Distributed multi-query optimization of continuous clustering queries

Sobhan Badiozamany
Supervised by Tore Risch
Department of Information Technology
Uppsala University
Box 337, SE-751 05,
Uppsala, Sweden
sobhan.badiozamany@it.uu.se

ABSTRACT

This work addresses the problem of sharing execution plans for queries that continuously cluster streaming data to provide an evolving summary of the data stream. This is challenging since clustering is an expensive task, there might be many clustering queries running simultaneously, each continuous query has a long life time span, and the execution plans often overlap. Clustering is similar to conventional grouped aggregation but cluster formation is more expensive than group formation, which makes incremental maintenance more challenging. The goal of this work is to minimize response time of continuous clustering queries with limited resources through multi-query optimization. To that end, strategies for sharing execution plans between continuous clustering queries are investigated and the architecture of a system is outlined that optimizes the processing of multiple such queries. Since there are many clustering algorithms, the system should be extensible to easily incorporate user defined clustering algorithms.

1. INTRODUCTION

Compared to conventional database applications, a Data Stream Management System (DSMS) has different data processing requirements. First, continuous queries run for very long periods of time over data streams. Second, as the data flows through the system, only a limited window of data is presented at a g iven point in time. Sliding windows are commonly used for capturing the evolving behavior of data streams, which requires efficient incremental algorithms. Finally, since queries stand for a v ery long time, at any point in time there are potentially many queries that have overlapping computations. In particular, they might share expensive computations such as clustering, aggregations, and filtering in presence of overlapping window specifications.

Examples of such data streaming workloads can be found in monitoring applications with many users and queries, e.g. urban traffic monitoring, stock trading, and industrial sensor data monitoring. The essence of data streaming is to continuously summarize the data. When the exact grouping of data is unknown, clustering is a very good candidate for explorative grouping of similar data over which statistics is computed.

Proceedings of the VLDB 2014 PhD Workshop

Multi-query optimization has been studied in conventional databases since 80s [12] motivated by the fact that several queries might share the same data. Multi-query optimization is even more beneficial in data streaming applications. To elaborate the extra benefits of multi-query optimization, we compare the characteristics of conventional OLAP and OLTP workloads with data streaming workloads in Table 1. Data streaming has similar characteristics as OLAP: Since they both have long query life spans there is higher potential for overlapping computations, and since they both contain expensive summarization queries shared computation of queries in beneficial. Note that since the life span of queries is even longer in data streaming, the sharing is more beneficial.

A sharing solution has to be distributed in today's widespread distributed computing platforms where resources are limited and have a p rice tag. Therefore the focus of this PhD project is to develop novel methods and system architecture for optimization of multiple clustering queries over data streams in a distributed environment.

Table 1. Characteristics of different data processing workloads

Workload	Life span of active queries	Prevalence of summarization queries	Computation overlap in the active query set
OLTP	Short	Low	Low
OLAP	Long	Very High	Potentially high
Data Streaming	Very Long	Very High	Very high

Clustering data points into disjoint sets is similar to conventional grouped aggregation, with two differences, first the process of clustering is much more expensive than grouping, and second, incremental maintenance of clusters is challenging. While there have been several publications on optimizing multiple *Aggregate Continuous Queries (ACQs)* [9] [10] [11] [13] [14], there has been little research on the task of optimizing multiple *Continuous Clustering Queries (CCQs)*.

A general system that optimizes shared execution of multiple CCQs must fulfill the following three main requirements:

1. Since in real scenarios resources are always limited, the system must provide resource oriented scalability, i.e. given a certain resource allocation, it must minimize the response time. The key here is using shared processing techniques.

2. To facilitate exploiting new resources, the system must be distributable.
3. The sharing techniques should be independent of specific clustering methods. Therefore a general system should be extensible so that new clustering algorithms can be added to it in a non-intrusive manner.

The rest of the paper is organized as follows. Section 2 covers the background, mainly the related work on multi-ACQ and multi-CCQ optimization indicating the relevance to our research problem, leading to Section 3 where the research questions are stated. Section 4 defines an extensible generic clustering query operator and sketches how it can be distributed over several computation nodes. Section 5 outlines the architecture of a distributed multi-CCQ processing system.

