UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1431

Real-time data stream clustering over sliding windows

SOBHAN BADIOZAMANY

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2016

ISSN 1651-6214
ISBN 978-91-554-9698-2
urn:nbn:se:uu:diva-302799

Dissertation presented at Uppsala University to be publicly examined in ITC 2446, Lägerhyddsvägen 2, Uppsala, Wednesday, 23 November 2016 at 10:00 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor Tamer Özsu.

Abstract

Badiozamany, S. 2016. Real-time data stream clustering over sliding windows. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 1431. 33 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-554-9698-2.

In many applications, e.g. urban traffic monitoring, stock trading, and industrial sensor data monitoring, clustering algorithms are applied on data streams in real-time to find current patterns. Here, sliding windows are commonly used as they capture concept drift.

Real-time clustering over sliding windows is early detection of continuously evolving clusters as soon as they occur in the stream, which requires efficient maintenance of cluster memberships that change as windows slide.

Data stream management systems (DSMSs) provide high-level query languages for searching and analyzing streaming data. In this thesis we extend a DSMS with a real-time data stream clustering framework called *Generic 2-phase Continuous Summarization framework (G2CS)*. G2CS modularizes data stream clustering by taking as input clustering algorithms which are expressed in terms of a number of functions and indexing structures. G2CS supports real-time clustering by efficient window sliding mechanism and algorithm transparent indexing. A particular challenge for real-time detection of a high number of rapidly evolving clusters is efficiency of window slides for clustering algorithms where deletion of expired data is not supported, e.g. BIRCH. To that end, G2CS includes a novel window maintenance mechanism called Sliding Binary Merge (SBM). To further improve real-time sliding performance, G2CS uses generation-based multi-dimensional indexing where indexing structures suitable for the clustering algorithms can be plugged-in.

Keywords: Data streaming; Sliding windows; Clustering;

Sobhan Badiozamany, Department of Information Technology, Division of Computing Science, Box 337, Uppsala University, SE-75105 Uppsala, Sweden. Department of Information Technology, Computing Science, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-554-9698-2
urn:nbn:se:uu:diva-302799 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-302799)

To my family

List of Papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I Badiozaman, S., Risch, T. (2012) Scalable ordered indexing of streaming data. *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*

II Badiozamany, S., Melander, L., Truong, T., Xu, C, Risch, T. (2013) Grand Challenge: Implementation by Frequently Emitting Parallel Windows and User-Defined Aggregate Functions. *The 7th ACM International Conference on Distributed Event-Based Systems*

III Badiozamany, S. (2014) Distributed multi-query optimization of continuous clustering queries, *VLDB PhD Workshop*

IV Badiozamany, S., Orsborn, K., Risch, T. (2014) Framework for real-time clustering over sliding windows. *SSDBM 2016 Conference on Scientific and Statistical Database Management*

Reprints were made with permission from the respective publishers. All papers are reformatted to the one column format of this book.

Table of contents

Abbreviations

DSMS	Data Stream Management System
G2CS	Generic 2-layer Continuous Summarization
SBM	Sliding Binary Merge
PGS	Partial Grouped Summary
RM	Repetitive Merge

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Tore Risch for his continuous support of my research, for his patience and encouragement. Thank you Tore for the sleepless nights we were working together before deadlines. I would like to thank my co-supervisor Kjell Orsborn for discussing research ideas and providing valuable feedback.

I thank former and current colleagues Ahmad Alzghoul, Andrej Andrejev, Matteo Magnani, Khalid Mahmood, Lars Melander, Silvia Stefanova, Thanh Troung, Cheng Xu, Erik Zeitler, and Minpeng Zhu for sharing their experience and helping me in teaching and research activities.

More specifically, I would like to thank Silvia, Minpeng, Matteo, and Thanh for all the fun, the barbeques and parties. Thanks for all the chats we had about ordinary life as PhD students, your friendship made me feel at home during the PhD studies. Thanh, I am also grateful for sharing your extensive knowledge of indexing.

Last but not the least; I would like to thank my family: Elham and Avina for supporting me spiritually throughout my PhD and my life in general. I would also like to thank my parents for their dedication and for many years of support during my whole life that has provided the foundation for this work.

This project was supported by EU FP7 Project Smart Vortex, the Swedish Foundation for Strategic Research, and eSSENCE under contract RIT080041.

1 Introduction

In the big data era, the data is produced at extremely high velocities and volumes. In many cases the data is in the form of data streams, making the approach of storing and querying the data offline infeasible. Examples of data streaming applications are urban traffic monitoring, stock trading, and industrial sensor data monitoring. To address these applications, Data Stream Management Systems (DSMSs) are used where queries continuously process data in real-time and emit output, as opposed to querying stored data.

Many data streams represent the current state of dynamic systems, e.g. geo-located streams of vehicle positions in an urban area, where knowing the current characteristics of the systems with low latency is highly desired. To enable real-time stream processing the DSMS should not fall behind the current stream and have a low response time, which requires efficient data indexing and stream maintenance mechanisms. Most DSMSs use main memory for query processing to meet low latency requirements.

To dynamically capture evolution of the underlying system data streams, usually a window of most recent data is continuously queried where the window slides forward as a data stream progresses using continuous GROUPBY queries.

When the exact grouping of data is unknown conventional GROUPBY is insufficient because it is based on equality of group keys. In this case clustering algorithms are used, e.g. KMEANS [1] and DBSCAN [2], to form the groups and maintain the statistics. Clustering streaming data is particularly challenging because it involves dynamically merging and splitting evolving clusters over which statistical summaries are maintained in real-time as the stream progresses [3] [4] [5].

Most of the previous work on data stream clustering is focused on developing monolithic algorithms where the sliding window and indexing mechanisms are included in the algorithm, resulting in intertwined implementations that are not easily reusable. For example EXTRA-N [4] and SGS [5] have a spatial index on window contents: both windowing and indexing mechanisms are implemented within the clustering algorithm. BIRCH [6] is another example of a clustering algorithm that uses an application specific indexing technique, called CF-tree. Implementing data stream mining algorithms from scratch requires a very rare combination of skills so there is a need for high-level frameworks where data scientist can express data analy-

ses on a high level, while complex algorithms, indexing and other low-level storage options can be reused and plugged-in beforehand [7].

In the context of this Thesis, clustering algorithms and conventional GROUPBY aggregation are different forms of *data stream summarization algorithms* as they both group and summarize the data streams.

This Thesis addresses the generic problem of data stream summarization over sliding windows in real-time by the *Generic 2-layer Continuous Summarization (G2CS)* framework where different summarization algorithms and indexing structures can be plugged-in.

The following research questions are investigated:

1. What is the most suitable window sliding mechanism for different kinds of stream summarization algorithms?
2. What is the suitable indexing mechanism for data summarization over sliding windows?
3. How can the window sliding mechanism be separated from both indexing and the applied summarization algorithm to avoid intertwined implementations?

To address the research questions we developed G2CS where we evaluated different implementation alternatives.

To address research question 1, in Paper II we analyzed a use case to investigate different query processing methods for GROUPBY queries. A two-phase approach [8] [9] [10] [11] [12] for stream summarization over sliding windows was then picked for further development. Based on this case study, in Paper IV a two phase approach is presented to efficiently support clustering queries using a novel window maintenance mechanism called Sliding Binary Merge (SBM).

Paper I addresses research question 2 by presenting a generic approach for indexing the data in sliding windows that continuously maintain GROUPBY aggregates by slicing both the data and the index in sliding windows. This approach is generalized in Paper IV to index the data required for maintaining dynamically changing clusters over sliding windows.

To address research question 3, in paper IV it is shown how G2CS separates sliding window maintenance and indexing from plugged-in summarization algorithms, which makes it significantly easier for data scientists to perform clustering over data streams. G2CS allows for re-use of software components and simplify the introduction of new algorithms.

This Thesis overview is organized as follows. Chapter 2 presents the technology background and reviews the related work. Chapter 3 presents G2CS, the framework that implements all the contributions. Chapter 4 states the contributions made in each of the four papers, Chapter 5 presents possible future directions of this Thesis work, and the Thesis summary in Swedish is presented in Chapter 6.

2 Background and related work

In this chapter first general technologies and definitions for real-time data stream analysis are presented. Then approaches for real-time data stream mining related to G2CS are discussed in details.

2.1 Data Stream Management Systems

A data stream is a continuously extended sequence of tuples, which is usually ordered, commonly by tuple arrival time or tuple number. In most scenarios, due to high data volume and velocity, the elements can be read only once. Data streams are produced for example by sensor readings from machines [13], live update of stock prices [14], and spatio-temporal readings from a number of moving objects [15] [16]. The massive amount of data produced by data streams need to be systematically processed and analyzed.

There are a number of systems, e.g. Storm [17], Amazon Kinesis [18], MillWheel [19], Flink streaming [20], and Microsoft Streaminsight [21] for high-performance data stream processing using a regular programming language where a processing pipeline is explicitly programmed. In general, it is desirable to have a high level query language to analyze data streams, as in Streambase [22], SQLStream [23], and Gigascope [24]. This is because, similar to processing data in conventional applications, a high level query processor enables data analysis for non-programmers since the users express "what" information is to be retrieved instead of "how" to retrieve it.

Data Stream Management Systems (DSMSs) [25] are software systems that manage and support querying of continuous data streams. Example DSMSs are STREAM [26], TelegraphCQ [27], Gigascope [24], and SCSQ [28]. Since data streams are unbound and echo the changing behavior of a monitored system, the query model in a DSMS reflects this dynamic behavior using *continuous queries* [29] where, unlike traditional database queries, the result continuously changes as the stream progresses. Continuous queries run and emit updated output until they are explicitly terminated by the user.

Data streams often have high volume and velocity, so DSMSs need to meet the following requirements:

- The arriving stream elements have to be processed on-the-fly.

- Processing data streams usually is done in a single pass, in contrast to regular query processing methods that rely on visiting tuples multiple times.
- To be able to keep up with the stream flow, the processing engine needs to have low latency and high throughput. Therefore, the stream processing engines often operate in main-memory and use responsive main-memory indexing structures.

Event driven systems [30] are similar but optimized for analyzing complex event patterns using a reactive language.

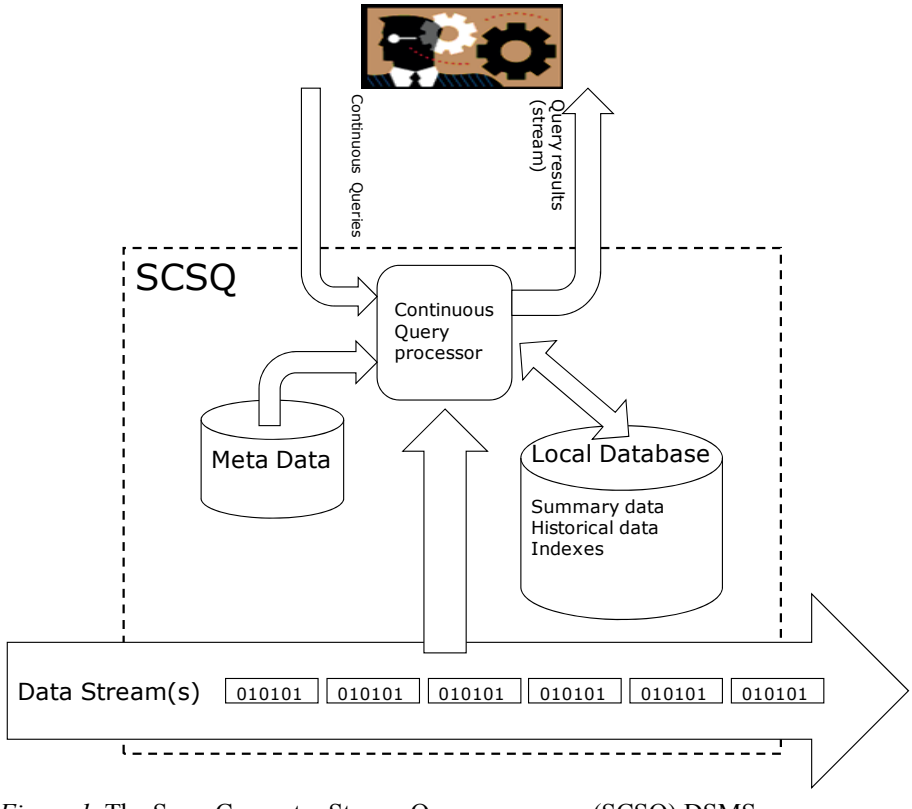


Figure 1. The SuperComputer Stream Query processor (SCSQ) DSMS

Figure 1 illustrates the architecture of SCSQ. The *meta-database* stores the schema used to express queries. The *local database* maintains in main-memory data representing current data summaries as well as historical data using main-memory indexes for efficient search. The *continuous query processor* optimizes and executes continuous queries over the data streams by accessing the meta-data and the local database. Continuous queries can span both streaming and local data.

