Framework for real-time clustering over sliding windows

Sobhan Badiozamany
Uppsala University
Box 337, SE-751 05,
Uppsala, Sweden
sobhan.badiozamany@it.uu.se

Kjell Orsborn
Uppsala University
Box 337, SE-751 05,
Uppsala, Sweden
kjell.orsborn@it.uu.se

Tore Risch
Uppsala University
Box 337, SE-751 05,
Uppsala, Sweden
tore.risch@it.uu.se

ABSTRACT

Clustering queries over sliding windows require maintaining cluster memberships that change as windows slide. To address this, the *Generic 2-phase Continuous Summarization framework (G2CS)* utilizes a generation based window maintenance approach where windows are maintained over different time intervals. It provides algorithm independent and efficient sliding mechanisms for clustering queries where the clustering algorithms are defined in terms of queries over cluster data represented as temporal tables. A particular challenge for real-time detection of a high number of fastly evolving clusters is efficiently supporting smooth re-clustering in real-time, i.e. to minimize the sliding time with increasing window size and decreasing strides. To efficiently support such re-clustering for clustering algorithms where deletion of expired data is not supported, e.g. BIRCH, G2CS includes a novel window maintenance mechanism called Sliding Binary Merge (SBM), which maintains several generations of intermediate window instances and does not require decremental cluster maintenance. To improve real-time sliding performance, G2CS uses generation-based multi-dimensional indexing. Extensive performance evaluation on both synthetic and real data shows that G2CS scales substantially better than related approaches.

Keywords

Data stream processing; Sliding windows; Clustering; Framework

1. INTRODUCTION

In the big data era, the data is produced at extremely high velocities and volumes. Data Stream Management Systems (DSMSs) [1] address these challenges by processing continuous queries (CQs) over streaming data. Examples of data streaming applications are urban traffic monitoring, stock trading, and industrial sensor data monitoring. When the exact grouping of data is unknown, *continuous clustering queries* (CCQs) enable real-time identification of continuously evolving clusters over which statistical summaries are computed as the stream progresses.

Sliding windows are widely used in DSMSs since they enable processing of infinite data streams. In particular, they capture the real-time and evolving behavior of data streams by processing only the most recent data at a given point in time. Here we focus on time-based sliding windows, but the proposed techniques are applicable to count-based sliding windows too.

An example CCQ is given in Listing 1 where a modified version of the clustering algorithm BIRCH for sliding windows, *C-BIRCH,* is used to detect congested areas with radius 50 meters over a window of vehicle positions *X* and *Y*. The window has range 10 minutes and slides every 2 minutes. Given a window, C-BIRCH forms a set of clusters identified by *cid* on which the aggregate functions CENTER and COUNT are applied. In the query the clustering algorithm C-BIRCH is a parameter.

Listing 1: An example Continuous Clustering Query

```
SELECT CENTER(cid), COUNT(cid)
FROM   VEHICLE_POSITIONS (RANGE = 10, STRIDE = 2)
WHERE  SPEED<30
CLUSTER BY X, Y AS cid
USING C-BIRCH(50)
```

To be able to utilize existing clustering algorithms in such queries, a framework where clustering algorithms can be plugged into a DSMS is needed. Such algorithms need to maintain algorithm specific data per cluster where the schema of such data depends on the algorithm. The *Generic 2-phase Continuous Summarization framework (G2CS)* provides an algorithm independent and efficient sliding mechanism for clustering queries, called *sliding binary merge (SBM)*. G2CS simplifies the development of clustering algorithms by defining them in terms of queries over cluster data represented as tables. The system provides transparent algorithm independent multi-dimensional indexing of clustering data.

Figure 1 illustrates how data overlaps when windows slide. A *window instance* $W_{b,e}$ represents the state of the window *W* during the *valid time interval [b,e)*. In G2CS time intervals are represented as object called *contexts*. In Figure 1 the data in window instance $W_{0,10}$, covering the time interval (context) *[0,10)* overlaps (gray boxes) with the data in the window instance $W_{2,12}$ covering *[2,12)*. The window instance $W_{2,4}$ is a common *partial window* instance of the *complete window* instances $W_{0,10}$ and $W_{2,12}$.

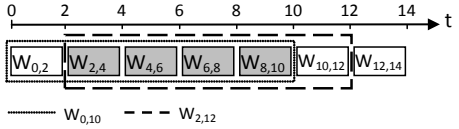


Figure 1. Data overlap in sliding windows

Complex data mining queries may have nested windows, i.e. the queries that form windows on top of other windows. To enable

efficient processing of such queries, it is not sufficient to destructively maintain the latest window instances in-place, but old instances of windows need to be retained as well. Therefore, a generation based window maintenance technique need to be devised where older instances of windows are maintained as long as they are referenced by other windows. This approach also facilitates shared execution plans for queries having different window ranges.

To avoid unnecessary re-computations when data is summarized over sliding windows, efficient *differential maintenance* techniques can be used [2]. Differential processing is usually done by introducing functions for adding/removing deltas to/from the aggregation state [3]. For example the aggregate function COUNT is differential because both of the following equations hold:

$COUNT(A \cup B) = COUNT(A) + COUNT(B)$ *(Incremental)*

$COUNT(A \backslash C) = COUNT(A) - COUNT(C)$ *(Decremental)*

Here A, B, and C are sets.

Aggregation queries over sliding windows are commonly processed in two phases [2] [4] [5] [6] [7]:

1. In the first phase, called *partial aggregation*, fine-grain non-overlapping partial window instances are formed where aggregate data is accumulated.
2. The second phase, called *final aggregation*, combines consecutive aggregates from the first phase to produce the *total aggregate* over the complete window instances.

With the 2-phase window maintenance approach the performance is improved because the incremental property of an aggregate function enables pushing down incremental computations into partial windows in the first phase, thus reducing the data volume in the second phase. Second, the decomposition allows distributed and parallel processing since phase one and two form a pipeline [8]. In the 2nd phase, at every slide, the incremental property enables a partial aggregate to be *merged* into the total aggregate, while the decremental property allows the contributions of expired partial aggregates to be *excluded*.

We note that the 2-phase approach is also beneficial for clustering-algorithms, where expensive cluster formation can be done in phase one and the formed partial clusters are combined using the clustering algorithm in phase two. However, there is a fundamental difference between GROUP-BY queries and clustering queries, which has implications on how the two phase approach is implemented. In GROUP-BY queries, the groups on which aggregate functions are applied are formed based on equality of grouping keys, whereas clusters are formed based on algorithm-dependent similarity between data points. Therefore, a window slide in a GROUP-BY query does not move elements between groups. In contrast, for clustering algorithms the window slides dynamically change cluster memberships as clusters might merge or split when new data arrives or old data expires. This has the following implications on how clustering algorithms are processed over sliding windows, compared to conventional GROUP-BY queries:

a) Streamed clustering algorithms requires grouping and aggregation to be combined, whereas group formation mechanisms in GROUP-BY queries are implemented by first splitting the stream based on the group key in a grouping operator followed by an aggregation operator [2][19]. Thus, each clustering algorithm needs to maintain its own data structures to represent clusters that are updated as the window slides. In order to efficiently support queries that involve nested windows, these data structures need to be retained for different window instances.

b) For many clustering algorithms, incremental deletion of data points from clusters is not defined [9] [10], i.e. they are not decremental. Even when a decremental method can be devised as in [11], it can be very expensive and must be avoided, as suggested by previous work [12].

c) Efficient grouping by similarity in streamed clustering algorithms require multi-dimensional indexing to find which clusters are influenced by a regrouping, while streamed GROUP-BY queries can hash on fixed group keys.

In this paper we present the Generic 2-phase Continuous Summarization (G2CS) framework, with the following main contributions to the state-of-the-art methods:

1. To address a) G2CS relies on a query language for modeling the clustering algorithms. Contexts are used for allowing clustering algorithms to store multiple generations of summarization data as the cluster memberships evolve over time with window slides, described in Section 3.1.
2. To address b) G2CS maintains and reuses several intermediate window instances by organizing them using contexts and analyzing their temporal dependencies as a lattice called the *SBM-lattice*, described in Section 3.2.
3. To address c) G2CS provides a method for transparent multi-dimensional indexing of the contents of each window instance, called *contextualized indexing*, described in Section 3.3.

