted window, and then incrementally stores the growth information for those representing older predicted windows. Figure 13 gives an example of the *IntView_W* built for the meta query discussed earlier.

7. PUTTING IT ALL TOGETHER

Finally, we consider the general case with arbitrary pattern and window parameters. 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 combining the two techniques introduced in the last two sections, namely the *IntView_θ* technique (Section 5) and the meta query technique (Section 6). These two techniques can be easily combined, because they are orthogonal to each other. In particular, the *IntView*$_\theta^{cnt}$ technique is designed to share among a group of predicted views representing the same predicted window. We can consider this to be an "**inner-predicted-window**" optimization technique. On the other hand, the meta query technique is designed to minimize the number of predicted windows generated for multiple queries and to share the maintenance costs across them. So, it can be considered to be an "**inter-predicted-window**" optimization technique. These two orthogonal

techniques can be applied together to realize the full sharing of the member queries on both inner- and inter-predicted window levels.

We use an example to demonstrate this solution. 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 those shown in Figure 12. Then, for each predicted window built, we apply the *IntView_θ* technique to build a predicted view tree to integrate the predicted views (of different queries) in this window. For the predicted window starting from 00:00:04, a predicted view tree (*IntView_θ*) is built to represent the predicted views for Q_1 and Q_3.

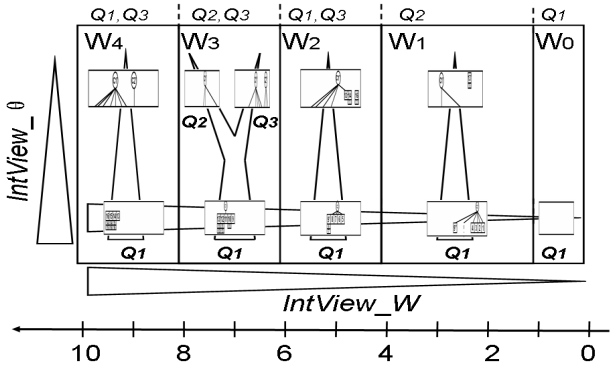

Figure 13: *IntView*: integrated representation for predicted views Identified by 3 Queries in 5 Predicted Windows

To apply the *IntView_W* technique (in Section 6), which allows us to share across multiple predicted windows, we use the most restricted query of the whole query group as the root of all the predicted view trees built in each of the different windows. By doing so, the predicted view trees in different windows predicted now start from the predicted view representing the 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 in different predicted windows, forming a single tree structure that realizes completely incremental storage and maintenance We call this ultimate hierarchical structure the *IntView*. Except the root of the *IntView*, all the predicted views within *IntView* only maintain incremental information. Figure 13 demonstrates the *IntView* built for the three queries mentioned in the earlier example. We call the final algorithm based on the *IntView* ***Chandi***. We sketch pseudo code describing the overall flow of the *Chandi* algorithm below. More details of *Chandi*, including the complete pseudo code, can be found in [17].

In conclusion, computation-wise, *Chandi* only requires a single pass through new data points, each running one range query search and communicating with its neighbors once for all shared queries on each path of *IntView*. 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 proven using the same methods as those used for proving Lemma 5.2) [17]. In short, *Chandi* achieves full sharing for multiple density-based cluster queries over sliding windows in terms of both CPU and memory resources.

Chandi **(**$stream$**,** QG**)**
1 For each new data point p_{new} in $stream$
  // purge
2 **if** $p_{new}.T > W_{oldest}.T_{end}$ (ending time of W_{oldest})
3 Purge(W_{oldest}); //purge the oldest predicted window
  // load
4 load p_{new} into index // we use GRID
  // 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(QG);
9 add new predicted window W_{newest} to *IntView*;
10 T_{output}=ScheduleNextOutputMoment();

8. EXPERIMENTAL STUDY

Our experiments are conducted on a HP Pavilion dv4000 laptop with Intel Centrino 1.6GHz processor and 1GB memory, running Windows XP. We implemented all algorithms with VC++ 7.0.

Real Datasets. We used two real streaming data sets. The first data set, GMTI (Ground Moving Target Indicator) data [7], 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 [12], which has one millions transaction records throughout the trading hours of a day.

Alternative Algorithms. To evaluate our proposed algorithm, for any input QG, we compare *Chandi*'s performance of two major alternative methods, executing QG with four alternative methods, namely executing one *Extra-N* algorithm [18] for each member query with and without sharing of range query searches (henceforth referred as *Extra-N with rqs* and *Extra-N*), and executing one *IncDBSCAN* algorithm [8] for each member query with and without sharing of range query searches (referred as *IncDBSCAN with rqs* and *IncDBSCAN*). The reasons why we choose them are: 1) *Extra-N* algorithm is the only algorithm we are aware of in literature solving density-based clustering over sliding windows; 2) *IncDBSCAN* algorithm is the most well known method for incremental density-based clustering.