This Thesis work extends the DSMS SCSQ to process real-time data mining queries over sliding windows.

2.2 Real-time data stream clustering

In many applications, complex data mining algorithms are applied on the stream in real-time to find current data patterns. For example, automated financial systems use real-time modeling and monitoring of the stock price trends for predictive applications. Another example is monitoring urban traffic in real-time where the data streams produced by connected vehicle GPS systems are analyzed to detect traffic regions by forming clusters of cars.

Traditional data clustering algorithms such as K-means [1], Self Organizing Maps [31], density based clustering techniques such as DBScan [2] and CLIQUE [32], are applied on finite static data. This allows for several passes through the stored data. In contrast, because data streams are infinite, data stream mining algorithms need to process the data in single pass as in Denss-tream [3], BIRCH [6], Extra-N [4], SGS [5]. In addition to the single-pass requirement in data stream clustering, real-time data stream clustering requires detecting the continuously evolving clusters formed as the stream progresses. Responsive real-time cluster detection requires early detection of clusters as soon as they occur in the stream, which is addressed by G2CS. For example, a moving car can use traffic data streams to form clusters of cars in the road to actively detect traffic congestion ahead and slow down. In this case responsive detection of clusters is essential for safety reasons.

While data stream clustering can be done using data stream processing engines (e.g. Storm, Kinesis, Stream Mill, Flink Streaming, stream insight, IBM system S), DSMSs are better fit for data stream clustering because the end users can express the clustering task in terms of high level queries. For example, the following query [Paper IV] detects congested areas with radius 50 meters over a window of vehicle positions X and Y using a modified version of the clustering algorithm BIRCH [6] for sliding windows, C-BIRCH.

```
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)
```

Here, C-BIRCH is an algorithm that is plugged into the DSMS and the FROM clause specifies a sliding window, which is discussed in the next section. G2CS allows for plugging-in clustering algorithms such as C-

BIRCH and executing them with low response times over sliding windows. The plug-ins can be defined in terms of high level queries to the local main-memory database. This simplifies the implementation of the algorithms and improves their performance by utilizing query optimization techniques, e.g. automatic index utilization.

2.3 Sliding windows

To enable processing of blocking operators like average and sum over infinite data streams, *windowing* is commonly used in data stream processing because it limits the extent of data to a sequence of most recent elements in the data stream. Clustering algorithms are blocking since they require having all the data points to compute the clusters. Therefore, windowing is needed for data stream clustering.

There are different ways of defining a window over a data stream [33]. The *window specification* defines how recent stream elements are selected for windowing in the FROM clause of a continuous query. When the window specification is applied on a live data stream it produces new *window instances* at different points in time. A window instance logically contains a set of stream elements.

For example, a *sliding window* is specified by defining its *range* and *stride*. The range R of a sliding window specifies the length of the window while the stride S specifies the portion of the range that is evicted from the window when the window moves forward. A sliding window is specified as a tuple $<R,S>$, where $S<R$. Two common kinds of sliding windows are *time-based* and *count-based* sliding windows. In time-based sliding windows R and S are defined using time intervals while in count-based sliding windows they are defined in terms of the number of elements. For example a time based sliding window with $R=10min$ and $S=2min$ produces window instances that cover the data in the last 10 minutes of the stream and a new window instance is created every 2 minutes. Without loss of generality, we present sliding windows using time-based sliding windows.

Real-time data stream clustering can be done using sliding windows because they reflect the recent elements in the stream. To responsively detect rapidly changing clusters, a smooth sliding specification is highly desired where the stride S is small relative to the range R, i.e. the *partitioning ratio* $PR=R/S$ is high. G2CS provides a sliding mechanism that allows for efficient processing of clustering algorithms with high PR [Paper IV].

2.4 Indexing sliding windows

The contents of window instances can be stored as a sequence of data items in a main-memory buffer. However, when the execution of continuous queries involves searching the contents of the window instance, the search can become expensive if there are many elements in it, e.g., finding vehicles with a particular pattern in their plate number in a window of a live traffic stream, or finding the activities of certain mobile phone users within the past hour. Data indexing [34] is a general technique that provides efficient search in databases which can also be applied on indexing large data stream windows [35].

Real-time data clustering algorithms often require continuous search for similar objects in a multi-dimensional space. Calculating the similarity of all the objects in a large window can be prohibitively expensive; therefore multi-dimensional indexing that supports efficient similarity search is required for responsive real-time query processing. Each clustering algorithm might have a different indexing structure so it is desirable to be able to plug-in different indexing data structures.

Indexing the contents of sliding window instances brings about the following challenges:

- The indexing structure need to support very high insertion rates to be able to add the arriving data to the window.
- They also need to have a high performing deletion mechanism to evict expired data.

In data stream processing main memory indexing structures need to be used to keep up with the stream flow.

G2CS provides a method for plugging-in algorithm-specific main-memory indexing of the elements in window instances, while supporting high insertion rates and a bulk deletion method [Paper I]. For responsive clustering over sliding windows, G2CS uses a general indexing framework described in [Paper IV] that separates the indexing from both the applied clustering algorithms and the sliding mechanisms. This is shown to significantly improve the response time.

2.5 Two-phase aggregation over sliding windows

Figure 2 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 Figure 2 the data in window instance $W_{0,10}$, covering the time interval *[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 $W_{2,12}$.

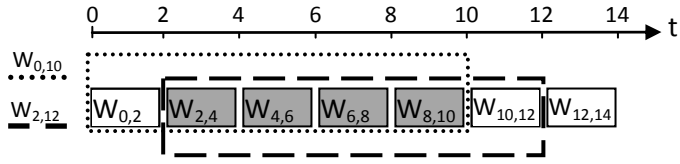


Figure 2. Data overlap in sliding windows

To avoid unnecessary recomputations when data is summarized over sliding windows, efficient *differential maintenance* techniques can be used [36]. For example in Figure 2, the summary for $W_{2,12}$ can be obtained by adding $W_{10,12}$ to the summary for $W_{0,10}$ and removing $W_{0,2}$ from the summary for $W_{0,10}$.

Differential processing is usually done by introducing functions for adding/removing deltas to/from the aggregation state [36]. For example, the aggregate function COUNT is differential because both of the following equations hold:

$$COUNT(A\ B) = COUNT(A) + COUNT(B)\ \textit{(Incremental)}$$
$$COUNT(A \backslash C) = COUNT(A) - COUNT(C)\ \textit{(Decremental)}$$

Here A, B, and C are sets. Using the incremental and decremental properties, the summary for a sliding window is differentially maintained as follows. When a slide happens the summary of the most recent window instance is reused by adding the incoming elements using the incremental function and removing the expired elements using the decremental function.

For many data stream summarization algorithms, e.g. clustering algorithms, there is no decremental function, e.g. in BIRCH. Even when a decremental method can be devised as in [37], it can be very expensive and must be avoided, as suggested by previous work [4]. G2CS provides a novel window maintenance technique for efficiently maintaining summaries without requiring the decremental function [Paper IV].

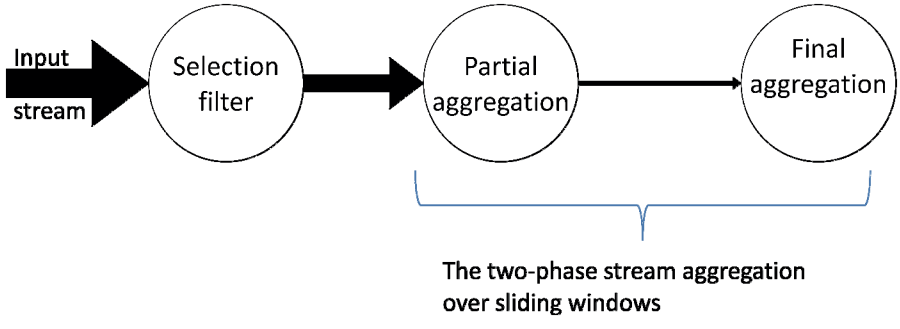


Figure 3. Sliding window aggregation

Figure 3 shows the widely used *two-phase aggregation* technique [8] [9] [10] [11] [12] for computing aggregation over sliding windows. The thick-

ness of the arrows in the figure indicates the volume of the stream. First, the *selection filter* removes the stream elements that are outside the query selection criteria. The second and third processes constitute the two-phase aggregation. After applying the selection filter, the two-phase aggregation summarizes the stream per group as follows:

1. In the first phase, called *partial aggregation*, fine-grain non-overlapping partial window instances are formed where aggregated 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 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 [24]. 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*.

G2CS extends the two-phase approach to also support non-decremental summarization algorithms such as clustering algorithms by introducing a sliding mechanism for responsive maintenance of evolving clusters in the second phase for clustering algorithms that do not have the decremental property. The first phase is similar for both clustering and aggregation, only the aggregated data is algorithm dependent.

2.6 Two-phase clustering over sliding windows

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.

Figure 4a shows an example of dynamic group membership in clusters *a1*, *a2*, and *a3*. Figure 4b illustrates two arriving points (green) and two expired ones (red). The resulting point-to-cluster memberships in Figure 4c are completely new.

G2CS allows for such dynamic change of group memberships for clustering algorithms by allowing for combined group formation and data summarization [Paper IV].

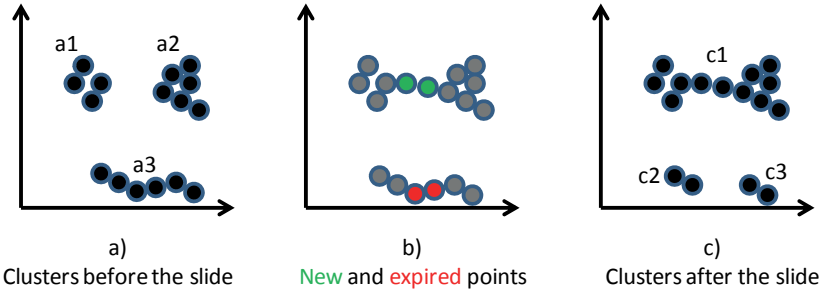


Figure 4. Evolving group memberships in clustering algorithms over sliding windows

The dynamically changing group memberships in clustering algorithms has the following implications on how they are processed over sliding windows, compared to conventional GROUP-BY queries:

a. Streamed clustering algorithms require 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 [8] [38]. Therefore stream clustering algorithms needs to maintain their own data structures to represent clusters that are updated as the window slides.
b. For many clustering algorithms, incremental deletion of data points from clusters is not defined [39], i.e. they are not decremental. Even when a decremental method can be devised as in [37], it can be very expensive and must be avoided, as suggested by previous work [4].
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.

G2CS addresses a. by allowing clustering algorithms to store multiple generations of summarization data as the cluster memberships evolve over time with window slides. G2CS addresses b. by a novel window maintenance technique called Sliding Binary Merge (SBM) which is very efficient when the applied summarization function is non-decremental allowing for respon-

sive slides. To address c. G2CS provides transparent index plug-ins to speed up the multi-dimensional search in clustering algorithms, which improves the response time.

In related work [40] [4] [5] [41], to support non-decremental clustering algorithms, 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) approach is illustrated in Figure 5 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 $PGS_6 – PGS_9$.

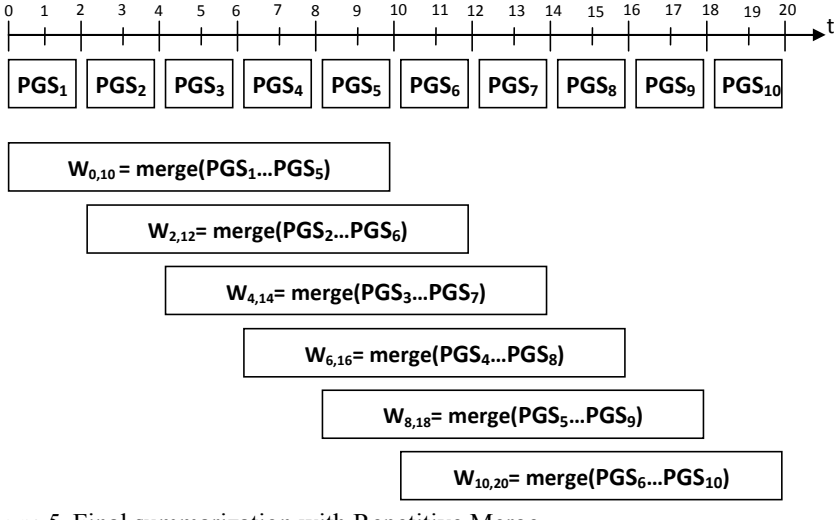


Figure 5. Final summarization with Repetitive Merge

With repetitive merge the number of merges per slide becomes high when *PR* is high, which substantially decreases responsive detection of clusters, as shown in [Paper IV]. Unlike the repetitive merge approach, G2CS avoids the overlapping merges by maintaining small intermediate data stream summaries and organizing them using SBM.