2. Background and related work

First we define multiple *Continuous Summarization Queries (CSQs)* over sliding windows as a general concept covering both ACQs and CCQs. We then cover the related work on maintaining non-shared CSQs over sliding windows. Then multi-ACQ optimization is discussed and finally the related work and remaining challenges for multi-CCQ optimization is presented.

2.1 Multiple CSQs over sliding windows

Assume we have a data stream *DS* with a set of attributes A $\{A_1 \ldots A_n\}$. We define Q $\{Q_1 \ldots Q_n\}$ as a set of CSQs where each $Q_i \in Q$ has the following properties:

- A window specification tuple $W=(R, S)$, where R and S are the range and stride parameters for the window.
- A subset of the attributes $G \subseteq A$ that specifies data grouping or clustering.
- A selection predicates P that selects tuples from *DS*.

We also define the set of all window specifications in Q, W^*, the set of all G in Q, G^*, and set of all selection predicates in Q, P^*.

For example, assume we have *DS* with A= {a, b, c, d}, Q= {Q_1, Q_2} where

- Q_1 is specified by W= (10, 2), G= {a, b}, P= (d=c_1), where c_1 is a constant.
- Q_2 is specified by W= (6, 3), G= {b, d}, P= (b=c_2), where c_2 is a constant.

Then W^*= {(10, 2), (6, 3)}, G^*= {a, b, d}, and P^*= {(d=c_1), (b=c_2)}.

2.2 Non-shared CSQs over sliding windows

ACQs are similar to CCQs because both of them form groups of data points. However, they differ in the cost of the group formation because in the conventional group-by aggregates the group formation is only dependent on equality of values, whereas in clustering the group formation is a very expensive similarity based operation. Furthermore, sliding a window for CCQs is more complex than removing elements from groups.

Consider a sliding window specified by the two parameters stride S and range R. Figure 1 illustrates how a sliding window can be maintained, for R=10 and S=2. The data stream is first broken down into contiguous pieces, i.e. *partial windows* (*PW0* to *PW9*).

To form the sliding window several consecutive partial windows are assembled. The size of these partial windows is determined by the stride S and range R parameters of the window specification. In Figure 1, each window is formed by assembling 5 partial windows, each of size 2. Notice that if S and R are time based, depending on the stream rate, there might be varying number of data points in each of the partial windows.

Based on such partial windows, processing an ACQ is commonly done in two steps, *partial aggregation* and *final aggregation* [10] [11] [13] [14]. The partial aggregation step is applied on each partial window (PW0 to PW9), thereby summarizing its contents to produce *partial grouped aggregates*. The final aggregation step forms ACQ results by rolling up the partial grouped aggregates corresponding to the full window.

For example, the query in Listing 1 calculates the number of vehicles in segments of streets over a window. The total number of vehicles is first calculated per partial windows of size 2 minutes, producing partial grouped aggregates. Then, to find the total number of vehicles in each 10 minute window, in the final aggregation step the corresponding 5 consecutive partial grouped aggregates are summed.

Listing 1: simple aggregate queries over data streams

```
SELECT seg-id, COUNT(*)
FROM   Traffic [RANGE 10 Minutes,SLIDE 2 Minutes]
GROUP BY seg-id
```

When the window slides, a partial window expires and needs to be evicted. Handling deletion under sliding window semantics for conventional grouped summaries (aggregates) is done incrementally by simply deducting the contribution made by the expired partial grouped aggregates from the ACQ results.