The rest of the paper is organized as follows. Section 2 defines the basic concepts and terminology used in the paper. Section 3 presents the G2CS framework and the three main contributions. The approach is exemplified by adapting the well-known BIRCH clustering algorithm [9] for real-time stream clustering, called C-BIRCH, which is another contribution. The experimental evaluation in Section 4 uses synthetic and real data to measure the performance of the proposed methods, showing significant improvements over previous work, while retaining clustering quality. Section 5 discusses related work and Section 6 concludes and proposes some future research directions.

2. PRELIMINARIES

In this section the basic concepts that are used throughout the paper are briefly reviewed.

Repetitive merge, RM

To support non-decremental clustering algorithms, in previous approaches [10] [12] [13] the summary in each partial window instance, here called *Partial Grouped Summary (PGS)*, is *repetitively merged* into all complete windows it is part of. The repetitive merge (RM) is illustrated in Figure 2 where a sliding window of range R=10 and stride S=2 is formed in the 2nd phase.

When PGS_5 arrives, it is merged into the five complete window instances $W_{0,10}$, $W_{2,12}$, $W_{4,14}$, $W_{6,16}$, and $W_{8,18}$. This causes redundant computations, e.g. both $W_{8,18}$ and $W_{10,20}$ merge all the common partial summaries $PGS_6 - PGS_9$. The *partition ratio PR* of a window is defined as:

$$PR = \frac{R}{S}$$

For example, in Figure 2 PR=5. Scaling up PR is important to track fast changing clusters with fast concept drifts in real-time. With the repetitive merge approach, maintaining the sliding windows in the 2nd phase becomes expensive when PR is large.

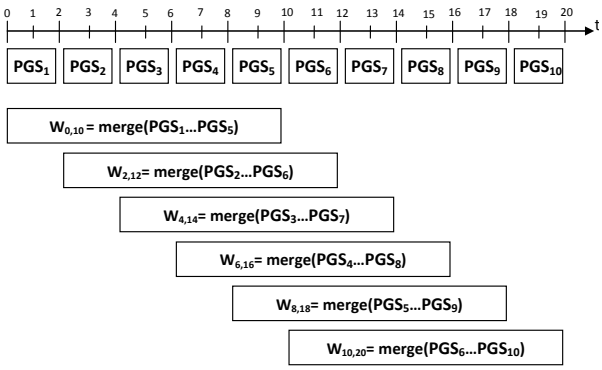


Figure 2. Final summarization with Repetitive Merge

<u>2-phase decomposition of clustering algorithms</u>

Suppose that a clustering function C over a dataset DS is computed using two components: first a "partial-clustering" function $P(DS_i)$->CS_i computes cluster summaries CS_i over the disjoint data sets $DS_1, DS_2, \ldots, DS_n$ where $DS = \cup_{1 \le i \le n} DS_i$. In the 2nd phase a binary *merger* function M (CS, CS) ->CS is repeatedly applied by an *orchestrator* function F to combine the output from all $P(DS_i)$ in some order:

$$C(DS) = F(M, \{P(DS_i) \mid 1 \le i \le n\})$$

In G2CS the functions P and M are algorithm dependent plug-ins while F is an optimization strategy for sliding windows that G2CS executes.

We call such a clustering function C, represented by the combination of functions M, P, and F an *incremental* clustering function.

There are incremental variants of both density based and centroid based clustering algorithms. For example, DBSCAN for data warehouses [11] incrementally merges batches of data points into a database of clusters while *STREAM* [10] incrementally merges disjoint subsets of datasets that are pre-clustered using K-means.

Now, assume that there are two datasets DS_1 and DS_2 such that $DS_1 \subseteq DS_2$. An incremental clustering function C is *differential* if it is also *decremental*, i.e. there exist an *exclude* function $EX(CS,CS)$ ->CS that removes the contributions made by expired partial clustering of CS_1 from CS_2:

$$C(CS_2 \setminus CS_1) = EX(CS_2, CS_1)$$

Overall, most of the clustering algorithms are not decremental. G2CS provides an efficient real-time sliding mechanism for such non-decremental clustering algorithms.

3. GENERIC 2-PHASE CONTINUOUS SUMMARIZATION FRAMEWORK

The G2CS framework generalizes the 2-phase aggregate function processing frameworks over sliding windows [2] [4] [5] [6] [8] to process continuous clustering queries (CCQs) by separating the sliding and indexing mechanisms from the plugged-in summarization algorithms.

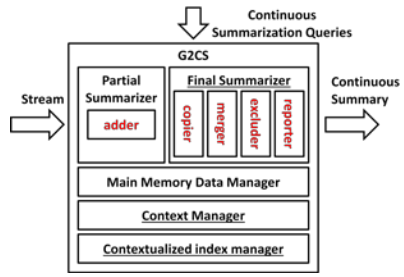


Figure 3. Generic 2-Phase Continuous Summarization framework

Figure 3 illustrates the architecture of G2CS. The contributions of this paper are the modules that are underlined in the figure.

Unlike aggregate functions where the operator state is simple due to separation of grouping from aggregation, clustering algorithms require maintaining complex and algorithm-dependent relationships between data points in order to continuously maintain the evolving clusters. To enable high-level modeling of such clustering state information for the plugged-in algorithms G2CS provides a built-in *Main-Memory Data Manager* having full query processing and indexing capabilities over a local database.

The algorithm developers define summarization algorithms using a number of plug-in functions, marked as red in Figure 3, that modularize the clustering algorithms and separate them from the sliding mechanism.

Window instances may be referenced from other objects. For example, they can be saved in the local database or several windows specified over the same stream may cause the same window instance to belong to more than one window specification. This requires generation based window management where the system retains window instances as long as they are referenced from other objects.

The *context manager* organizes window instances by contexts. A context is represented as a triple <*b, e, cxtid*> where *cxtid* is a unique *context identifier* of the time interval *[b,e)* per window. Contexts enable non-destructive updates of window instances. They are allocated by the context manager and their identifiers are passed to the plugged-in clustering algorithms. The *contextualized index manager* in G2CS maintains an index per context in the local database and separates indexing from the sliding mechanism and the plugged-in clustering algorithm.

The plug-ins functions are called by the partial and final summarization phases as follows.

The *partial summarizer* implements the first phase of clustering over sliding windows. As new data arrives, it slices the incoming stream into partial window instances. It assigns a new context for each new partial window instance and then iteratively calls the

adder plug-in for each arriving data point to incrementally populate summary data for the context identifier.

When the summary data for the partial window instance is fully populated the *final summarizer* is called, causing the sliding mechanism to be invoked in order to form and emit the clusters in a complete window instance. It implements the second phase of the clustering and is the focus of this paper. For differential algorithms the user can provide methods for both incremental (*merger* plug-in) and decremental (*excluder* plug-in) maintenance of clusters. When there is no excluder for a clustering algorithm, the final summarizer minimizes redundant computations by analyzing dependencies between different contexts.

G2CS internally maintains a number of intermediate window instances to optimize the sliding mechanism. In order to populate new window instances the *copier* is invoked to copy data from old to new window instances. Then G2CS makes a number of calls to the *merger* and *excluder* plug-ins to generate complete window instances. By calling the copier calls prior to the merger and excluder, G2CS retains old window instances. The *reporter* plug-in extracts the data to be emitted from a complete window instance.

An incremental garbage collector deallocates summary data for window instances whose contexts are no longer needed.

3.1 Context Management

In this section we explain how the state of a clustering algorithm is represented using contexts and how the decomposed clustering algorithms operate on contexts. We have adapted the well-known BIRCH [14] algorithm for sliding windows, called C-BIRCH. C-BIRCH is used as a running example of a 2-phase non-decremental clustering algorithm. We first briefly describe BIRCH.