3 Generic 2-Phase Continuous Summarization

The different user roles in G2CS are shown in Figure 6. The *data scientists* are the end users submitting *continuous queries* to G2CS to find clusters and perform other analyses. The *algorithm designers* implement new stream summarization algorithms and plug them into the system. Plug-ins can be written to support either conventional GROUPBY aggregation or complex clustering algorithms; the latter is the focus of this Thesis. Clustering algorithms often require specific indexing for responsive cluster maintenance as the clusters evolve. G2CS allows arbitrary indexing structures to be plugged-in by *indexing experts*. By defining plug-ins in terms of queries over the main-memory local database [Paper IV], the algorithm design is separated from index implementation. This is because the query optimizer automatically utilizes indexing [42] when executing the queries over the local main-memory database.

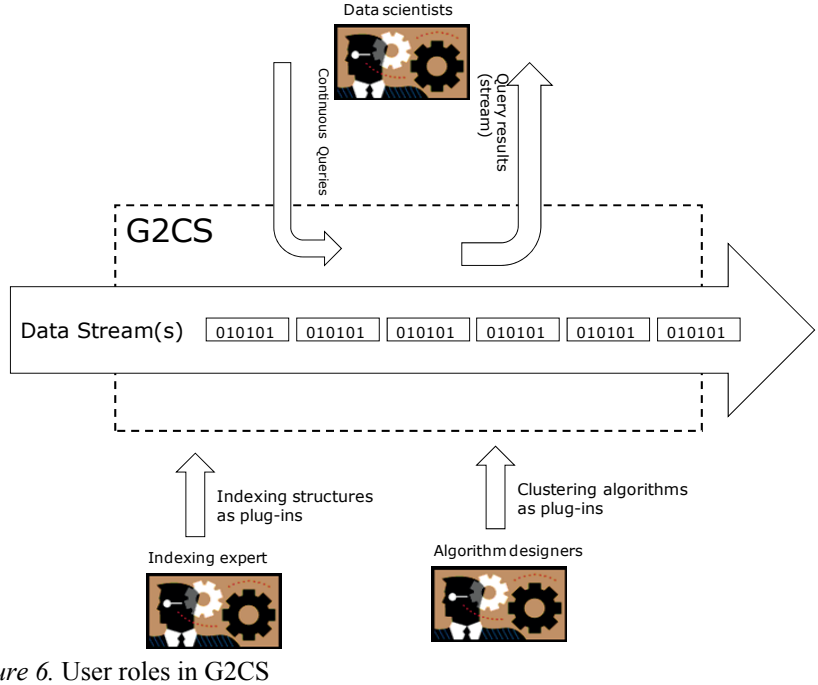


Figure 6. User roles in G2CS

Figure 7 illustrates the architecture of the framework. It utilizes previous work on query processing [43], data stream management [28], and extensible indexing [42]. The contributions of this Thesis are the modules that are blue-shaded in the figure. The *query processor* receives continuous queries and produces execution plans that invoke the G2CS kernel.

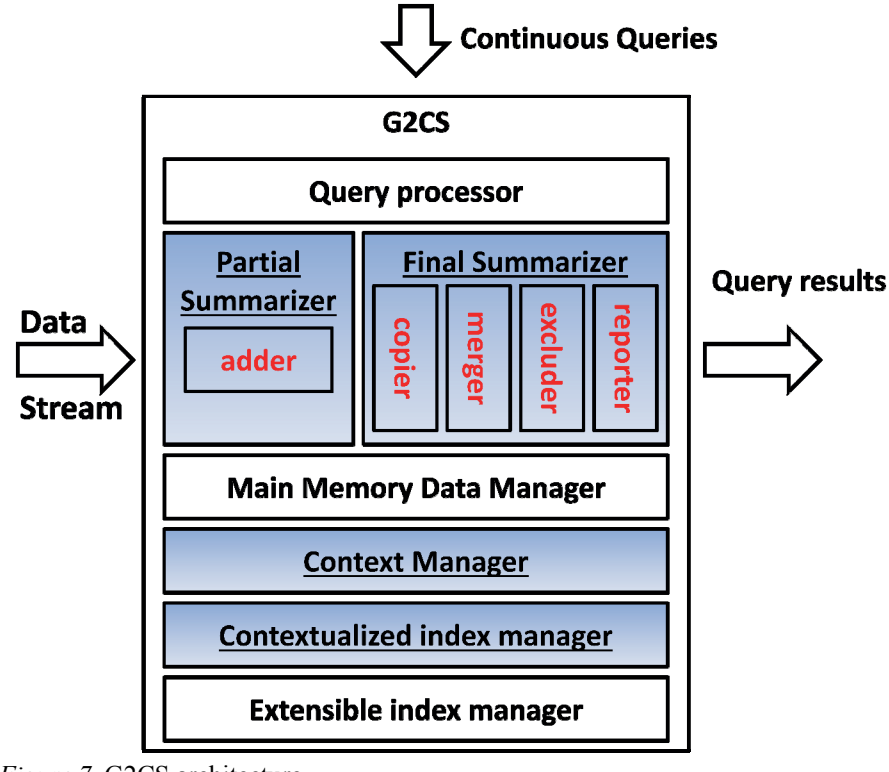


Figure 7. G2CS architecture

The *context manager* organizes window instances by contexts. A context represents the valid time interval of a window instance as a triple <*b, e, cxtid*> where *cxtid* is a unique *context identifier* of the time interval *[b,e)* per window. Contexts 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 algorithms.

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 then assigns a new context for each new partial window instance and 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. The final summarizer implements the second phase of the clustering. For differential algorithms the user can provide methods for both incremental maintenance of clusters (*merger* plug-in) and decremental ones (*excluder* plug-in). For non-decremental algorithms it is crucial to avoid the repetitive merge approach in the final summarizer because of its redundant calculations. In this case G2CS maintains and reuses several layers of intermediate window instances by organizing them by contexts using SBM.

Since SBM maintains a number of intermediate window instances to optimize the sliding mechanism, the *copier* plug-in is invoked to populate new window instances by copying 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 the necessary old and intermediate 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.

4 Contributions

This section summarizes the contributions made by each of the four papers in the Thesis.

4.1 Paper I

This paper presents an efficient main-memory ordered indexing framework for sliding windows over data streams. There are three requirements for indexing sliding windows. First, the search needs to be responsive, second high insertion rates need to be handled to support high stream rates, and third, piecewise bulk deletion need to be supported for responsive deletion of expired data as the window slides. A number of main-memory ordered indexes were investigated where the highly optimized cache-aware compact trie implementation Judy [44] was particularly interesting for responsive indexing as it supports very fast insertion in constant time. However, Judy had a very slow range search. We developed a mapper function that applies a user defined aggregate function while traversing the Judy data structures. The mapper approach does not require source code modification of the very complex Judy implementation while significantly improving its range search. The need for piecewise bulk deletion in sliding windows was addressed by a framework for indexing sliding windows that allows bulk deletion for any plugged-in indexing structure. This framework was further improved in Paper IV by contextualized indexing.

I am the primary author of this paper. The second author contributed in discussions and in writing the paper.

4.2 Paper II

In this paper a solution to the DEBS2013 grand challenge [16] is presented. The problem is implementing four continuous aggregation queries over a stream of spatio-temporal sensor readings produced in a real soccer game. The particular challenge was to minimize the response time of the four continuous queries running in parallel on a single 4-core machine. We used the two-phase stream aggregation over sliding windows to improve the performance of standalone queries. Furthermore, to decrease overall resource con-

sumption, we needed to utilize shared execution for the queries when possible. The two-phase approach allowed for sharing partial aggregations between queries, reducing the overall resource consumption, and providing responsive execution of the four queries.

I am a co-author to this paper. I designed and implemented two of the queries, specifically the two-phase approach for sharing the partial aggregations. I also contributed in writing most of the manuscript.

4.3 Paper III

This paper outlines the overall research project. It includes an investigation of query processing over sliding windows, common multi-query optimization techniques for aggregate queries over sliding windows, and identifies the challenges of designing a framework for responsive data stream clustering. In particular, it identifies supporting non-decremental clustering algorithms as one of the main challenges for real-time data stream clustering over sliding windows, which is addressed in Paper IV.

I am the author of this paper.

4.4 Paper IV

This paper presents the G2CS framework in details. G2CS uses SBM and contextualized indexing for real-time data stream clustering over sliding windows. The design details of SBM and contextualized indexing is outlined in the paper. Furthermore, the paper includes a thorough computational complexity analysis for SBM and contextualized indexing, which is also verified by extensive performance experiments. The paper uses the clustering algorithm BIRCH as a running example to explain how clustering algorithms can be plugged-in, resulting in a variant of BIRCH for sliding windows called Continuous BIRCH, C-BIRCH.

I am the primary author of this paper, developed G2CS and C-BIRCH, and performed extensive performance experiments. The other authors contributed in discussions and writing the manuscript.

5 Conclusion and future work

G2CS is a framework that supports real-time data stream clustering over sliding windows by extending the two-phase approach [8] [9] [10] [11] [12] for stream summarization over sliding windows. It supports responsive detection of clusters over streaming data by a novel sliding mechanism for non-decremental clustering algorithms and multi-dimensional indexing. It uses SBM to avoid overlapping computations which is shown to perform more responsive compared to the previous repetitive merge approaches [40] [4] [5] [41]. To support scalable index insertion and deletion under high volume stream rates, G2CS uses contextualized indexes that separate the implementation of indexing mechanisms from both the applied clustering algorithms and the window maintenance mechanisms. This modularization structures and simplifies the implementation of clustering algorithms over sliding windows and allows for responsive bulk deletion of indexes as the windows slide.

There are a number of future research directions. First, parallelizing and distributing the query processing in G2CS can be investigated to further improve the response time and throughput of the query processing. Second, there are often opportunities for minimizing resource consumption by sharing computations when several continuous queries are submitted to the system having similar query components [9] [10] [11] [12] [40]. For example, queries might share window fragments and selection predicates. Third, more dynamic windowing semantics, like predicate-based and partition-based windowing [45], can be supported by SBM. Fourth, the Thesis assumes that stream elements arrive in order, whereas in some applications some elements might arrive with delays. Therefore, supporting out-of-order element arrival in SBM is desirable.

6 Summary in Swedish

Digitala data produceras numera i extremt höga hastigheter och volymer. Databaser som MySQL och sökmotorer som Google används ofta för att söka och analysera stora datamängder som lagrats på disk i datorer på internet. Emellertid produceras i många fall kontinuerligt strömmande data, t.ex. ljud, aktiedata, trafikdata och mätvärden från sensorer på maskiner. Eftersom strömmande data är obegränsade i storlek och produceras i realtid är det ofta inte möjligt att först lagra dem på disk innan man söker bland dem. I stället vill man söka och analysera data direkt i strömmen snarare än att först mellanlagra dem.

För att hantera strömmande data har speciella sökmotorer utvecklats som brukar kallas *dataströmhanteringssystem* (eng. DSMS, Data Stream Management Systems) till vilka man kan ställa frågor mot dataströmmar på liknande sätt som *databashanteringssystem* (eng. DBMS, Data Base Management Systems) som MySQL hanterar frågor mot lagrade data. En fråga mot strömmande data ger ett kontinuerligt svar i form av en ström, t.ex. genom att kontinuerligt detektera skadliga resonansfrekvenser från en eller flera sensorer som mäter vibrationer i en maskin. En sådan *stående fråga* filtrerar data kontinuerligt så länge den är aktiv medan konventionella databasfrågor utförs omedelbart och avslutas när resultatdata returnerats.

Ofta används stående frågor för att analysera dynamiska system vars uppförande kontinuerligt varierar över tiden, t.ex. strömmar av geo-positioner från fordon i ett stadsområde eller strömmar av uppmätta vibrationer hos en maskin. För att i tid upptäcka onormalt beteende hos det analyserade systemet, t.ex. trafikolyckor eller skadliga vibrationer, bör data från stående frågor produceras med så liten fördröjning som möjligt. Vidare måste dataströmhanteringssystemet vara snabbt nog att bearbeta data minst lika snabbt som de produceras, annars fördröjs systemet alltmer och klarar inte att analysera data med begränsad svarstid. Det vanligaste sättet att begränsa svarstiden i konventionella databaser är indexering, dvs. speciella datastrukturer på disk för att göra sökningar skalbara.

Dataströmhanteringssystem har stora krav på omedelbar bearbetning av mottagna data och därför måste alla data lagras och analyseras i primärminne, inklusive indexdatastrukturerna. Genom att göra all bearbetning i primärminne kan resultatet från stående frågor produceras med liten fördröjning.