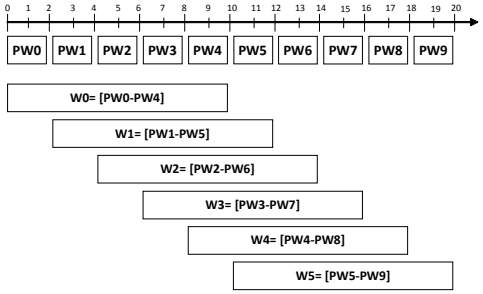

Figure 1. Sliding window maintenance

In contrast to ACQs, deleting expired elements from clusters is more complicated, since the impact of removing data points from clusters might not stay local to them: clusters might shrink, split, or disappear. For this reason CCQ deletion is handled in [15] [1] as follows. Instead of simply deducting the expired data points from the window summary, the need for deletion is eliminated by adding the new set of points into as many windows as they participate in. For example, referring back to the sliding window maintenance in Figure 1, the partial cluster summaries formed over PW5 are merged into all of the windows W1 to W9. Therefore at any point in time 5 windows are maintained simultaneously.

This strategy is similar to our earlier preliminary work on maintaining raw data indexes over sliding windows [3]. A potential drawback of the approach is the multiplied cost of

adding the individual partial grouped aggregates to all the windows they participate in. Furthermore, the amount of memory required to simultaneously maintain several windows will also be multiplied. The aforementioned drawbacks make the window maintenance expensive, particularly for long windows with short strides.

The incremental algorithm in [6] to delete expired elements from clusters in data warehouses was rejected in [15] motivated by the high cost of deletion from clusters. However, the experiments in [15] do not show the effect of scaling the window size when the incremental approach in [6] is compared to their deletion elimination approach.

Incremental deletion of expired elements from clusters remains a challenging open problem.

2.3 Multi-ACQs optimization

Gigascope [5] is a d ata stream management system which is designed for processing traffic monitoring queries over IP networks. In particular, it supports shared optimization of multiple ACQs [10]. An example ACQ in Gigascope is given in Listing 2.

Listing 2: simple Gigascope ACQ

```
SELECT win_time, source_IP, COUNT(*)
FROM   IP_header_stream [RANGE 1 Min, SLIDE 1 Sec]
GROUP BY win_time, source_IP
```

The execution strategy for such a query is illustrated in Figure 2 taken from [10]. It exemplifies the two tier distributed processing having partial aggregation and final aggregation, where the *Low level Filter, Transforms, and Aggregate (LFTA)* is the partial aggregation step and the *High level Filter, Transforms, and Aggregate (HFTA)* is the final aggregation step. For example, in Listing 2 the partial aggregation (LFTA) is COUNT, while the final aggregation (HFTA) is SUM.

Individual data points, e.g. C_1 in the figure, are continuously added to the partial grouped aggregates in LFTA (i.e. COUNT). The partial grouped aggregates in the LFTA are sent to the HFTA when a partial window becomes complete.

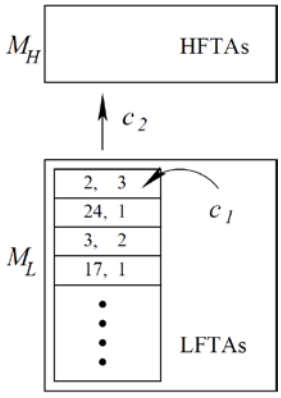

Figure 2. Single aggregation in Gigascope

Listing 3 is an example of a set of ACQs from [10], where Q={Q1, Q2, Q3}, W^*= {(1, 1)}, and G^*= {A, B, C}.

Listing 3: A set of ACQs with varying grouping attributes

```
/*Q1*/
SELECT A, COUNT(*)
FROM   R [RANGE 1 Min, SLIDE 1 Sec]
GROUP BY A


/*Q2*/
SELECT B, COUNT(*)
FROM   R [RANGE 1 Min, SLIDE 1 Sec]
GROUP BY B


/*Q3*/
SELECT C, COUNT(*)
FROM   R [RANGE 1 Min, SLIDE 1 Sec]
GROUP BY C
```

Figure 3 s hows the two main execution strategies in Gigascope investigated in [10]: naïve (none shared) (Fig. 3a), and *phantom* based (shared) processing (Fig. 3b).

In the naïve approach, each query maintains its index for grouping, so three hash indexes are maintained for grouping by A, B, and C, respectively. Every incoming tuple is matched against all three indexes.