3.1.1 The BIRCH Algorithm

The BIRCH clustering algorithm provides an approximate K-means computation in a single pass over the original dataset. It builds a spatial summary of the read multi-dimensional data points in main memory represented as micro clusters characterized by a *Cluster Feature vector*, *CF-vector*. The CF-vector contains three items summarizing the data points in a micro-cluster: their linear vector sum *ls*, their squared vector sum *ss*, and the number of points *c* in the micro-cluster. BIRCH builds this summary by loading the data points into a B-tree-like spatial indexing structure called a *CF-tree*.

The CF-tree maintains a hierarchy of clusters. The coarsest cluster is the root node, while the leaf nodes contain the most fine grain clusters, i.e. the micro-clusters. The CF-tree is constructed by adding the data points one by one as follows. For an incoming data point, *dp*, first the closest micro-cluster, *cf-near* is located by recursively selecting the closest cluster while the CF-tree is traversed. It then tests whether *dp* can be added to (absorbed by) *cf-near* without violating a pre-specified maximum allowed radius of a micro-cluster *r*. If absorbing *dp* into *cf-near* does not make its radius larger than *r*, then *dp* is added to *cf-near*, otherwise, a new micro-cluster containing only *dp* is created in the same leaf node of the CF-tree where *cf-near* is stored. The CF-vectors on the path from the root to the leaf are adjusted to reflect the addition of *dp* in the tree.

As BIRCH is sensitive to the arrival order of data points, an optional second pass significantly reduces the order dependence, making it a 2-phase clustering algorithm. There, all the CF-vectors in the leaf nodes are accessed to populate a second CF-tree as follows. For each CF-vector *CFV*, first its center of mass *cm* is calculated from its *ls*, *ss*, and *c*. Then *cm* is used to lookup the second CF-tree to find the *nearest* cluster to *CFV* and *ls*, *ss*, and *c* of *CFV* are added to those of the nearest cluster.

The final step of BIRCH scans the second CF-tree and applies a global clustering algorithm (e.g. K-means) on the micro-clusters in its leaf nodes.

BIRCH is non-decremental since deletion of an expired point, *ep*, is meaningless. This is because more data points are added after the addition of *ep*, potentially moving the center of the micro-cluster of *ep*. Therefore when *ep* expires, the closest micro-cluster to it is not necessarily the one it contributed to, and hence there is no guarantee that deletion of *ep* cancels out the effects of its addition.

C-BIRCH fully supports streaming K-means but does not maintain the hierarchy of clusters maintained inside the CF-tree in BIRCH.

3.1.2 The Contextualized Clustering Table

Clustering algorithms associate summaries with cluster identifiers. In order to adapt a clustering algorithm for sliding windows, we need to associate algorithm specific data of each cluster identifier with different window instances. In the G2CS framework information about clusters are stored in a *contextualized clustering table*, *CCT* with the schema:

$$CCT(\underline{cid}, cxtid, a_1,\ldots a_n)$$

Here *cid* is a cluster identifier, *cxtid* is the context identifier for the valid time of *cid*, and $a_1,\ldots a_n$ are algorithm-dependent summary information about *cid*.

CCT for C-BIRCH

Rather than a data representation that is highly integrated as the CF-tree in BIRCH, CCT provides a general representation of clustering data that can be indexed independently. The CCT of C-BIRCH has the schema:

$$CCT\text{-}BIRCH(\underline{cid}, cxtid, cm, ls, ss, c)$$

The center of the micro-cluster *cm* is a vector computed by the adder and merger as $cm = ls / c$. By explicitly storing *cm* in CCT-BIRCH it can be indexed by a multi-dimensional index to facilitate finding the nearest micro-cluster, as will be explained later.

3.1.3 Plug-in Definitions

The G2CS plug-ins signatures are presented in Listing 2. We will exemplify them by sketching the plug-in definitions for C-BIRCH.

Listing 2: G2CS decomposition of clustering algorithms

```
adder (Integer c_partial, Vector dp, Object p)
copier (Integer c_org, Integer c_dest, Object p)
merger (Integer c_incoming, Integer c_res, Object p)
excluder (Integer c_expired, Integer c_res, Object p)
reporter (Integer c_rep, Object p) -> Set of (Number cid, Object summary)
```

C-BIRCH adder

The C-BIRCH adder receives at every call from G2CS a context identifier *c_partial* representing the current partial window instance *pwi*, a data point *dp*, and an algorithm-specific parameter(s) *p* (the maximum radius *r* in BIRCH). First the closest micro-cluster *cf_near* is found by running a nearest neighbor query over the CCT-BIRCH table where *cxtid* = *pwi*.

Then the adder checks whether *cf_near* can absorb the new data point; otherwise a new micro-cluster object is added to CCT-BIRCH for context *pwi*.

C-BIRCH copier

The C-BIRCH copier makes a copy of the rows in CCT-BIRCH where *cxtid = c_org* and assigns *cxtid = c_dest* to the copied rows.

C-BIRCH merger

As shown in Listing 3, the C-BIRCH merger receives from G2CS the two context identifiers *c_incoming* and *c_res*. It merges the CCT-BIRCH rows where *cxtid=c_incoming* into *c_res*. For each micro-clusters *mc* in *c_incoming*, *cf-near* is found by a nearest-neighbor search over the CCT-BIRCH rows where *cxtid=context_res*. If *cf-near* can absorb *mc*, its row is updated otherwise a new row representing *mc* is added to CCT-BIRCH for context *c_res*.

C-BIRCH reporter

Given a context identifier *c_rep*, the values of *cm*, and *c* of all the micro-clusters in CCT-BIRCH are directly emitted. An optional post-processing step is to apply a global clustering algorithm, e.g. K-means.

There is no C-BIRCH *excluder* plug-in since BIRCH is not a differential algorithm and therefore G2CS uses SBM to minimize the number of merger calls.

Listing 3: C-BIRCH merger plug-in

```
CBIRCH-merger(Integer c_incoming, Integer c_res, Number r)
{
  for each ROW mc in (
  select cid, c_incoming, cm, ls, ss, c
  from CCT-BIRCH where cxtid = c_incoming)
  {
    cf-near= nearest(c_res, mc.cm)
    if can-absorb(cf-near, mc.cid, r)
      update table CCT-BIRCH set
        cm = cm + mc.cm,
        ls = ls + mc.ls,
        ss = ss + mc.ss,
        c  =  c + mc.c
      where cid = cf-near.cid;
    else
    insert into CCT-BIRCH values
      (new_cid(), c_res, mc.cm,
      mc.ls, mc.ss, mc.c);
  }
}
```

BIRCH is generally sensitive to the order of the data points, but as explained by the authors of BIRCH [14], the second pass alleviates the sensitivity since the first CF-tree has captured most of the locality of the data. Similarly, by building a global CCT-BIRCH table in the final summarizer, C-BIRCH is less vulnerable to order sensitivity. In the experimental section we compare the quality of C-BIRCH clustering with a baseline BIRCH recomputed over each window instance, indicating that they have comparable quality, while C-BIRCH is substantially faster for large *PR* to track fast concept drift.

3.2 Sliding Binary Merge

SBM avoids the overlapping merges of RM by generating and retaining intermediate window instances incrementally based on analyzing the SBM lattice. The lattice illustrates how SBM works; it is not explicitly stored in G2CS.

The SBM lattice represents temporal relationships between window instances in terms of their time intervals, i.e. contexts. We illustrate it with the example in Figure 4. Suppose that we want to cluster the data over a sliding window *W* with range *R=64* seconds and stride *S=4*. In this case, the partial summarization phase performs partial clustering over tumbling windows of *S*=4, and the final summarization phase combines *PR=R/S=16* consecutive PGSs to form a complete window instance. To simplify the discussion we first assume that *PR* is a power of two, which will be relaxed later. The SBM-lattice in Figure 4 has *R=64* and *S=4*. Assuming time *t* starts from 0, each node in the lattice in the figure represents a window instance $W_{b,e}$.