Ett *fönster* mot en dataström är en begränsad senaste del av dataströmmen, t.ex. det sista 100 mätvärdena från en sensor eller alla mätvärden under den senaste millisekunden. Allteftersom strömmen fortskrider *glider* fönstret framåt över strömmen. Eftersom dataströmmar kan ha obegränsad längd måste algoritmer som kräver tillgång till en begränsad mängd data för sin analys appliceras på sådana fönster. Vidare behövs fönster för att snabbt upptäcka onormalt beteende hos strömmar från dynamiska system. Ofta samlas statistik från varje fönster m.h.a. frågor som grupperar data m.a.p. en kategori eller *gruppnyckel*, t.ex. uppmätt maximal och genomsnittligt tryck per sensor för ett antal trycksensorer på en maskin(.). Varje sensor har ett heltal som gruppnyckel och dataströmhanteringssystemet upprätthåller kontinuerligt en tabell av statistik per gruppnyckel allteftersom strömmen fortskrider och fönstret glider framåt.

I många fall kan emellertid ingen gruppnyckel identifieras för att kontinuerligt tabulera statistik, t.ex. när man vill identifiera grupper av maskiner med likartat beteende. I sådana fall används istället s.k. klustringsalgoritmer som KMEANs [1] och DBSCAN [2] för att forma grupperna och beräkna statistiken. Sådan klustring av strömmande data är särskilt utmanande eftersom det innebär att grupper dynamiskt skapas, slås samman och delar sig allteftersom strömmen fortskrider. Över dessa dynamiskt formade grupper applicerar algoritmerna därvid statistiska sammanfattningar i realtid [3] [4] [5].

Det mesta av tidigare forskning inom dynamisk klustring av strömmande data är inriktad på att utveckla monolitiska algoritmer där fönster- och indexeringsmekanismer ingår i algoritmerna, vilket resulterar i sammanflätade implementeringar där kod inte kan återanvändas. Till exempel i EXTRA-N [4] och SGS [5] ingår algoritm-specifika index över data i glidande fönster. BIRCH [6] är ett annat exempel på en klusteralgoritm som använder en egen indexeringsteknik, som kallas CF-träd. Att implementera strömmande klustringsalgoritmer från grunden kräver en mycket sällsynt kombination av kompetens. Därför finns behov av ramverk där analytiker kan uttrycka sina dataanalyser på en hög nivå, medan olika klustrings- och indexeringsalgoritmer kan utvecklas oberoende och pluggas in i ramverket [7].

Denna avhandling behandlar det allmänna problemet med sammanfattning av strömmande data i realtid. Följande frågeställningar undersöks:

1. Vilken är en lämplig generell mekanism för glidande fönster för olika sorters strömmande sammanfattningsalgoritmer?
2. Vad är en lämplig indexeringmekanism för datasammanfattning över glidande fönster?
3. Hur kan fönstrets mekanism för framåtskridande separeras från både indexering och den sammanfattningsalgoritm som appliceras för att undvika sammanflätade implementeringar?

Ansatsen är att utveckla ett generellt system G2CS *(Generic 2-layer Continuous Summarization)* där olika sammanfattningsalgoritmer och indexeringsstrukturer kan pluggas in oberoende av varandra. Olika implementeringsalternativ för G2CS har utvärderats.

För forskningsfråga ett analyserar vi i publikation II olika metoder att utföra frågor som grupperar strömmande data från en fotbollsmatch i realtid. En två-fas strategi [8] [9] [10] [11] [12] för strömmande summering över glidande fönster väljs för vidareutveckling. Baserat på denna fallstudie, utvärderas i publikation IV två tillvägagångssätt för att effektivt kunna stödja frågor över kluster med hjälp av en ny fönstermekanism som kallas *Sliding Binary Merge (SBM)*.

Publikation I adresserar forskningsfråga två genom att presentera en generisk ansats för indexering av data i glidande fönster där grupperade data kontinuerligt aggregeras genom att dela upp både data och index i ett glidande fönster. Detta tillvägagångssätt är generaliserat i publikation IV för att indexera de data som krävs för att upprätthålla dynamiska kluster över glidande fönster.

För forskningsfråga tre visas i publikation IV hur G2CS separerar sin mekanism för glidande fönster från de indexerings- och sammanfattningsalgoritmer som pluggats in. Detta förenklar implementeringen och gör det möjligt att plugga in olika klustringsalgoritmer. Således möjliggör G2CS återanvändning av mjukvarukomponenter och förenklar införandet av nya algoritmer.

7 Bibliography

[1] James MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. Fifth Berkeley Symp. on Math. Statist. and Prob.*, 1967, pp. 281-297.

[2] Martin Ester et al., "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Knowledge Discovery and Data Mining (KDD)*, 1996, pp. 226–231.

[3] Feng Cao, Martin Estert, Weining Qian, and Aoying Zhou, "Density-Based Clustering over an Evolving Data Stream with Noise," in *SDM*, 2006, pp. 328-339.

[4] 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.

[5] 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.

[6] 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.

[7] Volker Markl, "Breaking the Chains: On Declarative Data Analysis and Data Independence in the Big Data Era," in *International Conference on Very Large Data Bases (VLDB)*, Hangzhou, 2014, pp. 1730-1733.

[8] 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.

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

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

[11] 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.

[12] 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.

[13] Cheng Xu et al., "Scalable Validation of Industrial Equipment using a Functional DSMS," *Journal of Intelligent Information Systems*, vol. 47, August 2016.

[14] Hillol Kargupta et al., "MobiMine: monitoring the stock market from a PDA," *ACM SIGKDD Explorations Newsletter*, vol. 3, no. 2, pp. 37-46, Jan 2002.

[15] Gyozo Gidofalvi, Torben Bach Pedersen, Tore Risch, and Erik Zeitler, "Highly scalable trip grouping for large-scale collective transportation systems," in *11th international conference on Extending database technology: Advances in database technology*, 2008, pp. 678-689.

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

[17] (2016, Aug) Storm home page. [Online]. http://storm.apache.org/

[18] Mathew Sajee, "Overview of amazon web services," *Amazon Whitepapers*, Nov 2014.

[19] Tyler Akidau et al., "MillWheel: fault-tolerant stream processing at internet scale," in *Proceedings of the VLDB Endowment*, 2013, pp. 1033-1044.

[20] Paris Carbon et al., "Apache Flink™: Stream and Batch Processing in a Single Engine," *IEEE Data Engineering Bulletin*, 2015.

[21] Seyed Jalal Kazemitabar, Ugur Demiryurek, Mohamed Ali, Afsin Akdogan, and Cyrus Shahabi, "Geospatial stream query processing using Microsoft SQL Server StreamInsight," in *Proceedings of the VLDB Endowment* , 2010, pp. 1537-1540.

[22] (2016, Aug) StreamBase. [Online]. http://www.streambase.com/

[23] (Aug, 2016) SQLStream. [Online]. http://sqlstream.com/

[24] 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.

[25] M. Tamer Ozsu Lukasz Golab, "Issues in Data Stream Management," *SIGMOD Record*, vol. 32, no. 2, pp. 5-14, June 2003.

[26] Arvind Arasu et al., "STREAM: The Stanford Data Stream," Stanford, 2004.

[27] Sirish Chandrasekaran et al., "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World," in *SIGMOD*, 2003, pp. 668-668.

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

[29] S. Babu and J. Widom, "Continuous queries over data streams," *ACM SIGMOD Record*, vol. 30, no. 3, pp. 109-120, 2001.

[30] David Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*.: Springer, 2008.

[31] Teuvo Kohonen, "Self-organized formation of topologically correct feature maps," *Biological Cybernetics*, vol. 43, no. 1, pp. 59–69, 1982.

[32] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan, "Automatic subspace clustering of high dimensional data for data mining applications," in *SIGMOD '98 Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, 1998, pp. 94-105.

[33] Cheng Xu, "Scalable Validation of Data Streams," Uppsala University, PhD thesis ISSN 1651-6214, 2016.

[34] Ramez Elmasri and Shamkant Navathe, *Database systems*, 6th ed.: Pearson, 2011.

[35] Lukasz Golab, Shaveen Garg, and Tamer Özsu, "On Indexing Sliding Windows over Online Data Streams," in *International Conference on Extending Database Technology (EDBT)*, 2004, pp. 712-729.

[36] 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.

[37] 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.

[38] 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.

[39] Sudipto Guha, Nina Mishra, and Rajeev Motwani, "Clustering data streams," in *Foundations of computer science, 2000. proceedings. 41st annual symposium on*, 2000, pp. 359--366.

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

[41] 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.

[42] 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.

[43] Tore Risch, Vanja Josifovski, and Timour Katchaounov, "Functional Data Integration in a Distributed Mediator System," in *The Functional Approach to Data Management*. Berlin: Springer, 2004, pp. 211-238.

[44] D. Baskins, "Judy home page [http://judy.sourceforge.net/]," 2003. [Online]. http://judy.sourceforge.net/

[45] Xu Cheng, Daniel Wedlund, Martin Helgoson, and Tore Risch, "Model-based validation of streaming data: (industry article)," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*, 2013, pp. 107-114.

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1431

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology".)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-302799

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2016

Paper I

Scalable ordered indexing of streaming data

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

ABSTRACT

In order to efficiently answer continuous queries requiring range search in large stream windows, data stream management systems (DSMSs) need ordered indexes. Conventional DBMS indexing methods are not specifically designed for data streaming applications with extremely high insert and delete rates for windows over streams. This motivates a scalability investigation for various ordered main memory indexing methods in a streaming environment, through implementation and experiments. Our experimental studies show that a state-of-the-art implementation of cache-aware compact tries is a very suitable indexing structure for data streaming applications allowing constant time insert and access rates. However, in the best of the investigated implementation the range search was slow. Since a highly optimized implementation of compact tries is very complex we developed a framework for scalable range search in an index without any change to its source code. Another important issue is that index maintenance in window based data stream environments require a scalable way of deleting data, which is addressed by an index independent window aware bulk deletion technique, also without changing any source code.

1 INTRODUCTION

A Data Stream Management System (DSMS) usually has a local main memory database against which high volume streaming data is matched. This local database includes storage for windows of data streams flowing through the system. The windows may become large, so indexing data in stream windows is an interesting problem. In many cases ordered indexing is needed, which is investigated here.

The requirements of data stream indexing is not exactly the same as conventional DBMS indexing, causing some traditional DBMS indexing structures to fall behind the requirements of DSMS applications. The following

are important differences between conventional DBMS indexing and DSMS window indexing:

- Because of very high stream rates DSMS indexes need to be stored in main memory and the indexing data structure should be main-memory oriented, i.e. be CPU cache conscious and compact.
- Stream window indexes need to be able to handle very high insert, update, and delete rates. By contrast, most conventional DBMS applications behave based on a high watermark. That is, once the database is filled up, it does not rapidly grow or shrink in size. In other words, DBMS applications have lower demand for massive insertion and deletion than DSMS application while fast search is desirable in both.

In this paper we investigate the performance of different kinds of ordered indexing methods for main memory databases in context of window based stream processing w.r.t. the three aspects of ordered indexing for massive data streams: insertion, search, and deletion. The goal is to find the best suitable ordered sliding window indexing method for massive data streams.

To improve the performance of deletion from indexes over time stamped stream windows, we propose a window aware indexing maintenance method, *partitioned temporal window index (PTWI)*, and through experiments we show that it outperforms a naïve incremental index deletion strategy. In addition PTWI can be used together with any kind of underlying indexing structure.

We implemented and compared the performance of indexing sliding windows over data streams of main memory B-trees, cache sensitive B+ trees [12], burst tries [10], and the highly optimized but complex compact trie implementation Judy [18]. For empirical investigations we used randomly generated synthetic data as well as data generated by the Linear Road Benchmark (LRB) [1] for streaming data. Benchmark queries were used to compare the scalability of insertion, deletion, and range search for different indexing structures.

Judy is a highly optimized compact trie implementation that focuses on both compactness and CPU cache utilization to improve the performance. However, the current Judy implementation lacks efficient range search iteration, as also noted by [15] [16]. To improve memory and CPU cache utilization, Judy dynamically changes between its around 50 different internal node structures based on the current key distribution in each node, which makes the implementation of Judy very complex and difficult to change. We therefore developed a method to improve range search in a complex index implementation such as Judy, without changing its source code. The experimental results show that our extended version of Judy scales the best among the other investigated main memory indexing structures.

This paper is organized as follows. Section two makes an overview of related main-memory ordered indexing methods. Section three first defines the benchmark scenario and then describes required extensions to the indexing methods for range search and massive deletion. Section four evaluates the scalability of the different indexing methods through experiments on implementations. Section five summarizes the result and proposes future work.

2 Background and related work

Sampling techniques like *window aware load shedding* [19] have been proposed for processing approximate queries when the stream rates are higher than the DSMS can handle. Load shedding is not suitable when all stream elements in the window must be maintained, such as in monitoring communication networks [2] and urban traffic [1].