In the phantom based approach, the input stream is first grouped by the *phantom* ABC, from which individual A, B, C groupings are *fed*. A phantom is a virtual grouping that is not used in any of the queries in Q, but is used to facilitate the shared processing of queries by the simple intuition that a finer grain grouping can feed several coarser grain groupings. Here the phantom ABC can feed any grouping specified using a subset of {A, B, C}, for example {A}, {B}, {C}, {A,B}, etc. The key point is that the *feeding* of A, B and C happens only when the partial grouped aggregates in a partial window are emitted. Since all queries in Listing 3 have the same slide = 1 second (W^*= {(1, 1)}), the feeding occurs once per second.

When not all windows have the same slide, i.e. when W^* is not a singleton set, [13] generalizes the method by materializing finer grain partial windows in the pre-aggregation phase, from which all windows in $W^*$are formed.

The phantoms and other hash tables form a *feeding graph* [10]. Since there might be many different feeding graphs, the optimization problem of finding the right feeding graph is also studied in [10].

Maintaining phantoms is beneficial for sharing the execution of several ACQs because, in contrast to the naïve approach, only a single look-up of the ABC index is made per incoming stream tuple. At the feeding time, there will be updates to the A, B and C indexes, but those are relatively infrequent, since the original tuples are already partially grouped in phantom ABC.

Multiple ACQs were also present in the DEBS 2013 Grand Challenge [8], where resource limitations were critical and therefore a shared execution strategy was vital, as shown in [2].

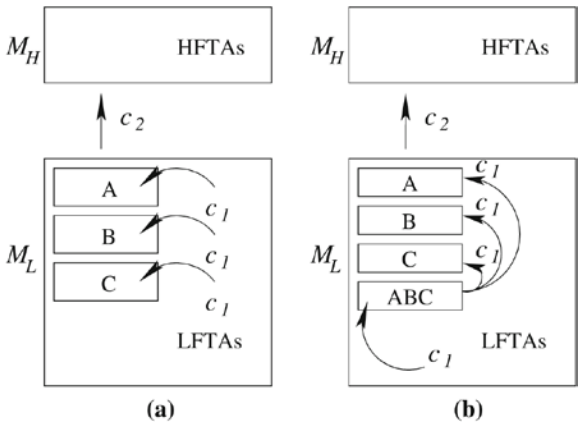

Figure 3. processing multiple aggregates in Gigascope

Gigascope does not support dynamic query optimization, and does not consider selection predicates P^* in multi-ACQ optimization. As an improvement Krishnamurthy et al. in [11] proposes a solution focusing on dynamic multi-ACQ optimization in presence of high query churn, i.e. queries frequently join and leave the system. To address dynamic query optimization, a single execution pipeline is proposed where addition and removal of queries from the pipeline is implemented by modifying data structures in different parts of the pipeline.

There are a number of shortcomings with the approach in [11]. First, no strategy for parallel or distributed execution is proposed. Second, the sharing scheme for selection predicates does not scale for the following reason. The system tags all input tuples with a bitmap signature, indicating what combination of query selection predicates they fulfill. Having this tag, the tuples can be assigned to *fragments,* which are the non-overlapping groups of tuples. The total number of fragments is 2^N, where N is the number of queries. Therefore the proposed solution is not scalable w.r.t. the number of queries. Another shortcoming in [11] is the lack of support for a general GROUP BY.

Guirguis et al. [14] [13] improve the single pipeline approach in [11] by incorporating the optimizations in Gigascope on sharing grouped aggregates, but the non-scalable selection predicate sharing problem remains, as does the problem of parallelized or distributed execution.

In general there is no system that combines the following two aspects of shared execution of multiple ACQs:

1. Efficient sharing of selection predicates.
2. Shared execution of a general GROUP BY operator.

Furthermore, automatic distributed or parallel multi-ACQ execution has not been addressed. For example, in [2] the parallelization was manual, which becomes very complex when there are many complex queries. This leaves room for improvement in multi-ACQs optimization.