The nodes at the leaf level *L*=0 of the SBM-lattice represent the successive incoming PGSs, $W_{0,4}$, $W_{4,8}$,…,$W_{b,b+S}$ each *S=4* time units long. The root level nodes (*L*=4) represent complete sliding window instances over *R=64* time units. Each intermediate node combines summary information from two nodes with *non-overlapping* time spans forming a *contiguous* interval, as indicated by the arrows in Figure 4. The number of levels in the lattice is 5, in general $log_2(PR) + 1$. Each arriving PGS at a leaf triggers a cascade of merges down the lattice. First, $W_{0,4}$ arrives to the final phase at *t=4* and becomes the leftmost leaf node in the figure. When $W_{4,8}$ arrives at *t=8*, it is combined with $W_{0,4}$ to produce level *L=1* window instance $W_{0,8}$ covering the first 8 seconds. When $W_{8,12}$ arrives at *t=12* it is combined with $W_{4,8}$ into $W_{4,12}$, and this process is continued for each arriving PGS. In general the range of level *L=1* window instances are $2{\cdot}S$, while their strides are still *S*.

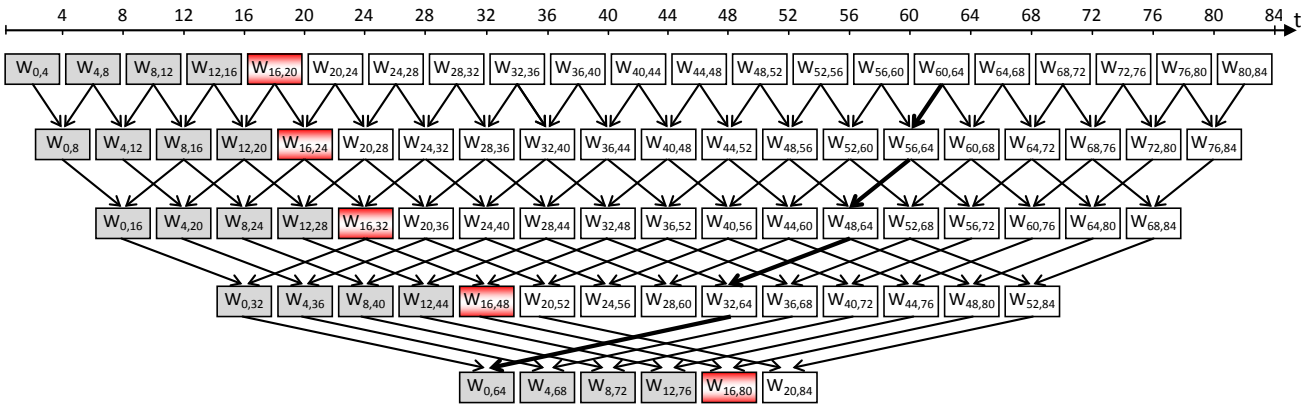


Figure 4. Sliding Binary Merge (SBM) dependencies

For example, when $W_{12,16}$ arrives, it is first combined with $W_{8,12}$ to form the level $L=1$ sliding window instance $W_{8,16}$ and then $W_{8,16}$ is combined with $W_{0,8}$ to form the level $L=2$ sliding window instance of range $4 \cdot S$, $W_{0,16}$.

The cascading merges continue until complete window instances of size $R=64$ are formed on level $L=4$ (the SBM-lattice root) where sliding window instances of range $16 \cdot S$ are represented. The first final window is not formed until $W_{60,64}$ arrives and triggers four cascading merges as indicated by the bold arrows.

In general *SBM-level* L in the SBM-lattice represents a sliding window of range $S \cdot 2^L$, where $L \in \{0, 2, \ldots log_2 PR\}$. The stride is always S. For the highest L the range becomes $S \cdot PR = R$.

The sliding window mechanism continuously identifies *expired* nodes, i.e. the nodes that are not going to be combined with any new node in the SBM-lattice. These nodes represent window instances that can be passed to the garbage collector. At every slide, the oldest node in each level, that is the *left-most* node, is identified as expired. For example, when $W_{64,68}$ arrives the complete window instance $W_{0,64}$ slides into $W_{4,68}$. Thereby, $W_{0,4}$, $W_{0,8}$, $W_{0,16}$, $W_{0,32}$, and $W_{0,64}$ are no longer needed in any new node, and can be released. In general, when the complete root level window instance slides, the left-most *expired path* in the lattice is expired. For each slide the expired path moves to the right, so when $W_{80,84}$ arrives and the red-marked $W_{16,80}$ expires, all the red-marked nodes in the figure are expired, while the gray-shaded nodes expired earlier. In general, when the complete window instance $W_{b,b+R}$ slides S units into $W_{b+S,b+R+S}$, all $log_2(PR)$ $+1$ windows $W_{b,*}$ are expired.

It can be noticed that more nodes in Figure 4 can be identified as expired, for example, at $t=64$, all level 0 nodes to the left of and including $W_{52,56}$ are no longer needed. Identifying them improves the memory footprint of SBM.

3.2.1 Computational Complexity and Memory Consumption

The number of cascading merges per incoming PGS increases during the initialization period until the first complete window is formed at time $t=R$. In Figure 4 this happens when $W_{60,64}$ arrives. After the initialization period, since the size of the sliding window doubles at each level, each PGS triggers exactly $log_2 PR$ merges. Therefore the number of merges required to build a complete window instance using SBM is also $log_2 PR$. By contrast, for RM (Figure 2), each PGS is combined with PR complete windows at each slide.

So far we have considered only the costs in terms of the number of merges per slide. However, the sliding involves several operations in addition to the merge, even though the algorithm dependent merger plug-in is usually the dominating cost. In both RM and SBM a single slide involves iteratively calling the copier and merger plug-ins for each level. Let CC be the average copier cost and MC be the average merger cost. We define the cost of one iteration IC as:

$$IC = CC + MC$$

CC depends on the data volume and on the window maintenance method. Therefore, we subscript their costs with RM or SBM, respectively, e.g. CC_{RM}. MC is independent of the window maintenance method, but is very much dependent on the data mining algorithm, where e.g. indexing can play a key role, as will be discussed later.

The total cost of a slide for RM is defined as

$$TC_{RM} = PR \cdot (CC_{RM} + MC)$$

An iteration in RM always involves applying the copier and the merger plug-ins on a complete window instance, so there will be PR iterations per slide. Let AC be the average number of clusters in the complete window instances. We assume that CC is proportional to AC. Furthermore, without loss of generality we set its proportion to one. Then TC_{RM} becomes:

$$TC_{RM} = PR \cdot (AC + MC)$$

Unlike RM where all iterations involve complete window instances, the intermediate nodes in the SBM-lattice cover shorter ranges, and therefore they should contain fewer clusters than AC. The number of iterations in SBM is $log_2 PR$. Let CC_L denote the copier cost at level L of the SBM-lattice. Assuming the intermediate nodes in the SBM-lattice contain fewer clusters than the complete window instances, we can overestimate the average merger costs at all levels as MC, thus:

$$TC_{SBM} = \sum_{L=0}^{log_2(PR)-1} (CC_L + MC)$$

Assuming CC_L is proportional to the window instance range, we get:

$$CC_L = 2^L \cdot \frac{AC}{PR}$$

$$TC_{SBM} = \sum_{L=0}^{log_2(PR)-1} 2^L \cdot \frac{AC}{PR} + \sum_{L=0}^{log_2(PR)-1} MC$$

$$= AC - \frac{\mathrm{AC}}{PR} + MC \cdot \log_2 PR$$

Real-time clustering becomes computationally expensive when the granularity PR or the number of generated clusters AC are scaled up. Therefore PR and AC are the significant terms in our performance analysis.

Since $PR > 1$ for any sliding window, it always holds that $TC_{SBM} < TC_{RM}$. Furthermore, the costly MC is multiplied by $log_2 PR$ in SBM in contrast to being multiplied by PR in RM. Thus SBM scales better than RM with finer sliding granularity. Furthermore, notice that the overall copy cost in RM is $AC \cdot PR$, while it is less than AC for SBM. Thus SBM will scale better than RM with the number of clusters.

<u>Memory footprint</u>

We use the total number of clusters in the final node as an implementation independent measurement of the memory footprint. Since PR complete window instances are maintained for RM, the total memory footprint is $AC \cdot PR$.