A complement to load shedding is indexing. Proper indexing increases the performance of the DSMS and decreases the need for sampling techniques. We have investigated the performance of the most common main memory ordered indexing structure for our setting. In particular we review different kinds of B-trees and tries.

The compact trie implementation Judy was found to be particularly interesting to investigate. However, Judy needs some extensions for supporting efficient streaming range search and massive deletion. Since the implementation of a highly optimized compact index structure such as Judy is very complex, we have devised methods to improve range search and deletion for an index implementation without altering it.

B-trees

The CSB+ variants of B-trees [12] [9] and the binary T-tree [14] have been proposed to index main memory data in a cache conscious way. A recent study [12] suggests that in the context of in-main-memory indexing on modern processors T-trees do not perform better than classical B-trees. Therefore classical B-trees regained the research focus and there have been attempts to make B-trees cache conscious. By exploiting the CPU cache more effectively, the CSB+ tree improves the search time at the cost of using more space and slightly slower insertion and updates than regular B+ trees [12]. We show that the major problem with CSB+ trees compared to B-trees is space inefficiency.

Tries

In the simplest form, a trie is a multi-way tree structure in which each node is an array of pointers. The size of each array is equal to the number of letters in the alphabet, e.g. 26, and each level in a trie indexes a letter in a word. The main advantage of tries is constant insertion and access time if the

length of the key is fixed. Thus tries should be very well suited for indexing data stream windows with very high insert rates. Figure 1 shows a naïve trie. Each node in the trie represents a *sub-expanse* [18], which is a set of keys that are accessed through it. In Figure 1, all keys in the range [COAAA,COZZZ] are in the same sub-expanse accessed through the node marked as "CO".

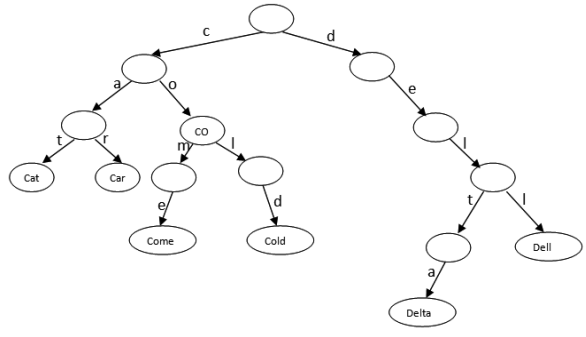


Figure 1. A naïve trie example that stores string keys "cat", "car", "cone", "cold", "dell", and "delta".

Although tries were originally introduced to index character strings, they can be easily modified to index any ordered domain. An order preserving key transformation function can be defined that returns a binary key representing the rank of the original key in the domain. If prior knowledge about the domain exists, such a transformation can be done on-the-fly as done by, e.g., [12]. A binary key can then be indexed by breaking it down into bytes and then introducing them to the trie like characters of a string. For simplicity, here we consider the binary keys to be 32 bit integers broken into 4 bytes. In a naïve implementation for integer keys, the trie is then always 4 levels deep. Each node is a simple array of 256 pointers to the nodes in the next level or, in case of nodes in the 4th leaf level, pointers to values. Tries can be extended to support longer integers and other forms of breaking integers [6].

The memory utilization problem with tries is that they are sensitive to the distribution of keys. In the worst case, when the keys are uniformly scattered across the whole domain, naïve tries waste memory because there will be many null pointers in the sparse pointer arrays representing trie nodes. Several compression techniques have been introduced to overcome naïve tries' weak memory utilization [5] [10] [13] [18]. The main objective in most of them is to achieve a compact representation that, despite its compactness, can still support constant insertion/search time.

A burst trie [10] is based on the idea that as long as the population is low, keys that share the same suffix can be stored in the same *container*. Containers are sorted lists of partial keys together with their associated values. During index lookup, once the right container is found, the key is located using

binary search. Containers have a limited capacity and therefore, in an attempt to insert more keys into a full container depending on the implementation particulars, the container is transformed into a larger internal node, and thus 'bursts' into several new containers. The keys will thereby be redistributed to the new containers based on deeper suffix calculations, and the pointers in the new internal node will refer to new containers. This is an effective approach to decrease memory consumption. However, since the container capacity is fixed in all nodes, the internal nodes often still have null pointers and the memory utilization can still be a problem.

Judy compact trie implementation

Judy [18][3] can be categorized as a variation of burst tries, but with an important distinction: the node (container) data structure and its size is not fixed. To improve memory and CPU cache utilization, Judy dynamically changes node structures according to the current distribution of keys in each node choosing among around 50 different representations of internal nodes. Judy is a highly tuned but very complex data structure. Judy's approach towards an efficient compression technique is to use a variety of compact node structures that fit in a single cache block for different kinds of local sub expanse populations. This allows the contents of any kind of node to be moved to the CPU cache for fast consecutive access. Furthermore, Judy maintains the most interesting characteristic of tries. That is, the depth of Judy is constant, e.g. for indexing integer keys Judy is always 4 levels deep. This means constant time is guaranteed for all single element operations.

Judy supports iteration based range search. However, in the current Judy implementation the iterator always starts at the root, which makes it perform worst among the investigated methods w.r.t. range search. The J+ tree [16] and PJ+ tree [15] address this problem by introducing a sorted linked list as an extra level of leaf nodes. We were unable to obtain the source code for J+ or PJ+ trees for making an empirical evaluation. However, compared to Judy, the J+ tree worsens the performance of single key operations in Judy because it adds an extra level of search and maintenance of the leaf node lists; it also consumes much more memory since prefixes are stored uncompressed in the leaf node lists. The prefetching variant of the J+tree, the PJ+ trees [15], improves the range search performance by adding prefetching pointers, but does not address any of the J+ tree deficiencies.

Non intrusive range search

Implementations of indexing structures for highly tuned indexing structures such as Judy might become very complicated. To improve software reusability and eliminate unnecessary modifications to highly optimized implementations, we use *mappers* as a general method that simplifies traversal of data

structures. A mapper is a second order function that applies a *mapping function* on a set of elements. In the ordered indexing context a mapper is a function that traverses a range of keys specified by low and high bounds, and applies a user provided mapping function on the key-value pairs in the range.

Using mappers we added range search to Judy without any modification to it. We show that the mapper approach substantially improves range search compared to the built-in implementation. This makes Judy extended with mappers perform better than other investigated approaches.

GIST [11] is a general framework for adding tree-based indexes to an extensible DBMS for supporting range search. It is challenging to make the code changes required by GIST for a complex trie structure such as Judy, and we therefore instead used mappers.

Non intrusive bulk deletion for sliding windows

If there is massive stream flow through a tumbling window, deleting the expired stream elements from the window index becomes an issue. Naive element by element deletion is slow. Common methods to speed up bulk insertion and deletion are to use partitioned indexes and create/delete entire partitions in bulk [17] or prefixing keys in a B-tree with partition identifiers [7]. We adapted partitioned bulk deletion to support non-intrusive bulk deletion of indexes over sliding time stamped windows, called *partitioned temporal window index* (PTWI). The main difference to regular bulk deletion is that PTWI maintains a circular array of pointers to time stamped subwindow indexes, which are completely deleted as the main window slides.

3 Ordered indexing of data streams

We address three main challenges in indexing data in sliding windows: scalable insert, fast range search, and scalable deletion. The suitability of several indexing methods w.r.t. these aspects have been investigated. For the investigation of the methods we used own implementations, publically available implementations, and publicly available versions extended with our improvements.

3.1 Scenario

To analyze the problems of maintaining proper ordered indexing structures for window based stream processing and comparing scalability of different indexing solutions, we use the Linear Road Benchmark data generator. It generates for a predefined number of expressways L an input data stream with the following tuples:

[T, X, D, S, VID, VEL]

Where

- T is a time stamp.
- X is the expressway on which a vehicle is traveling, 0 to *L*-1.
- D is the direction in which the vehicle is traveling, which is either east or west.
- S is the segment of the expressway, 0 to 99.
- VID is the vehicle's identifier.
- VEL is the speed of the vehicle.

Our performance evaluation simulates index search for the following index intensive queries:

- Q1: What is the velocity of a specific vehicle *v* on expressway *x* traveling in direction *d* in segment *s* during the last minute? This query selects a single tuple.

```
select VEL
from [last minute window]
where X=x and D=d and S=s and VID =v;
```

- Q2: What is the average velocity of all vehicles on expressway *x* traveling in direction *d* in segment *s* during the last 5 minutes? This query is selecting *1/L %* of the position reports in the window.

```
select average (VEL)
from [last 5 minutes window]
where X=x and D=d and S=s;
```

An ordered index on the compound key <*X, D, S, VID*> provides scalable answers to both queries. The VID attribute needs to be included in the ordered index since, at traffic peaks, LRB generates a large number of vehicles per segment in a minute (around 100,000).

Query Q1 accesses a single element in the index having the key <*x,d,s,v*>.

Query Q2 is a range search where the lower limit of the compound key is <*x,d,s,0*> and the upper <*x,s,d,*∞>.

Since the main window covers 5 minutes and tumbles every 1 minute, the older data on the index must be removed, which requires massive deletion from the index.

3.2 Improving range search on Judy

The most common way to iterate over index ranges is to use a Volcano style scan structure with a *next* method [8]. Such a structure is indeed available in Judy, but it does not perform well because the *next* method always starts from the root in the current implementation, without using a scan data structure. For scalable range search, Judy has to be modified. However, to im-

plement a scan data structure in a highly complex indexing structure such as Judy, having over 50 different node types, is a challenging task since all state information has to be continuously maintained in the scan. The alternative to implement scans using linked leaf nodes as in B+ trees would require substantial modifications of Judy with unknown consequences.

To add efficient range iteration to Judy without the complexity of implementing scans, we instead implemented a second order C *mapper* function that applies another C function on every key-value pair in a given key range. This approach requires no change to Judy and no explicit code to maintain the complex state information as in scans. Our implementation also supports generic iteration over scans by using threads combined with a buffer of recently mapped key-value pairs.

Listing 1 shows the general signatures for mapper and mapping functions in C for range search operations in an ordered indexing structure.

```
typedef int (*mapping) (key k, value v,
                        void *xa);
Mapper(indexroot* tree, key lower, key,
       upper, mapping m,void *xa);
int SumMapping(key* k,value* v,void *xa)
{
  *(int *) xa += (int) *v;
  return TRUE;
}
```

The following code traverses the index structure pointed to by `tree` in the range [100,200] and applies `SumMapper` to all key-value pairs in the range. The sum of the values are accumulated in the variable *sum* passed by reference to the mapper.

```
key k1=100; k2=200;
int sum=0;
indexroot* tree=new_index();
Mapper(tree, k1, k2, SumMapping, &sum);
```

Listing 1. general mapper and mapping functions for range search

Based on the general mapper paradigm, we implemented a mapper function for Judy that performs the range search. The mapper recursively visits the nodes that cover sub expanses which are within the specified range. For each leaf node, it applies the mapping function to the key-value pairs in the leaf nodes that are within the range. In Judy the bytes of the key are not always implicitly stored in nodes, so the algorithm has to carry a prefix at any call

level. Listing 2 provides an outline of the algorithm (the C code can be downloaded from [21]).

```
JudyMapper(Judypointer jp, key lower, key
upper,  key prefix, mapping fn,void *xa)
{
  switch (type (jp) )
  {
    case internal_nodes:/* many variants
         of linear,bitmap,uncompressed */
      for all Judy pointers p in each
              internal node that covers
              the range [lower, upper]do
      {
        Update the prefix;
        JudyMapper(p, lower, upper,
                   prefix, fn, void *xa);
      }
    case leaf_nodes:/* linear, bitmap or
                       immediate leafs */
      for all keys k inside range
              [lower, upper]do
      {
        Construct the key by extending
           prefix;
        Find value v associated with k;
         (*fn)(k,v,xa); /* apply mapping
                           function */
      }
  }
}
```

Listing 2. Judy mapper

3.3 Window aware index deletion

We compare two different strategies for deleting time stamped elements from indexes over sliding windows: naive incremental deletion and the bulk window index deletion method PTWI.

3.3.1 Incremental deletion

In incremental deletion there is only one indexing structure for the whole window. In order to identify the right set of keys to be deleted, the time

stamp has to be explicitly stored as a part of the key. The index key thus takes the form of <*t, k*> where *k* is the application key (i.e. <*X, D, S, VID*> in LRB) and *t* is the time stamp associated with it. Notice that the order in the compound key proposed here preserves the temporal order of keys. Therefore, deletion is straight forward; after the time stamp *t* expires, all keys of form <*t,* *> need to be removed. Since an ordered indexing structure is used, all keys in this range are found and then deleted from the index one by one.

Naive incremental deletion of keys one by one might take considerable amount of time since the data structure is searched from the root for each deleted key.