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

In many applications, the domain knowledge or the specific analysis tasks may restrict some of the query parameters to particular values. To thoroughly understand the performance of the algorithms under different situations, we first evaluate at a time four test cases, each varying on only one of the four parameters. Then we relax this constraint to allow two arbitrary parameters. Finally we test the general cases with all four parameters arbitrary.

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 for the other three parameters for queries. The parameter settings in our experiment are selected based on a pre-analysis of the datasets. In

particular, we pick parameter ranges that allow member queries to identify all 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 more than the smallest number of clusters identified by the other, indicating that the cluster structures identified by different queries vary significantly. This is the worst case for our proposed sharing algorithm, because the larger the variations on cluster structures identified, the more maintenance costs are needed for our sharing strategy. To observe the performance of the algorithms when executing different numbers of queries, we use different random subsets of QG (sized from 5 to 20) to execute against the GMTI data.

Arbitrary θ^{cnt} ***case.*** We use $\theta^{range} = 0.01$, $win = 5000$ and $slide = 1000$, while varying θ^{cnt} from 2 to 20. In this 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 C1 and C2 of Figure 14, both the CPU time and the memory space used by all four alternatives increases as the number of queries increases. However, the utilization of CPU resources by *Chandi* is significantly lower than those consumed by other alternatives, especially when the number of the member queries increases. The memory consumption of *Chandi* is almost equal to *IncDBSCAN with rqs*, which maintains very little meta-information but relies on range query searches to re-collect them, and much lower than *Extra-N with rqs*. This matches our analysis in Section 5. In particular, since for any data point its neighbors identified by all queries are the same, the cluster growth information that needs to be maintained by *Chandi* among the queries is very simple and can be updated efficiently.

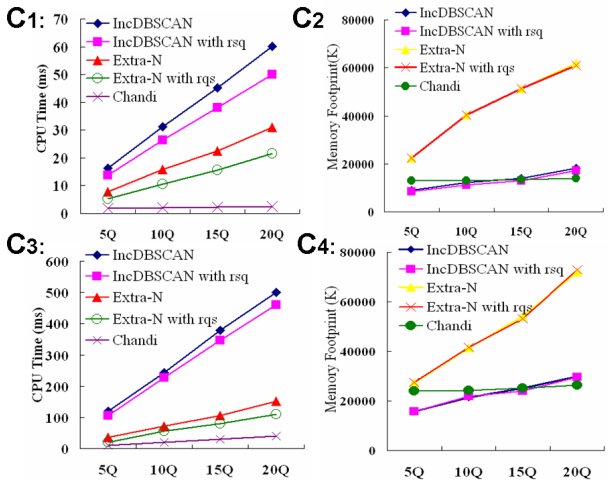

Figure 14: Comparison of CPU and memory utilizations for five algorithms in arbitrary θ^{cnt} case (C1 and C2) and in arbitrary θ^{range} case (C3 and C4)

Arbitrary θ^{range} ***case.*** In this experiment, we use $\theta^{cnt} = 10$, $win = 5000$ and $slide = 1000$, while varying θ^{range} from 0.01 to 0.1. 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 C3 and C4 of Figure 14, a similar situation can be observed, while the increase of the resource utilization for *Chandi* is more obvious in this scenario. This is for two main reasons. 1) Since θ^{range} varies among the queries, the range query search costs increase along with 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, the "extra-neighborships" are more likely to cause cluster structure changes and thus require *Chandi* to maintain more meta-information in *IntView*. The performance of other competitors, especially for *IncDBSCAN* that consumes large numbers of range query searches, is largely affected by the increasing cost of range query searches as well.

Arbitrary $window$ ***or*** $slide$ ***case.*** In these two cases, the cost of *Chandi* is almost unchanged as the number of queries increases. It thus achieves even more significant savings compared with the previous two cases. This is because we always maintain a single meta query and thus answer the whole query group, no matter how many queries are in the query group. The detailed experimental results and analysis for these two cases can be found in [17].

Evaluation for Two-Arbitrary-Parameter Cases. We further evaluate the two test cases by varying two parameters among the member queries.

Arbitrary Pattern Parameters. In this case, we use $win = 5000$, $slide = 1000$, while varying θ^{cnt} from 2 to 20 and θ^{range} from 0.01 to 0.1. As shown in Figures 15, *Chandi* still consumes significantly less CPU time compared with the alternatives, although the increase of its CPU consumption corresponding to the increase of queries is more obvious. This is because totally arbitrary pattern parameters leads to even larger differences among the clusters identified by different queries, and thus increases the maintenance costs of *Chandi*. In this test case, the largest number of clusters identified by the member query (with $\theta^{range} = 0.01$ and $\theta^{cnt} = 14$) reaches 35, while the smallest number of clusters identified by the query (with $\theta^{range} = 0.1$ and $\theta^{cnt} = 3$) is only 2. Even though the CPU time used by *Chandi* is still 63% less than that used by *Extra-N with rqs*, when executing the query group sized 20.