2.4 Multi-CCQ optimization

Initial work on shared execution of clustering queries can be found in [16]. The authors propose a method to share execution of multiple density based CCQs for a query set Q that contains diverse density parameters and window specifications. In general the method is based on the deletion elimination approach explained in section 2.2 with the following limitations:

1. There is no support for specific clustering attributes, G, in queries. It is assumed that all queries in Q^* cluster the data based on all the attributes.
2. There is no support for selection predicates, P, i.e. all queries in Q^* have the same filter.
3. Only one density clustering method [15] is supported. There is no support for plugging in new clustering algorithms.
4. There is no distributed or parallel execution strategy.

3. The research questions

The following research questions are not addressed by any related work on multi-CCQ optimization:

1. How can the combination of P^*, G^*, and W^* be exploited for optimizing shared execution of multiple CCQs?
2. How can extensible clustering be supported? That is, how can the sharing framework be made independent of a specific clustering algorithm?
3. How can the query execution components be automatically distributed over several nodes?

Next a generic clustering framework that facilitates answering the research questions is outlined.

4. An extensible framework for processing distributed clustering queries

In this section a general framework for processing clustering queries is introduced, with two requirements in mind. First, it has to be extensible, i.e. it should be easy to plug in a variety of clustering methods. Second, it has to support distributed query execution.

Listing 4: A CCQ

```
SELECT
       CONVEX_HULL(*), COUNT(*)
FROM
       TRUCK_POSITIONS(RANGE 1 Min, SLIDE 1 Sec)
WHERE
       CITY='Stockholm'
CLUSTER BY
       X, Y
ALGORITHM EXTRA_N(0.1, 5)
```

Listing 4 shows an example of a CCQ with syntax borrowed from [4] where the FROM clause is extended to support sliding windows over data streams. The attributes X, Y, and CITY are attributes of the tuples of the data stream TRUCK_POSITIONS. CONVEX_HULL and COUNT are aggregate functions applied per cluster returning a spatial object and a number, respectively. In the ALGORITHM clause an incremental clustering algorithm is specified, here EXTRA_N [15]. Unlike GROUP BY queries there is no explicit grouping key in clustering queries, which is why the SELECT clause only includes aggregate functions. This query is useful in active safety systems in modern vehicles where the focus is to avoid accidents. In this case, the query returns the boundaries of the congested areas every second to warn the drivers cruising at high speed prior to approaching congested areas.

Each cluster is represented by a system generated cluster identifier. At any point in time in a given window, the output of a clustering query is a set of aggregate objects for each cluster.

Unlike incremental maintenance of aggregate functions in ACQs, the definition of the clustering algorithm addresses group formation, rather than aggregation.

To allow the clustering algorithm to be executed incrementally, the definition of it is broken down into four components: *init*, *add*, *merge*, and *exclude* (Listing 5).

Listing 5: Incremental user defined clustering function

```
init(parms p)->cluster_set initial;

add(data_point d, cluster_set partial, parms p)
   ->cluster_set new_partial;

merge(cluster_set total, cluster_set partial, parms p)
   ->cluster_set new_total;

exclude(cluster_set total, cluster_set partial, parms p)
   ->cluster_set new_total;
```

The above components of a CCQ are distributed and executed over three processing nodes, as illustrated in Figure 4. The thickness of the arrows in the figure indicates relative volume of the stream.

As a preprocessing step, the *selection filter* process applies the selection predicate in the CCQ, typically reducing the stream volume.

Similar to the two level processing of ACQs, the clustering task in CCQs is broken down into two levels. First a *partial clustering* process slices the incoming stream into partial windows, similar to partial aggregation. Thereby the *init* function in Listing 5 creates an initial cluster set for the partial window. For example, in distance based clustering algorithms such as [7] the init function generates initial centroids. Following the invocation of the init function, the *add* function is called for each data point in the partial window to add new data points to the partial cluster set. This will support all single-pass algorithms like BIRCH [18]. The processing of data points in a partial window finishes by sending the partial cluster set to the *final clustering* process.