3.3.2 **Bulk window index deletion**

As an alternative to incremental deletion we also implemented a special bulk deletion technique for sliding time stamped windows called *partitioned temporal window index* (PTWI).

PTWI is applicable for sliding windows. Let *N* be the time span of the window and *S* be the stride for the sliding in time units. At each slide a subwindow of size *M*=*N*/*S* tumbles. In LRB *N*=300 seconds and *S*=60 seconds, thus *M*=5. With PTWI *M* non-overlapping partial indexes are maintained for the whole sliding window. When the window slides, the partial index that stores the oldest subwindow is dropped and a new empty partial index is created. PTWI is implemented as a one dimensional circular index array of size *M* of pointers to partial indexes, as illustrated by Figure 2.

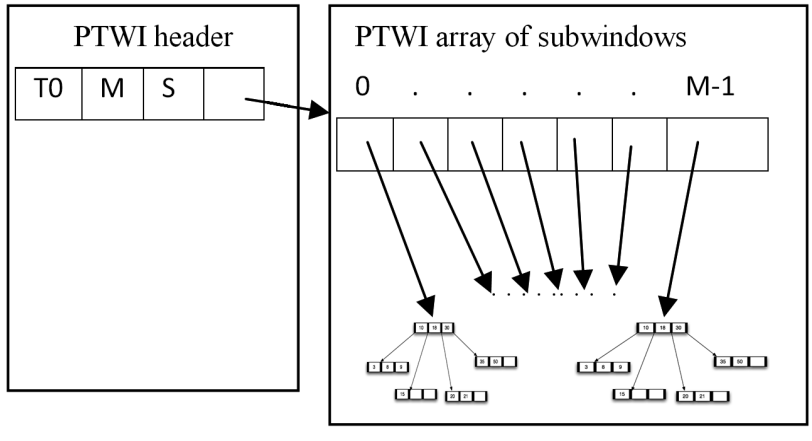


Figure 2. The PTWI structure

In the *PTWI header* the following information is maintained as the window slides:

T0: Starting time for the indexed stream, initialized to the time for the first arriving tuple.

M: Number of subwindows. In LRB *M*=5.

S: The stride of the subwindows as time units. In LRB *S*=60 seconds.

When a new tuple with time stamp *t* and data tuple *tpl*, <*t, tpl*>, arrives in the stream, the system first determines whether tumbling of a subwindow is needed or not. Tumbling is needed when *mod(t,S)=0*.

a) If *mod(t,S)≠0*, i.e. no tumbling, the system computes the position *i* in the subwindows array containing a pointer to the subwindow index where *tpl* should be inserted, accessed, or updated:
   $i = mod(t-T0,W*S)$
b) When mod(t,S)=0 the oldest window tumbles by completely dropping it from the subwindow array and replacing it with a new empty window index. The position d in the subwindows array for the window index to replace is computed by:
   $d = mod(t/S,W)$

For example, Figure 3 illustrates the evolution of the subwindows array for the LRB scenario. In the beginning of any minute *T*, the oldest partial index associated to minute *T*-5 needs to be dropped and a new empty one for minute *T* is created. Figure 3 shows the content of the subwindows array during minutes 1 to 10. In each minute *T* incoming data is inserted only to the index associated with the current minute, tagged as *@T* in the figure.

Queries that access a single tuple at a given time point *t*, such as Q1, can be directly answered by calculating *i* as in a) and then accessing window index *i* in the PTWI array of subwindows.

To answer queries that cover the whole window, they have to be divided into sub-queries – one for each minute – and their results merged. For example, for query Q2 the time period is the last 5 minutes and therefore the range query [<*x,d,s,0*>, <*x,s,d,∞*>] for given *x, s,* and *d* is issued over all 5 subwindow indexes in the array.

Notice that bulk deletion can be done in a lazy manner in a background process. In other words, deletion is no longer a burden on the real time expectations of the system.

Furthermore, notice that any kind of indexing structure can be used for storing the subwindow indexes.

The space overhead of PTWI compared to incremental deletion is negligible, since it just adds one extra array of *M* pointers and the PTWI header. The computational overhead is one extra simple numerical computation per stream tuple to obtain *i*, while *d* is computed only when the window tumbles.

The PTWI's window index array					minute
@1	nil	nil	nil	nil	1
@1	@2	nil	nil	nil	2
@1	@2	@3	nil	nil	3
@1	@2	@3	@4	nil	4
@1	@2	@3	@4	@5	5
@6	@2	@3	@4	@5	6
@6	@7	@3	@4	@5	7
@6	@7	@8	@4	@5	8
@6	@7	@8	@9	@5	9
@6	*@7*	*@8*	*@9*	*@10*	*10*

Figure 3. Contents of the *PTWI* array of subwindows during first 10 minutes, with 5 minutes window size and 1 minute stride. *@T* represents the pointer to the subwindow index for minute *T*.

4 Experimental evaluation

We experimentally compared the scalability of insertion, single element retrieval, incremental deletion, bulk deletion, and range search in a B-tree, CSB+tree, Burst trie, Judy, and Judy extended with efficient range search. We also compared the performance of PTWI with incremental deletion for Judy and B-trees. The outcome supports the initial hypothesis that Judy extended with efficient range search and PTWI outperforms other investigated in-main-memory indexing structures. The succeeding sections describe how tests were performed and present experimental results.

4.1 Experimental setup

The following ordered indexing methods were investigated:

- We implemented the classical B-tree algorithm as in [4]. Then we experimentally tuned the B-tree node size to minimize cache misses, which on our hardware happened when each B-tree node contained 750 bytes.
- The CSB+ tree implementation was downloaded from [20]. We used the full CSB+ tree variant, which is the best variant for high insertion and update rates according to the authors.
- We implemented the burst trie index as specified in [10] adapted for integer keys.
- Judy was downloaded from [3].
- To have fair comparisons, in all indexing structures both keys and values are 32 bit integers.

The C code for indexing methods used in the experiments is available at [21].

We performed two sets of experiments with different key distributions. The first key distribution consists of uniformly distributed random integers from the whole 32 bit integer range. This is the worst case for tries since under this key distribution the trie structure will have sparse pointer arrays with many null pointers. In our second key distribution the keys are the position reports from the more realistic LRB input stream.

Given that the intention was to compare the scalability of indexing structures, the size of the indexes was gradually increased in a number of steps, and then the required time to perform insertion, single element retrieval, and range search operations were measured. After the each step the amount of main memory so far used by each index is noted. Moreover, to typify the measurement, the operations were done in batch, i.e. instead of measuring the time of inserting a single key in each step, which could be affected by noise, we calculated the average time to insert an element over 0.5 million keys. Since we measure pure insertion, update, and retrieval time duplicated keys in the input are omitted.

The performance of deletion is measured in a separate experiment.

4.1.1 Random key distribution

In this experiment first 16 million random keys from a flat distribution of integers in range $[0,2^{32}-1]$ were generated and stored in a one dimensional array. The experiments were performed by reading the keys from the flat array as follows:

In steps of size 0.5 million simulated incoming tuples, perform the following actions and measure the time each one takes on all indexing structures:

1. Insert into the index 0.5 million keys and measure the average time to insert one key.
2. Measure the accumulated amount of main memory used by each indexing structure after each 0.5 million inserts.
3. Retrieve in random order 50000 keys from all so far inserted keys and measure the average time to retrieve one key from the index.
4. Generate a random interval covering 10% of the total domain and make one range search to measure the time.

4.1.2 LRB key distribution

For a realistic data distribution, the indexing keys in the second experiment were LRB position reports. To construct ordered preserving integer keys k for LRB, they are computed as follows:

$k = VID + SEG*2^{20} + D*2^{27} + X*2^{29}$ i.e.:

- Expressway number X, the most significant 4 bits (28-31).

- Direction *D*, bit 27.
- Segment *SEG*, bits 20-26.
- Vehicle identifier *VID*, bits 0-19.

First the whole LRB input file is scanned; the first appearance of each key *k* is stored in a flat array. During this preprocessing, duplicates are detected and discarded.

After forming the flat array containing unique keys, the test is executed in a number of steps similar to the procedure for random keys, but steps three and four are different:

- In step three, query Q1 is executed for 50000 randomly chosen so far inserted position reports and the average single element retrieval time is measured.
- In step four, for randomly chosen *x, s,* and *d,* query Q2 is executed 100 times and the average search time measured.

4.1.3 Deletion

In order to compare the scalability of incremental window deletion with PTWI, we performed two tests on the two indexing structures Judy and B-trees.

The first test measures a naive incremental deletion strategy. In this test the indexes were loaded with different populations of keys as in 4.1.2. For each population all individual keys are deleted one by one, until the index becomes completely empty. The total time to empty an index with a given population is measured.

To measure deletion with PTWI, indexes with the same sizes as in the first test were populated, but this time, in contrast to deleting individual keys as in the first test, the whole index structure is dropped at once by traversing all nodes in it.

In both tests, to avoid memory fragmentation issues from biasing the performance, the application is restarted before any new index is created, i.e. before any new population is examined.

All experiments were run under Windows 7 on an Intel (R) Core(TM) i5 760 @2.80GHz 2.93 GHz CPU with 4GB RAM, single threaded using the Visual Studio 10 32 bits C compiler.

4.2 Experimental results

In this section the experimental results from comparing the indexing structures w.r.t. insertion, single element retrieval, memory utilization, range search, and deletion are analyzed. Experimental results of each indexing operation are presented and discussed under LRB key distribution alongside the random distribution. In the deletion section only LRB key distribution is

presented since the results from random key distribution leads to the same conclusions.

4.2.1 Insertion

Figure 4 illustrates the time required to insert a single key into each indexing structure at a given index size when keys from LRB are used. The time is averaged over 0.5 million insertions.

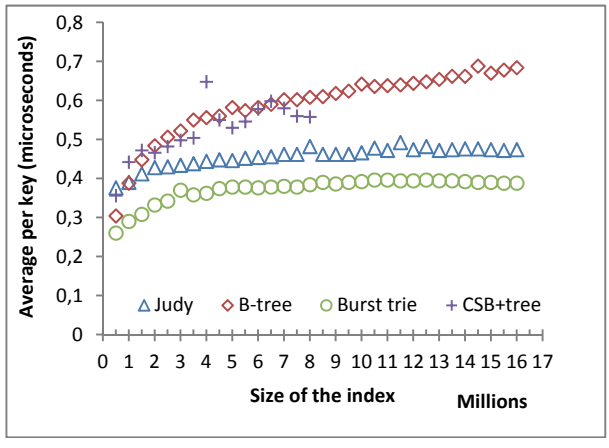


Figure 4. Insertion under LRB key distribution

The most important observation is that, in this key distribution, after around 4 million keys, Judy and burst tries reach their maximum depth of 4. Recall that in our experiments the keys are 4 byte integers and therefore their overall structure stabilizes. From this point on, it takes constant time to insert new elements into Judy and burst tries. The reason burst tries are faster than Judy is that their containers are not compressed, so insertion into them is simpler and computationally cheaper compared to Judy, where insertion into containers causes nodes to transform their representations.

As expected, B-trees and CSB+ trees scale logarithmic. CBS+ trees outperform B-trees w.r.t. insertion because of two reasons: First, all siblings of a node in a full CSB+ tree are allocated as soon the node is created, which reduces the costs of future node creation and potential structural balancing. Second, the CBS+ tree representation is cache conscious. However, after 8 million keys there is no more memory available in our 32 bit representation for the full CSB+ tree to grow.

Figure 5 illustrates the time required to insert a single key into each indexing structure at a given index size when keys are picked from a random distribution. The time is averaged over 0.5 million insertions.

As expected, B-trees and CSB+ trees scale logarithmic and show no sensitivity to the key distribution.

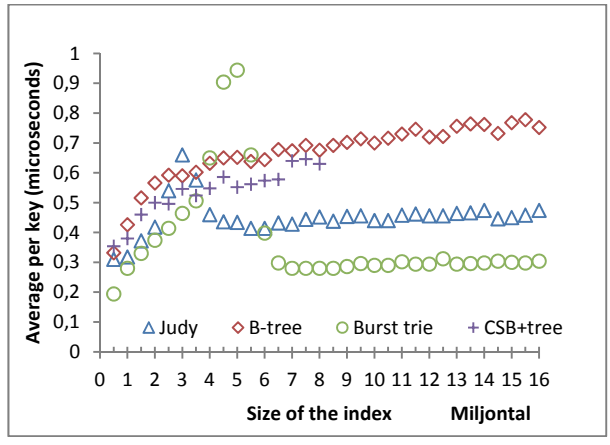


Figure 5. Insertion under random key distribution

Under the random key distribution, in particular burst tries and, to a lower extent, Judy undergo an unstable period when the size is around 3 and 2 million keys, respectively. The fluctuation is due to the specific conditions under which bursting happens. That is, since the keys are uniformly distributed within a very wide range, they rarely share prefix at the next level, so during the burst, new containers are created for most of the keys that are being re-distributed. The high computation and memory management costs involved results in poor performance during bursting.