The total memory footprint of SBM is the sum of the number of clusters in the window instances at each level L. Assuming that the number of clusters in a window instance is linearly proportional to the range of its window, the number of clusters in a window instance at level L is $2^L \cdot \frac{AC}{PR}$. Since there are $PR - 2^L + 1$ active nodes (white nodes in Figure 4) at level L the total memory footprint of SBM becomes:

$$\sum_{L=0}^{log_2 PR} 2^L \cdot \frac{AC}{PR} \cdot (PR - 2^L + 1) = \frac{2}{3} \cdot AC \cdot PR + AC - \frac{2 \cdot AC}{3 \cdot PR}$$

The memory footprints of both SBM and RM are *O(AC·PR)*, but with different constant multipliers

3.2.2 Sliding N-ary Merge Lattices

The SBM-lattice could be generalized to represent N-ary merges at each level, which we call a *sliding N-ary merge (SNM)* lattice. In an SNM-lattice at each level *L*, *N* nodes from level *L-1* having non-overlapping time spans are merged. Therefore the intermediate nodes at level *L* maintain window instances with a range $PR \cdot L^N$. *N* is called the *fan-in* of the SNM-lattice.

Lemma 3.1: The optimal fan-in for an SNM-lattice w.r.t. the total number of merges per slide is *N=2*, i.e. the SBM-lattice is optimal.

Proof: the number of merges per slide for an SNM-lattice that maintains a window with partition rate *PR* is

$$f(N) = (N-1) \cdot \log_N PR$$

This is because at each level *N-1* merges are performed.

Since $\frac{d(f(N))}{dN} = \frac{(N-1) \cdot \ln PR}{\ln N}$ is positive for *N≥2*, $f(N)$ is monotonically increasing, and, since in an SNM-lattice *N≥2*, *N=2* provides the minimum total number of merges.

3.2.3 Supporting Windows of Arbitrary Sizes

The example in Figure 4 covered the case where *PR* is a power of two. When *PR* is not a power of two, we define PR_{aux} as the highest power of two less than *PR*, i.e. $PR_{aux} = 2^{\lfloor log_2 PR \rfloor}$. For example, for *R=84*, *S=4*, and *PR = 21*, $PR_{aux}=16$. SBM uses a *base* lattice for PR_{aux}. In order to maintain the full range for *PR*, SBM needs to use and *extended* lattice where additional $PR\text{-}PR_{aux}$ nodes are retained in the root level of the base SBM-lattice. For example, the snapshot of the base lattice for *PR=21* at *t=84* is illustrated by the white nodes in Figure 4 plus all root nodes.

The problem is now: what windows in the extended SBM-lattice should be combined in order to maintain the complete sliding window when *PR* is not a power of two?

The first sub-problem is: what subset of levels in the extended SBM-lattice, $LS \subseteq \{0,1,2,\ldots,log_2 PR\}$, should be selected for combination? The constraint is that the sum of the ranges of the selected levels should add up to *R*:

$\sum_{L \epsilon LS} S \cdot 2^L = R$ i.e. $\sum_{L \epsilon LS} 2^L = PR$

To minimize the number of selected levels and thus minimize the number of merges, *LS* can be obtained by the bits in the binary representation of *PR*. For example, for $PR= 21 = (10101)_2$, *LS={0,2,4}* since $2^0+2^2+2^4=21$.

The second sub-problem is how to form the sliding window instances for *PR* at a slide at time *t*, i.e. what window instance from each level in *LS* should be selected at *t*? The concatenation of the valid time interval of the selected window instances to combine should be equal to the valid time of the complete window instance at time *t*, *[t-R,t)*. In our example, at time *t=84* we need to combine a number of window instances from the extended SBM-lattice to form $W_{0,84}$.

As mentioned in Sec. 2, G2CS assumes that the order in which the merge function is applied does not matter, therefore, the levels in *LS* can be chosen in any order, so G2CS selects them in decreasing level order. The first window instance chosen is the window starting at time *t-R* at the highest level L_0 in *LS*, i.e. $[t\text{-}R, t\text{-}R+ S \cdot 2^{L0})$. The valid time interval of next selected windows at level L_i starts when the previous at level L_{i-1} ends. In general a series of window instances $W_{b,e}$ at level L_i are selected where

$b_0 = t - R$, $e_i = b_i + S \cdot 2^{L_i}$ and $b_i = e_{i-1}$

In the example, the first selected window for *t=84*, *R = 84*, *S=4*, and $L_0=4$ is $W_{0,64}$. Then $L_1 = 2$ so $W_{64,80}$ is chosen. Finally $L_2 = 0$ so $W_{80,84}$ is chosen. This covers the entire valid time interval of complete window instance at time *t=84* since *[0,84) = [0,64) + [64,80) + [80,84)*.

So far we showed what levels in the extended SBM-lattice need to be used to compute *PR* and what window instances from each level should be picked at a given time. In order to form sliding window instances of size *PR*, we add an *extra auxiliary root node* to the extended SBM-lattice that combines the intermediate nodes as explained above. Since three bits were set in *LS* for this example, we have two additional merges in the auxiliary root node. The number of additional merges in the worst case is $log_2 PR$, which occurs when all bits are set. Therefore the total number of merges per slide is still $O(log_2 PR)$.

The memory footprint of using extended SBM-lattice is only slightly higher than for the base SBM-lattice since only $PR\text{-}PR_{aux}$ extra nodes are retained at the root level, so the overall memory footprints of the two are very similar.

The expired nodes in all levels of the extended SBM-lattice are identified using the same method as before, with the exception of the root level where $PR\text{-}PR_{aux}$ most recent nodes are retained.

3.3 Contextualized Indexing

Since clustering algorithms form clusters based on similarity between objects, they need to perform k-NN and proximity queries over multi-dimensional feature vectors. With G2CS these feature vectors are stored in some CCT attribute. For example, the function *nearest(Integer cxtid, Vector qp)* in C-BIRCH finds the micro-cluster that has context *cxtid* and whose *cm* is closest to the feature vector *qp*. In order to efficiently support such a query, indexing is needed on the attribute in CCT storing *qp*. In G2CS this is implemented by partitioning CCT based on context identifiers and having a separate multi-dimensional index per partition.

Figure 5 shows how such a *contextualized index* for a multi-dimensional attribute a_i in the CCT is represented. It is a two level secondary index where the first level is a hash table indexed on *cxtid* and the second level is a main-memory multi-dimensional index for each context. The key to the hash table in Figure 5 is *cxtid*, and the value is the address of a multi-dimensional index, currently a main-memory X-tree [15] plugged-in to our main-memory database engine using MEXIMA [16]. The key to the X-tree is a feature vector attribute a_i in the CCT and the value stores the primary key of the CCT, i.e. *cid*.

For the contextualized index of CCT-BIRCH the key to the X-tree is *cm*. To process the *nearest()* function, first the hash table is looked up with the key *cxtid*, returning the address to the X-tree corresponding to the *cxtid*, after which a nearest neighbor search of the X-tree is performed. Assuming that the X-tree provides the nearest neighbor in logarithmic time and the hash table look up takes constant time, the complexity of the merger plug-in in Listing 3 of I-BIRCH becomes reduced from $O(AC^2)$ without indexing to $O(AC \cdot \log AC)$, i.e. *MC* is substantially reduced.

However, indexing window instances increases the copy cost *CC* since the index data structure need to be built. In particular, since

RM involves large amounts of copying, the gain by indexing is expected to be lower for RM compared to SBM where less data is copied.

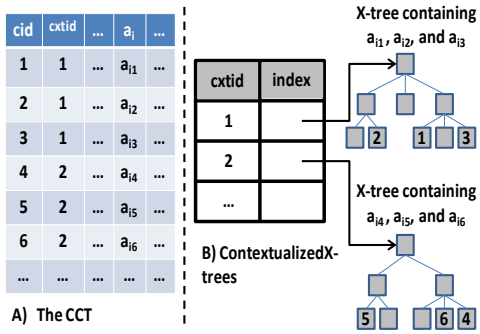


Figure 5. Contextualized multi-dimensional index

If the X-trees were not organized by context identifiers as in Figure 5, an alternative is a secondary index on only *cxtid*, which would make the cost of *nearest()* proportional to the number of clusters in each context.