Arbitrary Window Parameters. We use $\theta^{cnt} = 10$ and $\theta^{range} = 0.01$, while varying win from 1000 to 5000 and $slide$ from 500 to 5000 (for any Q_i, $Q_i.slide < Q_i.win$). As shown in Figure 16, the savings achieved by *Chandi* are even higher than in the arbitrary pattern parameter case. This is because, the variation of both window parameters does not affect the principle of how the meta query strategy works. In particular, the cost of maintaining the meta query 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.

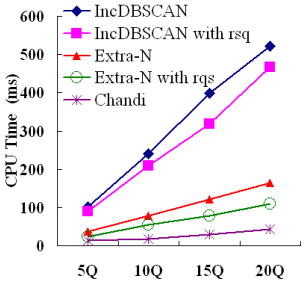

Figure 15: CPU time used by five algorithms in arbitrary pattern parameter case

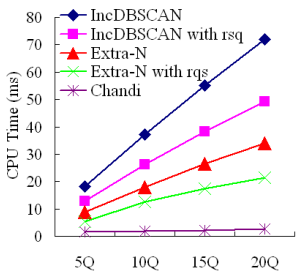

Figure 16: CPU time used by five algorithms in arbitrary window parameter case

General Case: Four Arbitrary Parameters. Finally, we evaluate the general case with all four parameters being varied arbitrarily. We divide this experiment into three cases, each measuring the performance of the algorithms when executing different numbers of queries. In particular, for each test case, we generate 30 query groups each with N member queries (N equals to 20, 40 and 60 for

three cases respectively). Each query group is independently generated, and the member queries in each group are randomly generated with parameter settings: $\theta^{cnt} = 2$ to 20, $\theta^{range} = 0.01$ to 0.1, $win = 1000$ to 5000 $slide = 500$ 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, 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 for raw data (for storing actual tuples) and the memory used for meta-data.

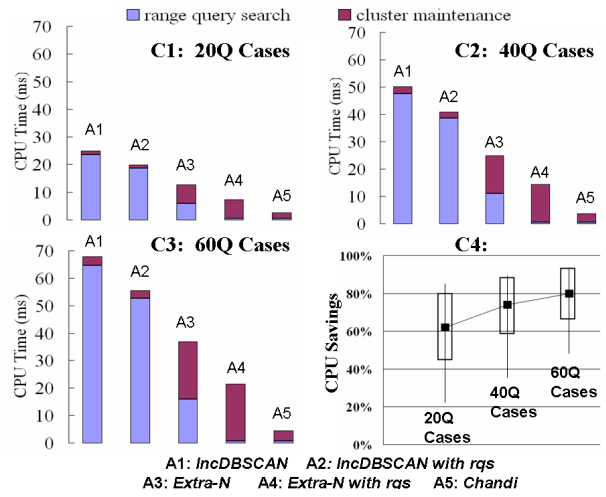

Figure 17: Detailed comparison of CPU time consumption of five algorithms

As shown in C1, C2 and C3 of Figure 17, we observe that the average CPU time used by *Chandi* is 70, 76, and 85 precent lower then the best alternative method, *Extra-N with rqs*, in the three cases respectively. In particular, the CPU time used by *Chandi* to conduct range query searches is always less than 10% compared with that needed by *IncDBSCAN* with rqs. This is because *Chandi* only requires each new data point to run one range query search when it arrives at the system, while *IncDBSCAN* relies on repeated range query searches to determine the cluster changes. The CPU time used by *Chandi* to maintain meta-information is at least 62% less than that used by *Extra-N with rqs*. This is because *Chandi* updates the meta-information for different queries integrally, while *Extra-N* maintains them independently.

Besides the comparison of the average system resource consumption, we also measure the savings of *Chandi* for each individual query group in all three test cases. In particular, for each query group, we measure the difference in resource utilization between *Extra-N with rsq* and *Chandi*, which corresponds to the difference between executing them using the best existing technique and our proposed strategy. More specifically, for each group, we first calculate the difference on CPU (or memory) utilizations between *Chandi* and *Extra-N with rqs*. 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 Figure 17, although the minimum savings achieved by *Chandi* is only around 20%, which is caused by too large variations on the parameter settings, it 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 are 23%, the maximum savings reach 84% , and the standard deviation is only 19% . As the number of queries in each group increases, the savings achieved by *Chandi* are even higher in the other two test cases. In particular, the average saving achieved by *Chandi* of CPU time increases to 80% when the number of queries in each group increases to 60. The minimum and maximum savings of 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 groups 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%. Due to page limitations, we omit the charts showing savings of the memory space.