It is worth to note that Judy stabilizes much earlier than the burst trie. This is because Judy maintains dynamic container structures depending on the population of each sub-expanse, which decreases the bursting cost.

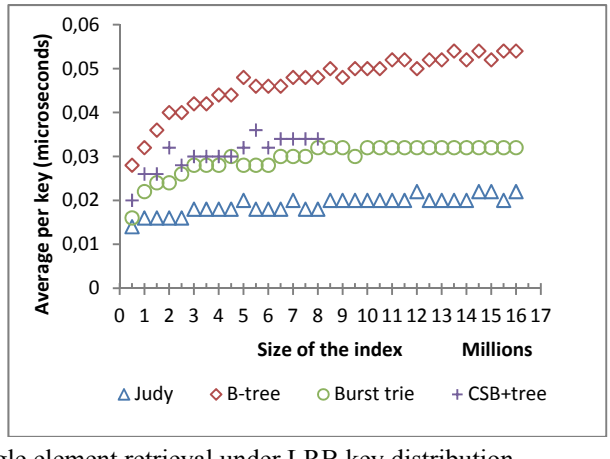


Figure 6. Single element retrieval under LRB key distribution

4.2.2 Single element retrieval

Figure 6 illustrates the time required to retrieve a single key from each indexing structure at a given index size under the LRB key distribution. B-

trees and CSB+ trees scale logarithmic as expected, with CSB+ trees being faster mainly because of cache awareness. Retrieval of single elements from Judy and the burst trie takes constant time after they reach their maximum depth at four levels. Judy is fastest due to two reasons: first its efficient compression techniques facilitate search in the nodes. For example, bitmap nodes store only populated sub-expanses and index them using a directly accessed bitmap. Second, it exploits the CPU cache more efficiently.

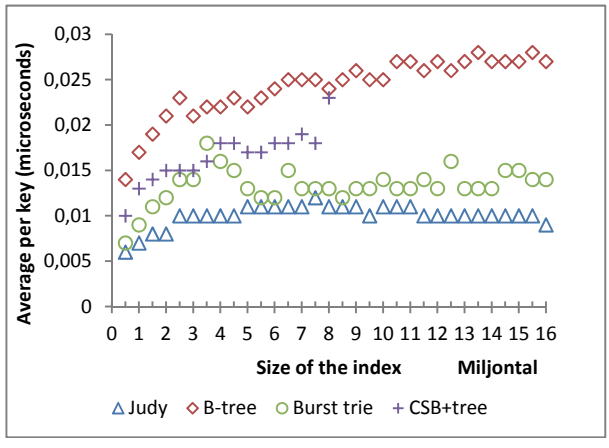


Figure 7. Single element retrieval under random key distribution

Figure 7 illustrates the time required to retrieve a single key from each indexing structure at a given index size under the random key distribution. Again, since B-trees and CSB+ trees are not sensitive to key distribution, they scale similar to the LRB key distribution in Figure 6. The search time for burst tries increases until a maximum at around 3 million keys, after which the retrieval performance improves. The peak happens almost right before the nodes burst. Before the bursting happens, most of the containers are highly populated and therefore performing binary search in them becomes costly. After the bursting happens, keys are distributed among containers with lower populations and therefore the cost for binary search decreases. In Judy, in contrast to burst tries, due to the maintenance of dynamic population-based node structures, the search time at each node is optimized and therefore Judy is much more stable than burst tries.

4.2.3 Memory utilization

Figure 8 and 10 show the memory utilization with LRB and random distributions, respectively. The inefficient memory utilization of the CSB+ tree implementation is displayed separately in Figure 9.

To illustrate the compactness of indexing structures the main memory utilization is measured and displayed in terms of byte per key-value-pair (B/KVP). As a theoretical base line we also plot **flat arrays** in which KVPs

are stored un-indexed in consecutive memory cells. Since all investigated indexing structures use 32 bit integers for both keys and value-pointers, storing KVPs in such a flat array –independent from the number of KVPs- achieves 8 B/KVP, the minimum uncompressed memory area needed to store key-value pairs.

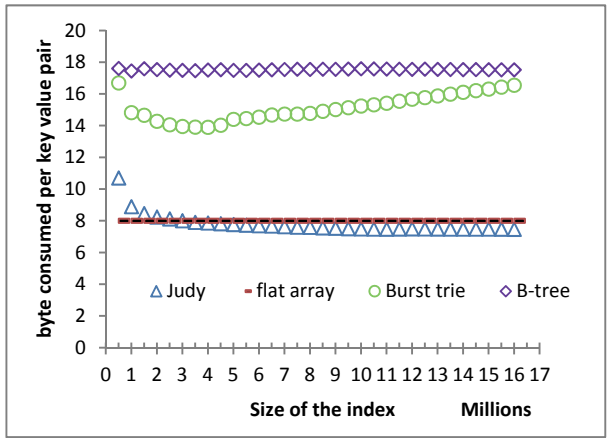


Figure 8. Memory utilization under LRB key distribution

Figure 8 shows the memory utilization when the LRB key distribution is used. It is worth to note that after a certain population, Judy becomes even more space efficient than flat arrays. The reason is that as the LRB simulation proceeds, the traffic increases. This means more vehicle ids per segments of expressways and consequently a more dense key distribution. Judy's dynamic node structure utilizes this to minimize memory consumption mainly through using bitmap nodes and leafs [18].

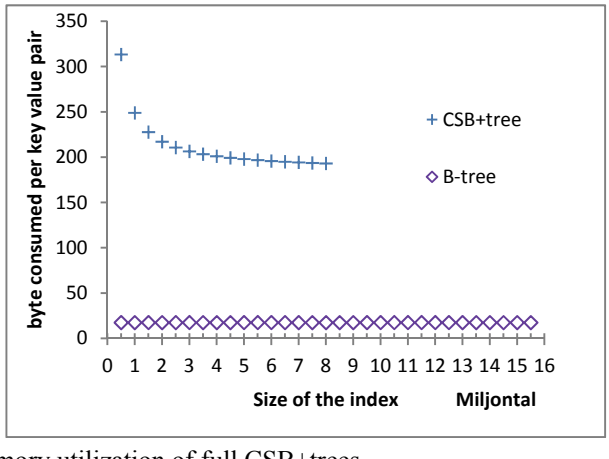


Figure 9. Memory utilization of full CSB+trees

The burst trie on the other hand improves the memory utilization up to a point where the number of keys in sub-expanses exceeds the maximum container size and the containers burst. Each bursting brings about an additional node, and many new containers, which leads to bad memory utilization.

The memory utilization of B-trees is very stable but not as compact as Judy.

Figure 9 shows the memory utilization for CSB+ tree compared to the B-tree for the LRB distribution. The main reason for poor memory utilization of the Full CSB+ tree is that, as described in [12], it creates all sibling nodes for each node to improve insertion and update time. As the total number of keys increases, the memory utilization improves, but since creating new nodes is essential, and all the sibling nodes are also allocated at the time of creating any new node, the improvement is limited.

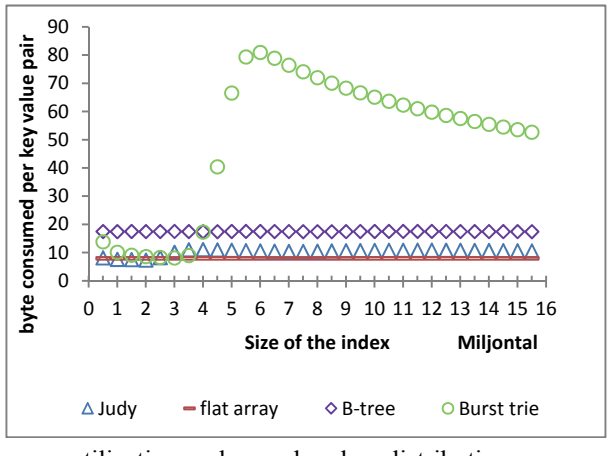


Figure 10. Memory utilization under random key distribution

Figure 10 shows the memory utilization when random key distribution is used. As expected, since B-trees are insensitive to the key distribution, they make the same B/KVP as in Figure 8.

Under random key distribution, burst tries go through extreme bursting and they have even worse memory utilization. This is due to that most containers created by the burst for a random key distribution contain a single element. As the experiment proceeds, more keys with the same prefix are added to the newly created containers, and therefore memory utilization of burst tries improves.

Judy shows very little sensitivity to random key distributions, because in Judy containers with very low populations are implemented using a very specific compact structure - the immediate pointers. In immediate pointers the contents of nodes and leafs with one or two keys are stored in the pointer itself. The immediate pointers make the memory utilization become stable very soon. With random key distribution Judy's memory utilization is

slightly worse than flat array and substantially better than the other indexing structures.

4.2.4 Range search

Figure 11 illustrates for LRB key distribution the time required to perform a range search at a given index size using B-trees, burst tries, CSB+ trees, the original iterator based Judy implementation, and Judy extended with the mapper for range search. As expected, our Judy mapper outperforms the original range search provided by Judy and it performs almost as efficient as a B-tree. The reason for the slightly better performance of the B-tree range search over Judy is that in B-trees a single node stores more keys compared to Judy for two reasons. First, Judy utilizes the most compact representation at each node which leads to nodes with lower populations. Second, compared to the rather large nodes in our B-tree, nodes in Judy are generally smaller since they need to fit in a 64 byte cache line.

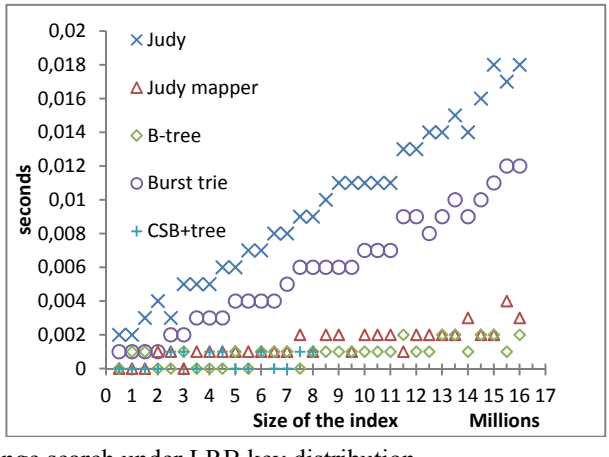


Figure 11. Range search under LRB key distribution

Figure 12 illustrates for the random key distribution the time required to perform a range search using original and extended Judy, B-trees, CSB+ trees, and burst tries at a given index size. The range search scales worse using Judy compared to B-trees with random key distribution, because Judy creates huge number of very small *immediate* nodes.

In both Figure 11 and Figure 12, the Judy mapper implementation scales better than the burst trie mapper. The reason is that the internal nodes in burst tries include many null pointers, which increases the traversal time. That is, the internal nodes in burst tries always contain 256 pointers, even though many of them represent empty sub-expanses, and consequently, the burst trie mapper needs to consider all sub-expanses within the [low-high] range, including the empty ones. The more compact representation of Judy avoids many unnecessary memory accesses and cache misses. It should also

be noted that the CSB+ trees in both cases are not faster than B-trees for range search. The main reason is the larger node size of B-trees, which utilizes the CPU cache better since the leaf nodes are very short in CSB+ trees compared to B-trees.

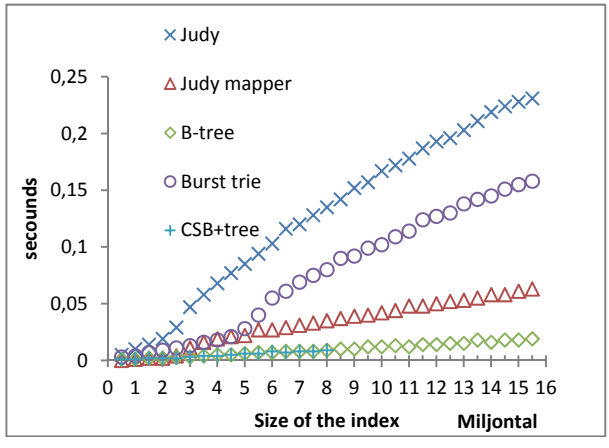


Figure 12. Range search under random key distribution

4.2.5 Deletion

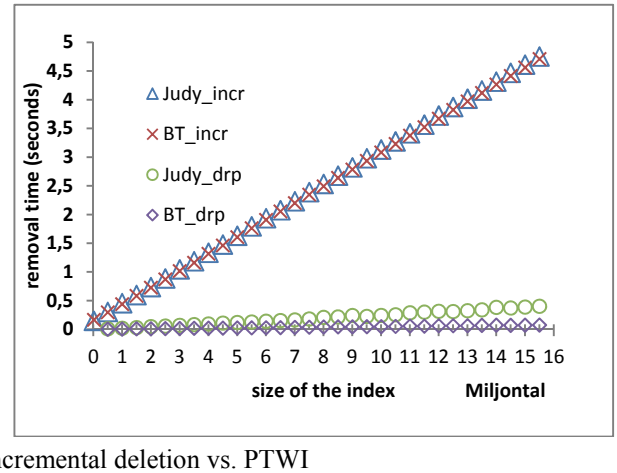


Figure 13. Incremental deletion vs. PTWI

Figure 13 measures PTWI performance using B-tree and Judy subwindow indexes. As expected, removing an index structure by deleting elements one by one, as in the incremental deletion, requires much more time than dropping of the whole indexing structure, as in PTWI. Notice that dropping a Judy index of a given size takes more time than dropping a B-tree index of the same size. This is due to the higher number of nodes in Judy, which needs more time to be traversed and freed. However, since in PTWI it is possible to perform the index drop operation at a background thread in a lazy

manner, slight performance improvements in the index drop operation has insignificant contribution to the overall performance of a DSMS.