Notice that a global X-tree on the a_i attribute would not facilitate execution of the nearest neighbor query for a given context identifier because the nearest neighbor returned by looking up a global X-tree is not guaranteed to be in the given context.

The contextualized indexing supports regular GROUP BY queries as well. The difference is that in Figure 5 instead of multi-dimensional X-tree indexes, a conventional hash table can index the groups in each context. In general any indexing structure required by the summarization algorithm can be plugged-in to our main memory database system [16] and used for contextualized indexing.

4. PERFORMANCE EVALUATION

To evaluate the performance of SBM we scale the window slide granularity *PR* by varying the window range *R* and keeping the slide *S* constant. We implemented Repetitive Merge (RM) as baseline algorithm. Contextualized indexing is evaluated by scaling *AC* while keeping *R* and *S* constant. As baseline, we compare it to using a secondary index only on the *cxtid* attribute of CCT.

Table 1 lists the four alternative approaches used in the experiments to combine sliding and indexing alternatives.

Table 1. Sliding window maintenance alternatives

indexing / Sliding approach	Contextualized Indexing (CI)	No Contextualized Indexing (NCI)
RM	RM-CI ●	RM-NCI ◆
SBM	SBM-CI ▲	SBM-NCI ■

The experiments report the average window sliding time in the final summarization phase. We omit the partial phase since it added less than 12% overhead to the total cost in all experiments and is not important when scaling *PR* or *AC*. The experiments thus start by loading pre-calculated partial summaries into main memory from which the window maintenance mechanisms read a stream of PGSs.

For each window size, the window maintenance algorithms are executed over the stream of PGSs. Every arriving PGS triggers a slide, so many slides are performed for a given window size. The average sliding time is shown in the diagrams and the corresponding standard deviation is reported for each experiment.

<u>The data streams</u>

We use both a synthetically generated and a real data stream in our experiments. The reason for using synthetically generated data is to have controlled experiments where only one parameter is scaled at the time. The real data stream is used to show how much difference the proposed methods make in practice, when both *PR* and *AC* are scaled.

The synthetic data streams for experiments involving clustering generates well separated 2D clusters randomly placed around the center of a number of cells in a square grid. Its schema is *SDC(ts,x,y)* where *x* and *y* are the 2D coordinates of some object at time *ts*. The data in SDC is generated as follows. Given the size of a grid cell *c_size* and the number of cells per dimension *nc*, the data stream generator randomly picks cells and randomly generates coordinates within a constant distance of *c_size/5* from the cell centers to guarantee one cluster per cell. The number of clusters is controlled by varying *nc* and *c_size*.

The real data stream is from the DEBS2013 Grand Challenge [11] with schema *DEBS(ts,pid,x,y,z)*, recording the 3D coordinate *x/y/z* of soccer player *pid* at time *ts* in a real soccer match. Notice when *PR* is scaled in this data stream, *AC* varies as well.

The stream rate was not scaled since it influences only the partial summarization phase, which is independent of our contribution. However, the number of clusters is scaled.

We ran our experiments on a HP PC with the following specifications: CPU: Intel Core i5 – 760 @ 2.80 GHz, RAM: 4 GB, OS: Windows 7 64-bit.

4.1 Conventional GROUPBY Queries

As a first simple illustration of the trade-offs between different window maintenance alternatives we compare the performance of RM and SBM when the summarization algorithm is conventional GROUPBY aggregation with COUNT. In this case we also implemented differential maintenance, DM, since we are interested in measuring the performance of SBM compared to DM for the simplest possible grouped aggregate function. For this experiment the stream has the schema *SDG(ts,gk,mv)*, where *mv* is the measured value for the group key *gk* at time *ts*. *gk* is a random integer in range *[1,20]*. The *gk* value range is chosen to ensure that *AC* is constantly 20 while *PR* is scaled. Contextualized indexing is not used as its impact has the same factor for all methods when *AC* is constant.

The experiment in Figure 6 scales *PR* in the following query by increasing *R*:

```
SELECT gk, COUNT(*)
FROM   SDG (Range = R, Slide = 1sec)
GROUP BY gk
```

As expected, RM slows down proportional to the size of the window, DM remains constant, and SBM_NCI slows down logarithmically. Since the cost of RM increases very quickly compared to the other two methods its performance is shown only when $PR \leq 50$. The standard deviation was below 2%.

It is observed that the performance of SBM is not that different from differential maintenance, making it very close to the ideal when DM is not possible. Since SBM performs slightly better for values of *PR* that are powers of two there are small dips there (Sec

3.1.3).

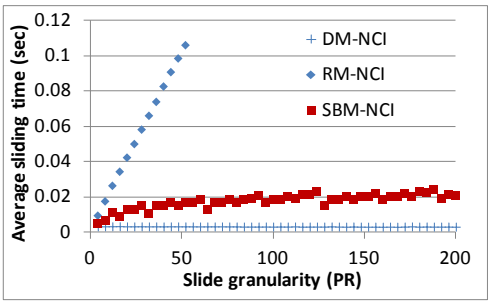


Figure 6. conventional GROUPBY over SDG stream.

4.2 Non-indexed C-BIRCH

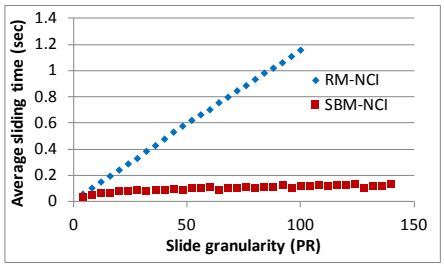


Figure 7. C-BIRCH over SDC stream.

This experiment shows the performance of SBM without contextualized indexing for clustering with C-BIRCH when *PR* is scaled by increasing *R* over the synthetic data stream SDC:

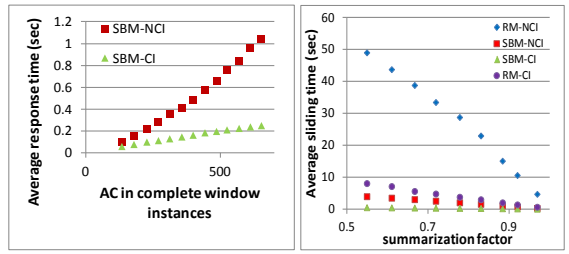
```
SELECT Center(cid), Radius(cid), COUNT(cid)
FROM   SDC (Range = R, Slide = 1sec)
CLUSTER BY X, Y, Z
USING C-BIRCH(Radius = 5 meters)
```

The number of micro-clusters in the window is kept constant by using a rather small grid with *nc=5*, and *c_size=10*. Since all the cells are present in all partial window instances, every complete window instance contains all 25 micro-clusters. The radius parameter of BIRCH is chosen as 5 since diameters of the cells are *c_size=10*. The standard deviation for RM-NCI was between 1% and 3%, while it was between 2% and 5% for SBM-NCI.

Figure 7 shows that RM slows down linearly to the size of the window while SBM scales logarithmically, as expected. The response time of SBM_NCI stays below 0.2, while it increases substantially for RM_NCI, making RM_NCI not suited for real-time clustering. With RM, if the sliding time goes above the stride being 1s, the system will start lagging.

4.3 Contextualized indexing of C-BIRCH

This experiment investigates the performance improvement by using contextualized indexing with SBM for C-BIRCH. We use the same query as in 4.2 but keep *R=40* sec while scaling *AC* by varying the grid size in SDC to produce different streams. For each data stream, we measured the average sliding time with C-BIRCH for 400 window instances with a contextualized index compared with a regular hash index on *cxtid* in CCT-BIRCH. As expected from the analysis in Section 3.3 and shown in Figure 8a, the contextualized index on *cm* substantially improves the scalability. The standard deviation was between 2% to 8% for both SBM-NCI and SBM-CI. Notice that even SBM_NCI does not keep up when *AC>600*, i.e. the response time exceeds one second, while SBM_CI scales much better with increasing *AC*.



a) Impact of indexing b) Impact of summarization factor

Figure 8. Varying AC in complete window instances

4.4 Summarization factor impact

To investigate the impact of the number of generated clusters, we define the *summarization factor* as:

$$1 - \frac{number\ of\ clusters\ in\ a\ complete\ window\ instance}{number\ of\ points\ in\ the\ complete\ window\ instace}$$

Figure 8b compares our sliding mechanism for different summarization factors. SDC is used with *PR=180*, while changing the grid size to vary the number of clusters from 56 (summarization factor 0.97) to 800 (summarization factor 0.54). Each complete window instance in this SDC had 1800 data points.