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 earlier experiment, the member queries in the query group are randomly generated with the arbitrary parameter settings in certain ranges. In particular, the parameters settings in this experiment are $\theta^{cnt} = 2$ to 30, $\theta^{range} = 0.001$ to 0.01, $win = 1000$ to 5000 and $slide = 500$ to 5000.

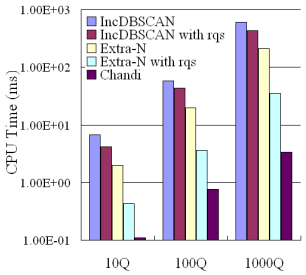

Figure 18: CPU time used by five algorithms in logarithmic scale

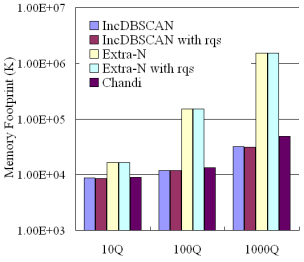

Figure 19: Memory space used by five algorithms in logarithmic scale

As shown in Figures 18 and 19, 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 increases 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.

9. RELATED WORK

Traditionally, clustering algorithms [9, 11, 20] are designed for static environments. More recently, as stream applications are be-

coming prevalent, the problem of clustering streaming data is being tackled in the literature [1, 4, 18].

As a well-known clustering algorithm, DBSCAN [9] is the first to propose density-based clustering for static data. Later an incremental DBSCAN [8] algorithm was introduced to incrementally update density-based clusters in data warehouses. However, as both analytically and experimentally shown in [18] as well as in our experimental study, since all optimizations in [8] were designed for single updates (a single deletion or insertion) to the data warehouse, it is not scalable to highly dynamic streaming environments.

Algorithms for density-based clustering over streaming data include [5, 6, 18]. Among these works, [6] and [5] have goals different from ours, because they are neither designed to identify the individual cluster members 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. The only algorithm we are aware of that detects density-based clusters in sliding windows is [18]. Our experiments show that our proposed sharing strategy largely outperforms the solution of executing *Extra-N* independently for multiple queries.

As a general query optimization problem, multiple query optimization has been widely studied for not only static but also streaming environments. Previous research on sharing focuses on operators, such as selection and join operators [10, 13, 15], and aggregation [3, 14]. To our best knowledge, none of them discuss the sharing for clustering operators. General principles used in these works, such as query containment [10] 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 traditional SQL query sharing. The meta-information we maintain, namely the cluster structures defined by individual objects as well as the topological relationships among them, is more complex than meta-information 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.

10. CONCLUSION

In this work, we resent the first framework, called *Chandi*, for efficient shared processing of a large number of density-based clustering queries over streaming windows. It is the first step of applying multiple query optimization principles from the field of database to process large number of data mining requests in stream environments. For answering multiple clustering queries with arbitrary parameter settings, *Chandi* achieves full sharing of 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 method while using 85% less memory space. More savings can be achieved if the queries have similar parameter settings. *Chandi* also exhibits excellent scalability in terms of being able to handle large numbers of queries under high speed input streams in our experiments.

Based on the general principles learned in this work, such as the hierarchical pattern representation for multiple queries, we will explore the potential for the shared execution for other mining types, such as other cluster types, outliers, associations, etc. Efficient storage and management mechanisms for those detected patterns also become an important topic for future exploration.

11. REFERENCES

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

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

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

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

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

[6] Y. Chen and L. Tu. Density-based clustering for real-time stream data. In *KDD*, pages 133–142, 2007.

[7] J. N. Entzminger, C. A. Fowler, and W. J. Kenneally. Jointstars and gmti: Past, present and future. *IEEE Transactions on Aerospace and Electronic Systems*, 35(2):748–762, april 1999.

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

[9] 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*, pages 226–231, 1996.

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

[11] J. A. Hartigan and M. A. Wong. A k-means clustering algorithm. *Applied Statistics*, 28(1).

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

[13] S. Krishnamurthy, M. J. Franklin, J. M. Hellerstein, and G. Jacobson. The case for precision sharing. In *VLDB*, pages 972–986, 2004.

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

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

[16] D. Yang, E. A. Rundensteiner, and M. O. Ward. Highly efficient neighbor-based pattern detection over streaming windows. *WPI Technical Report WPI-CS-TR-09-06*, june 2009.

[17] D. Yang, E. A. Rundensteiner, and M. O. Ward. Multiple query optimization for density-based clustering queries over streaming windows. *WPI Technical Report: WPI-CS-TR-09-04*, april 2009.

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

[19] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient computation of the skyline cube. In *VLDB*, pages 241–252, 2005.

[20] 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.