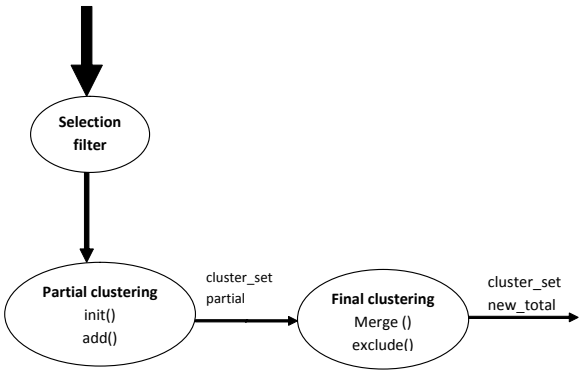

Figure 4. Data flow of a single CCQ

The final clustering process rolls up the consecutive incoming partial cluster sets sent by the partial clustering node to maintain the *total* cluster set corresponding to the whole window, similar to final aggregation in ACQs. This is done by applying the *merge* function on every incoming partial cluster set to update the total cluster set, for example, as done by the merge step in the STREAM algorithm [7]. When the window slides, the *exclude* function is executed to remove the contributions made by the partial cluster sets in the expired partial window. The exclude function is optional, i.e. if a remove algorithm is not specified, the final clustering task merges each delta cluster set with as many windows as it corresponds to. This supports density based approaches like Extra-N [15] where the need for implicit deletion is eliminated.

Notice that this framework is easily data parallelizable at all different stages. For example, if partial clustering becomes the bottleneck, the system can create other instances of it to parallelize the work and distribute the partial windows using, e.g., round-robin [17]. The merge can also be done in parallel using a divide and conquer paradigm in several steps forming a merge tree.

To conclude, the framework is general, extensible, and capable of representing both incremental and deletion elimination methods. It can handle both distance based and density based clustering methods. It is optimizable and parallelizable.

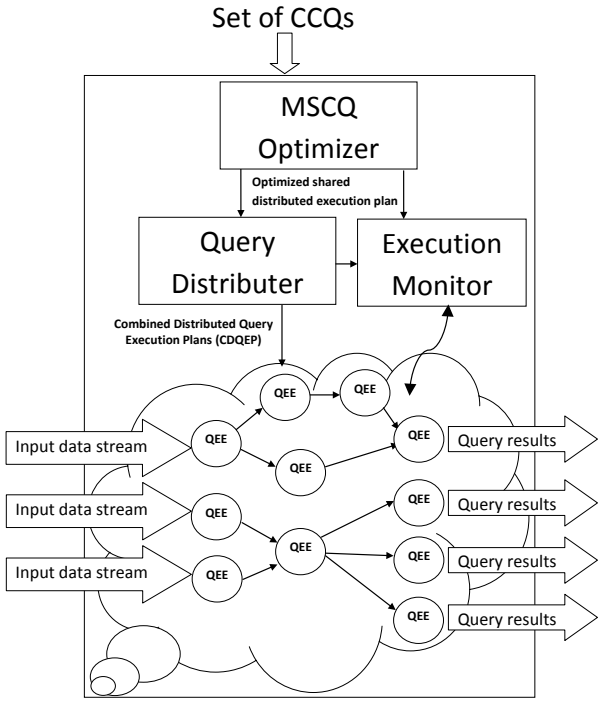

Figure 5. MSCQ system

5. Multiple Stream Clustering Query (MSCQ) Processor

Figure 5 sketches the overall architecture of the proposed MSCQ system to process multiple CCQs. The system receives a set of CCQs applied on input data streams. The MSCQ optimizer produces an optimized shared distributed execution plan for the CCQ set. The *query distributor* sets up combined distributed query execution plans (CDQEPs) by initializing distributed processes and establishing communication links. The components of the CDQEPs are locally executed by a *query execution engine (QEE)* on each processing node. An *execution monitor*

continuously observes CDQEPs to identify bottlenecks and adapts the local plans in the nodes to cure them.

An open problem is how to dynamically modify CDQEPs when queries join or leave. A naive approach is to generate a n ew CDQEP every time a n ew CCQ joins or leaves. More sophisticated approaches would incrementally modify running CDQEPs.

6. Conclusion

We introduced and motivated the problem of optimizing multiple continuous clustering queries (CCQs). We showed its similarities and differences with the well-studied optimization of multiple aggregate continuous queries (ACQs). Based on this, we showed the need for research on e xtensible, distributable multi-query optimization, specifically exploiting all sharing opportunities in multiple CCQs. As first steps, an extensible framework for processing distributed CCQs was presented and the initial system architecture was outlined.