SBM is always better than RM, since having a high number of clusters makes both methods have more clusters.

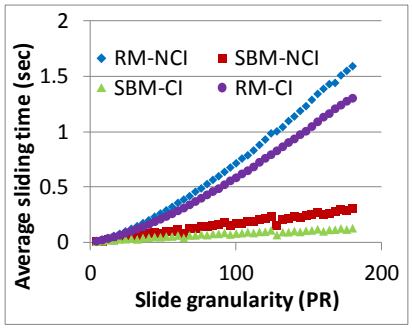
4.5 Contextualized C-BIRCH for Real Data

Figure 9a illustrates the effect of increasing range *R* in the following query over the real DEBS data stream:

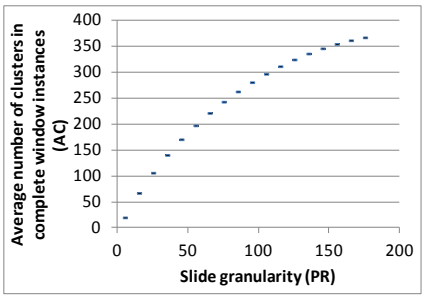
```
SELECT   Center(cid), Radius(cid), COUNT(cid)
FROM     DEBS (Range = R, Slide = 1sec)
CLUSTER BY X, Y, Z AS cid
USING    C-BIRCH(Radius = 2.0 meters)
```

Unlike the synthetic streams above, real data streams are more dynamic, causing *AC* to change with changing window sizes. As expected SBM scales better than RM. The SBM scalability is further substantially (factor 2) improved by contextualized indexing, while indexing does not improve RM very much (factor 1.1). This is expected because the excessive copying overhead of RM offsets the gains by the indexing, as mentioned in Sec. 3.2.1. RM with indexing starts lagging when *PR=150*. The total speed-up for this real data stream from RM_NCI to SBM_CI is 12.3. The standard deviation was 2%-10% for RM-CI, 2%-11% for SBM-CI, 2%-11% for SBM-NCI, and 2%-15% for RM-NCI.

We notice that in Figure 9a RM converges toward linear slow down for larger windows even though the complexity estimates in 3.2.1 indicate that it should slow down quadratically. To investigate this, Figure 9b shows how *AC* changes when *PR* is increased for the DEBS data stream. In general, *AC* gets saturated when *PR* increases, as is often the case for real data. In the DEBS data stream, as the window size increases more parts of the soccer field are covered by the players. This leads to an abundance of existing micro-clusters that absorb newly arriving data points, effectively slowing down the increase of *AC*. The standard deviation for Figure 9b was between 2% and 8%.



a) Sliding performance



b) AC variation

Figure 9 - C-BIRCH over DEBS2013 data stream

4.6 Memory Consumption

Figure 10 compares the memory consumptions of SBM with RM in terms of number of clusters when *PR* is scaled for the DEBS data stream and the same query as in Section 4.5. The number of clusters in both methods is proportional to *AC·PR*, validating the formulations in Sec. 3.2. Since *AC* increases as *PR* is scaled (Figure 9b), *AC·PR* is a quadratic curve. SBM has a higher total number of clusters because the number of BIRCH clusters saturates as *PR* scales. This means that the intermediate nodes in the SBM are not significantly smaller w.r.t. the number of clusters in them than the root node, unlike the proportional relationship assumed between *PR* and *AC* in the formulations in Section 3.1. Therefore for this saturating data stream, the constant multiplier of *AC·PR* is larger for SBM compared to RM. Furthermore, there are no dips in the SBM curve, i.e. the SBMs having *PR*s that are powers of two do not have significantly lower total number of clusters. This is because the auxiliary nodes in the extended SBM add an insignificant number of clusters to the total sum.

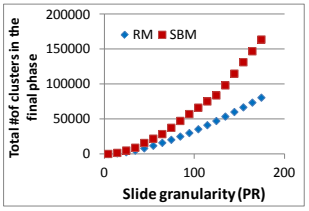


Figure 10. Memory consumption

4.7 Workload Breakdown

Figure 11 compares the percentage time spent in each plug-in function for different window maintenances methods for the query in Section 4.5, including the overhead of G2CS. For RM-NCI and RM-CI the copier plug-in dominates. This is because in RM all the maintained window instances are complete window instances, containing *AC* clusters. The copier plug-in takes even more time with RM-CI since copying indexed data involves costly index re-builds for many complete window instances. On the other hand, for SBM the copier takes much less time since many of the nodes in the SBM cover a shorter range *R* and thus contain much fewer clusters. This means that even though the copier cost increases for SBM-CI, unlike RM, the extra overhead is not significant enough to undermine the benefits of contextualized indexing. The adder and reporter take relatively more time with SBM, since the costly merger in the final phase is no longer the bottleneck. The added overhead of G2CS is between 5% and 17%.

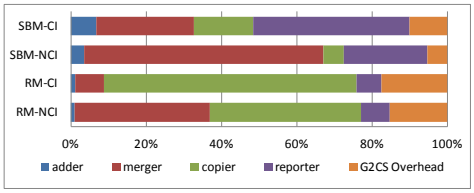


Figure 11. Workload break down for C-BIRCH variants

4.8 Clustering Quality

In this experiment we compare the quality of clusters detected by C-BIRCH with the clusters detected by regular BIRCH. We used a Java implementation of BIRCH [17]. The clustering quality measure used in the experiment is the *Weighted Average Radius* (WAR) [14] of clusters. We used a 200 sec portion of the DEBS data stream when high soccer activity is observed and used the radius threshold parameter of three meters. The BIRCH algorithm was applied in batches of overlapping window instances, while C-BIRCH used the SBM window maintenance.

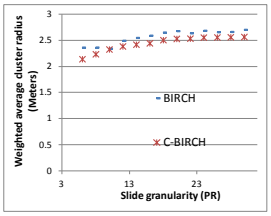


Figure 12. Comparing the accuracy of C-BICH and BIRCH

PR was scaled by keeping the stride constant at one second while the range was scaled to 30 seconds. For each *PR*, there were a number of window instances over which the WAR was calculated for the two algorithms. For the BIRCH algorithm, WAR was calculated over leaf-node-level micro-clusters in all window instances of a given *PR*.

Figure 12 compares the weighted average cluster radius of the two algorithms as *PR* is scaled. C-BIRCH produces clusters with 1 to 5 % lower weighted average cluster radius (a lower WAR is considered better [14]). This experiment verifies that the two-phase C-BIRCH provide clusters that are very similar to BIRCH clusters.

5. RELATED WORK

Differential maintenance approaches for continuous aggregate functions over sliding windows [8] [2] [4] [5] [6] are not applicable to clustering algorithms, which usually are not decremental. The scalable methods for handling non-decremental

aggregate functions by [18] [19] do not support clustering since they do not allow for evolution of group memberships, which is essential for continuous re-clustering of data streams.

Kanat et al. [19] proposed a non-decremental aggregate functions maintenance approach that uses a flattened Fixed Size Aggregator (Flat-FAT) binary tree. Flat FAT is not applicable to clustering algorithms since it requires a pre-split phase based on the group key, which disallows dynamic group membership changes. Furthermore, Flat-FAT does not support generation based window maintenance, disallowing nested windows. Very recently, Kanat et al. proposed an amortized constant time sliding solution for non-decremental aggregate functions over sliding windows based on two stacks [20]. While this solution further improves the conventional aggregation over sliding windows, it is still dependent of pre-splitting and is not generation based.

Slider [21] supports single pass clustering algorithms like BIRCH over sliding windows, but since it utilizing map-reduce tasks over Hadoop clusters it does not meet the real-time requirements of streaming applications. In contrast, the sliding times in our main memory oriented approach is typically below one second.

Most data stream clustering methods are one-pass algorithms [14] [10] [22] [23] where stream elements are read one by one, which is different from conventional data clustering algorithms like K-means and DBSCAN, where the whole database is available and can be searched in several passes. The one-pass algorithms reduce the memory footprint by maintaining an approximate summary of the clustering information in main memory while the data is being scanned. The single pass algorithm in [11] is designed for data warehousing where a *delta* is periodically added to the current database. This is similar to the merger plug-in. However, it does not efficiently handle concept drift since its method for deleting expired elements often implies complete re-clustering every time deltas expire, as clusters might shrink, split, or disappear [12].

The single pass algorithms look very promising for data stream clustering, first because they are non-blocking, i.e. they do not need the complete dataset to provide the data clustering, and second because they provide a summary of the clusters rather than including all the individual points in each cluster. However they fail to address the concept drift as deletion is either not defined or is inefficient, as discussed in this paper.

Single pass algorithms are sensitive to the order of read data. One way to overcome this is to have a second phase [14] where summaries built during a first scanning phase are re-processed. In G2CS this re-processing is performed by the merger plug-in in the final phase.

Babcock et al. [13] introduced repetitive merge to enable a single-pass algorithm to be used for continuous clustering over sliding windows. We have shown that repetitive merge is too inefficient for fast concept drift in real-time data mining, while SBM scales for fast concept drift.

STREAM [10] is another example of a stream clustering algorithm for sliding windows using repetitive merge. It uses the *small space algorithm* where K-means is first applied on disjoint partial window instances, each producing K intermediate centroids. Then repetitive merge is used for applying K-means on the intermediate centroids to obtain the final K centroids.

Extra-N [12] and SGS [24] modify DBSCAN for sliding windows by integrating the sliding mechanism into the algorithm. While this approach minimizes the number of spatial index lookups, it also uses the repetitive merge approach and therefore does not support fast concept drift. Furthermore, the algorithm is very complex since clustering and sliding mechanisms are tied together, while G2CS completely separates indexing and window maintenance from the plugged-in clustering algorithm.

The stream clustering frameworks proposed by Charu C. Aggarwal, et. al. in [25], [26], and [27] do not support sliding windows.

To the best of our knowledge G2CS is the only stream clustering framework that supports sliding windows while avoiding repetitive merge and separating cluster indexing and window maintenance from the plugged-in algorithm.

6. CONCLUSIONS AND FUTURE WORK

G2CS provides a real-time sliding window maintenance framework for non-decremental clustering algorithms using SBM. It supports cluster evolution by organizing clustering and summarization using contexts. Scalable multi-dimensional search for the closest cluster over sliding windows is provided by contextualized indexes. Furthermore, G2CS separates the sliding and indexing mechanisms from the applied clustering algorithm, which structures and simplifies the implementation of clustering algorithms over sliding windows. We also developed Continuous BIRCH C-BIRCH, an equally accurate variant of BIRCH that is applicable on sliding windows. By extensive experimentation over real and synthetic data streams we showed that the proposed methods significantly improve the real-time performance of the applied clustering algorithms while the accuracy of the algorithm is not sacrificed.

G2CS provides two main future research directions. First, there are opportunities for multi-query optimization by analyzing the SBM-lattice when the window specifications of the submitted queries are different since each level of the SMB-lattice maintains different window ranges. Second, there are parallelization and distribution opportunities as G2CS is easily data parallelizable at different stages. For example, if partial summarization becomes a bottleneck, the system can create other instances of it to parallelize the work and distribute the partial windows using, e.g., round-robin [28]. Parallelizing and distributing SBM is also an interesting topic to investigate given the dependencies between the contexts in the SBM-lattice.

REFERENCES

[1]Lukasz Golab and Tamer M Özsu, "Issues in data stream management," in *SIGMOD Record*, 2003, pp. 5-14.

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

[3]Carlo Zaniolo and Haixun Wang, "Logic-based user-defined aggregates for the next generation of database systems," in *The Logic Programming Paradigm.*: Springer Berlin Heidelberg, 1999.

[4]Z. Rui, N. Koudas, B. C. Ooi, and D. Srivastava, "Multiple aggregations over data streams," in *SIGMOD conf.*, Baltimore, Maryland, 2005.

[5]Krishnamurthy S., C. Wu, and M. Franklin, "On-the-fly sharing for streamed aggregation," in *SIGMOD conf.*, Chicago, Illinois, 2006.

[6]G. Shenoda, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis, "Optimized processing of multiple aggregate continuous queries," in *Proceedings of the 20th ACM international conference on Information and knowledge management*, Glasgow, 2011.

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

[8]C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, "Gigascope: a stream database for network applications," in *SIGMOD conf.*, New York, 2003, pp. 647-651.

[9]T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: an efficient data clustering method for very large databases," in *SIGMOD conf.*, Montreal, 1996., pp. 103-114.

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

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

[12]Di Yang, E. A. Rundensteiner, and M. O. Ward, "Neighbor-based pattern detection for windows over streaming data.," in *EDBT conf.*, Saint Petersburg, 2009, pp. 229-540.

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

[14]Tian Zhang, Raghu Ramakrishnan, and Miron Livny, "BIRCH: an efficient data clustering method for very large databases," in *Proceedings of the 1996 ACM SIGMOD international conference on Management of data* , 1996, pp. 103-114.

[15]Stefan Berchtold, Keim A Daniel, and Hans-Peter Kriegel, "The X-tree : An Index Structure for High-Dimensional Data," in *Proc. VLDB Conf.*, 1996, pp. 28-39.

[16]Thanh Truong and Tore Risch, "Transparent inclusion, utilization, and validation of main memory domain indexes," in *27th International Conference on Scientific and Statistical Database Management*, San Diego, 2015.

[17]Roberto Perdisci. (2015, November) JBIRCH - BIRCH clustering implementation in Java. [Online]. http://roberto.perdisci.com/projects/jbirch

[18]Jennifer Widom and Jun Yang, "Incremental Computation and Maintenance of Temporal Aggregates," in *Proceedings of the 17th International Conference on Data Engineering*, 2001, pp. 51-60.

[19]Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu, "General incremental sliding-window aggregation," *Proceedings of the VLDB Endowment*, vol. 8, pp. 702--713, 2015.

[20]Kanat Tangwongsan, Martin Hirzel, and Scott Schneider, "Constant-Time Sliding Window Aggregation," IBM, IBM Research Report RC25574 (WAT1511-030), 2015.

[21]Pramod Bhatotia, Junqueira P Flavio, Acar A Umut, and Rodrigo Rodrigues, "Slider: Incremental Sliding Window Analytics," in *Middleware'14*, Bordeaux, France., 2014, pp. 61-72.

[22]Fazli Can, "Incremental clustering for dynamic information processing," *ACM Transactions on Information Systems (TOIS)* , vol. 11, no. 2, pp. 143-164 , 1993.

[23]Douglas H Fisher, "Knowledge acquisition via incremental conceptual clustering," *Machine learning*, vol. 2, pp. 139--172, 1987.

[24]Di Yang, Elke A Rundensteiner, and Matthew O Ward, "Summarization and matching of density-based clusters in streaming environments," in *Proceedings of the VLDB Endowment*, 2011, pp. 121-132.

[25]Charu C Aggarwal, Jiawei Han, Jianyong Wang, and Philip S Yu, "A framework for clustering evolving data streams," in *VLDB '03 Proceedings of the 29th international conference on Very large data bases*, 2003, pp. 81-92.

[26]Charu C Aggarwal, Jiawei Han, Jianyong Wang, and Philip S Yu, "A framework for projected clustering of high dimensional data streams," in *VLDB '04 Proceedings of the Thirtieth international conference on Very large data bases*, 2004, pp. 852-863.

[27]Charu C Aggarwal and Philip S Yu, "A framework for clustering uncertain data streams," in *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, 2008, pp. 150--159.

[28]E. Zeitler and T. Risch, "Massive scale-out of expensive continuous queries," in *VLDB conf.*, Seattle, 2011, pp. 1181-1188.