7. ACKNOWLEDGMENTS

This work was supported by the Swedish Foundation for Strategic Research, grant RIT08-0041 and by the EU FP7 project Smart Vortex.

8. REFERENCES

[1] Babcock, B., Mayur, D., Rajeev, M., and O'Callaghan, L. Maintaining variance and k-medians over data stream windows. In *SIGMOD conf.* (San Diego 2003), 234-243.

[2] Badiozamany, S., Melander, L., Truong, T., Xu, C., and Risch, T. Grand Challenge: Implementation by Frequently Emitting Parallel Windows and User-Defined Aggregate Functions. In *Proceedings of Distributed Evenet Based Systems 2013* (Arlington 2013), DEBS 2013.

[3] Badiozamany, S. and Risch, T. Scalable ordered indexing of streaming data. In *Workshop proceedings of the Accelerated Data Management Systems 2012, in conjunction with VLDB 2012* (Istanbul 2012), ADMS Workshop at VLDB.

[4] Chengyang, Z. and Yan, H. Cluster By: a new sql extension for spatial data aggregation. In *Proceedings of ACM international symposium on Advances in geographic information systems* (Seattle, Washington 2007), 53.

[5] Cranor, C., Johnson, T., Spataschek, O., and Shkapenyuk, V. Gigascope: a s tream database for network applications. In *SIGMOD conf.* (New York 2003), 647-651.

[6] Ester, M., Kriegel, H-P., Sander, J., Wimmer, M., and Xu, X. Incremental clustering for mining in a data warehousing environment. In *VLDB conf.* (New York 1998), 323-333.

[7] Guha, S., Mishra, N., Motwani, R., and O'Callaghan, L. Clustering data streams. In *Proceedings of Foundations of Computer Science conference* (Redondo Beach, CA 2000), 359-366.

[8] Jerzak, Z. and Ziekow, H. http://www.orgs.ttu.edu/debs2013/index.php?goto=cfchallengedetails. In *DEBS 2013 Grand Challenge* (2013).

[9] Jin, L., Maier, D., Tufte, K., Papadimos, V., and Tucker, P. A. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD conf.* (Baltimore, Maryland 2005), SIGMOD.

[10] Rui, Z., Koudas, N., Ooi, B. C., and Srivastava, D. Multiple aggregations over data streams. In *SIGMOD conf.* (Baltimore, Maryland 2005), SIGMOD.

[11] S., Krishnamurthy, Wu, C., and Franklin, M. On-the-fly sharing for streamed aggregation. In *SIGMOD conf.* (Chicago, Illinois 2006), SIGMOD.

[12] Sellis, T. K. Multiple-query optimization. (March 1988), Transactions Of Database Systems TODS, 23-52.

[13] Shenoda, G., Sharaf, M. A., Chrysanthis, P. K., and Labrinidis, A. Optimized processing of multiple aggregate continuous queries. In *Proceedings of the 20th ACM international conference on I nformation and knowledge management* (Glasgow 2011), CIKM.

[14] Shenoda, G., Sharaf, M. A., Chrysanthis, P. K., and Labrinidis, A. Three-level processing of multiple aggregate continuous queries. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on* (Hannover 2012), ICDE.

[15] Yang, D., Rundensteiner, E. A., and Ward, M. O. Neighbor-based pattern detection for windows over streaming data. In *EDBT conf.* (Saint Petersburg 2009), 229-540.

[16] Yang, D., Rundensteiner, E. A., and Ward, M. O. A shared execution strategy for multiple pattern mining requests over streaming data. In *VLDB conf.* (Lyon 2009), 874-885.

[17] Zeitler, E. and Risch, T. Massive scale-out of expensive continuous queries. In *VLDB conf.* (Seattle 2011), 1181-1188.

[18] Zhang, T., Ramakrishnan, R., and Livny, M. BIRCH: an efficient data clustering method for very large databases. In *SIGMOD conf.* (Montreal 1996.), 103-114.