4.2.6 Discussion on experimental results

The following main aspects are involved in selecting the suitable ordered indexing structure for window based data streams: insert time, retrieval time, range search time, deletion time, and memory utilization.

Since in many DSMS applications, the input stream is massive, supporting scalable insertion is the most important aspect of indexing structures for sliding windows of data streams. From this perspective, a combination of constant and stable insert time is desired.

As important as insert scalability is scalability of deletions, which was addressed by PTWI. With the PTWI the scalability of deletion becomes non-critical.

The next important aspect is scalability of single element retrievals. To meet the real time requirements of continuous query processing, constant and stable retrieval time is essential.

Maintaining an index over a sliding window of massive streams requires indexing structures with acceptable memory utilization. An indexing structure with poor memory utilization restricts the stream rate that can be handled by the DSMS.

Finally in many applications, e.g. computer network and urban traffic monitoring applications, range search is needed for answering ad hoc queries from the current window.

Table 1. Qualitative summary of the experimental results.

	B- tree	CSB+ tree	Burst trie	Judy	Extended Judy
memory utilization	good	worst	bad	best	best
insertion	good	good	best	best	best
key access	good	good	good	best	best
range search	best	best	bad	worst	good
predictablity	best	good	worst	best	best
simplicity	best	bad	best	worst	worst

Table 1 summarizes the results of our experiments in a qualitative manner. For memory utilization Judy is clearly the best because of its extensive compression. Judy is also the best on insertion and single key accesses. For range searches the tree based indexing methods are the best, but the extended Judy

is very close in particular for non-random data. Burst tries are not stable in general and therefore problematic for streaming applications. The main disadvantage with Judy is its very complex implementation.

To conclude, in the context of indexing sliding windows of streams, our extended version of Judy outperforms all other indexing structures.

5 Conclusions and future work

Window based ordered indexing of data streams has additional requirements to traditional DBMS indexing. Other than scalable range search and individual element retrieval, it is essential for DSMS indexing structures to support scalable insertion and deletion under high volume stream rates. We investigated a number of data stream indexing methods by implementing them or extending available implementations. We compared the following index structures: B-trees, burst tries, CSB+ trees, and the advanced Judy implementation of compact tries.

Through extensive experiments we showed that, in the context of window based indexing of data streams, compact tries with improved range search capabilities outperform other investigated methods by consuming much less main memory, supporting constant access time for insertion and retrieval, and being capable of performing scalable range search. The combination of the above characteristics makes extended compact tries the best ordered indexing method for window based stream processing.

The performance of massive deletion is also very important when indexing high volume data stream windows, which was addressed by PTWI.

Highly tuned scalable indexing structures like Judy might become extremely complex, making any modification very costly. Therefore, we devised non-intrusive methods to add functionality to an existing index structure. PTWI and mappers are two examples of nonintrusive additions that enabled us to successfully re-use, improve, and integrate industry strength software.

In particular, our extended version of Judy with mapper-based range search and PTWI outperforms the other investigated indexing structures, making it attractive for sliding stream window indexing in general.

In addition to supporting range search, tries support more general pattern matching operations that might be useful in pattern recognition in data streams. Therefore, adding pattern matching to tries will be a valuable future work.

6 ACKNOWLEDGEMENTS

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

7 REFERENCES

1 Arasu, A., Cherniack, M., Galvez, E. et al. Linear road: a stream data management benchmark. In *VLDB '04 Proceedings of the Thirtieth international conference on Very large data bases* (2004).

2 Babu, S. and Widom, J. Continuous queries over data streams. *ACM SIGMOD Record*, 30, 3 (2001), 109-120.

3 Baskins, D. Judy home page [http://judy.sourceforge.net/] (2003).

4 Bayer, R. and McCreight, E. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1 (1972), 173–189.

5 Bentley, J. L. and Sedgewick, R. Fast algorithms for sorting and searching strings. In *SODA '97 Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms* (1997).

6 Boehm, M., Schlegel, B., Volk, P. B., Fischer, U., Habich, D., and Lehner, W. *Efficient In-Memory Indexing with Generalized Prefix Trees*. 2011.

7 Graefe, G. B-tree indexes for high update rates. *ACM SIGMOD Record*, 35, 1 (2006), 39 - 44.

8 Graefe, G. Volcano-an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6, 1 (1994), 120-135.

9 Hankins, R. A. and Patel, J. M. Effect of node size on the performance of cache-conscious B+-trees. In *Proceedings of the 2003 ACM SIGMETRICS* (2003).

10 Heinz, S., Zobel, J., and Williams, H. E. Burst tries: a fast, efficient data structure for string keys. *ACM Transactions on Information Systems*, 20, 2 (2002), 192 - 223.

11 Hellerstein, J. M., Naughton, J. F., and Pfeffer, A. Generalized Search Trees for Database Systems. In *Proceedings of the 21st VLDB Conference* (Zurich, Switzerland 1995).

12 Jun R., Kenneth A. R. Making B+- trees cache conscious in main memory. In *MOD* (Dallas TX 2000), Proceedings of the 2000 ACM SIGMOD international conference on Management of data.

13 Kurtz, S. Reducing the Space Requirement of Suffix Trees. *Software – Practice and Experience*, 29 (1999), 1149--1171.

14 Lehman, T. J. and Carey, M. J. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the 12th VLDB Conference* (1986), Proceedings of the Twelfth International Conference on Very Large Data Bases.

15 Luan, H., Du, X., and Wang, S. Prefetching J+-Tree: A Cache-Optimized Main Memory Database Index Structure. *Journal of Computer Science and Technology*, 24, 4 (2009), 687-707.

16 Luan, H., Du, X., Wang, S., Ni, Y., and Chen, Q. J + -Tree: A New Index Structure in Main Memory. In *Advances in Databases: Concepts, Systems and Applications*. Springer Berlin, Heidelberg, 2007.

17 Oracle®. Oracle® Database VLDB and Partitioning Guide. *Available at http://docs.oracle.com/cd/B28359_01/server.111/b32024/part_admin.htm*.

18 Silverstein, A. Judy IV Shop Manual. *Available at http://judy.sourceforge.net/doc/shop_interm.pdf* (2002).

19 Tatbul, N. and Zdonik, S. Window-aware load shedding for aggregation queries over data streams. In *VLDB '06 Proceedings of the 32nd international conference on Very large data bases* (2006).

20 http://www.cs.columbia.edu/~kar/software/csb+/

21 http://www.it.uu.se/research/group/udbl/DSMSOrderedIndexing/

Paper II

Grand Challenge: Implementation by Frequently Emitting Parallel Windows and User-Defined Aggregate Functions

Sobhan Badiozamany
Lars Melander
Thanh Truong
Cheng Xu
Tore Risch
Department of Information Technology, Uppsala University, Sweden
Emails: Firstname.Lastname@it.uu.se

ABSTRACT
Our implementation of the DEBS 2013 Challenge is based on a scalable, parallel, and extensible DSMS, which is capable of processing general continuous queries over high volume data streams with low delays. A mechanism to provide user defined incremental aggregate functions over sliding windows of data streams provide real-time processing by emitting results continuously with low delays. To further eliminate delays caused by time critical operations, the system is extensible so that functions can be easily written in some external programming language. The query language provides user defined parallelization primitives where the user can express queries specifying how high volume data streams are split and reduced into lower volume parallel data streams. This enables expensive queries over data streams to be executed in parallel based on application knowledge. Our OS-independent implementation was tested on several computers and achieves the real-time requirement of the challenge on a regular PC.

Categories and Subject Descriptors
H.2.4 [**Database Management**]: Systems – *Parallel databases*, *Query processing*

Keywords
Parallel data stream processing; continuous queries; spatio-temporal window operators.

1 INTRODUCTION
Monitoring a soccer game requires a system than can process, in real-time, large volumes of data to dynamically determine physical properties as they appear. This requires a system having the following properties:

- To keep up with the very high data flow the system must deliver high throughput while processing expensive computations over high volume data.
- Response in real-time requires continuous delivery of query results with low latency.

- Continuous identification of physical phenomena, such as moving balls and players, requires complex spatio-temporal algebraic computations over windows.

Our EPIC (Extensible, Parallel, Incremental, and Continuous) DSMS provides very high throughput and low latency through parallelization, extensibility, and user defined incremental aggregation of windowed data streams. The high level query language provides numerical data representations and data stream windows as first class objects, which simplifies complex numerical computations over streaming data and enables automatic query optimization. To provide very high performance of low level numerical and byte processing functions the system is easily extensible with user defined functions over streams and numerical data, which allows accessing external systems and plugging in time-critical user algorithms.

EPIC extends the SCSQ system [9] with several kinds of data stream windows and incremental evaluation of user-defined aggregate functions over the windows. In particular the window operator *FEW* (Frequently Emitting Windowizer) decouples the frequency of emitted tuples from a window's slide.

To process expensive queries with high-throughput and low latency the system provides application specific stream parallelization functions where general *distribution queries* specify how to parallelize and reduce outgoing data streams.

2 THE EPIC APPROACH

First FEW and its incremental user-define aggregation are presented in sections 2.1 and 2.2, and then the solution is outlined in section 2.3.

Figure 1 shows the overall data stream flow of the implementation. The thickness of the arrows in all data flow diagrams in this paper correspond to the relative volume of the data streams. Each node in the dataflow diagram is a separate OS process, called a *query processing node,* in which a partial continuous execution plan is running. The topology of the dataflow diagram is completely expressed in the query language where it is possible to specify continuous sub-queries running in parallel [9]. The system automatically creates OS processes running the execution plans of the sub-queries and the communication channels between them (local TCP). In the Grand Challenge implementation, the query processing nodes all run on the same computer and the OS is responsible for assigning CPUs to the processes. The system can also distribute query processing nodes over several computers but those features are not used here.

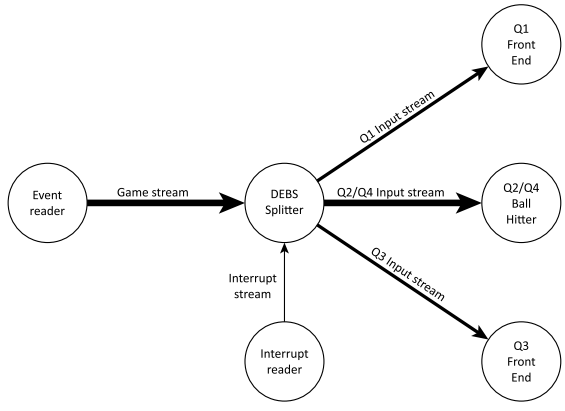


Figure 1. High level data stream flow

2.1 Frequently Emitting Windowizer, FEW

EPIC provides window forming operators that support several kinds of windows, including time, count, and predicate windows [5][2][7]. The windows are formed by *window functions* mapping streams to streams of objects of type *Window*. For example, the window function

```
tWindowize(Stream s, Number length, Number stride)
          -> Stream of Window ws
```

forms a stream *ws* of timed windows over a stream *s* where windows of *length* time units (seconds) slide every *stride* time units. To avoid copying, the windows are represented by pointers to their first and last elements. When a window slides the pointers are updated.

A naive implementation of *tWindowize()* would emit tuples only when the formed windows slide. This causes substantial delays, in particular for large windows. For example, when forming a 10 minutes window, it is not practical to wait 10 minutes for the aggregation to be emitted. To be able to emit aggregation results before a complete window is formed, we have introduced a window function having a parameter *ef*, the *emit frequency*:

```
fewtWindowize(Stream s, Number length, Number stride,
              Number ef)
          -> Stream of Window pw
```

The window forming function *fewtWindowize()* forms partial time windows*, pw,* every *ef* time units. The emitted partial windows are landmark subwindows of the elements of the window being formed. When the formed window is complete it is emitted as well before it slides, and then the landmark is reset to the start time of the newly slided window.

The FEW windows are required when:

- The results must be emitted before the window is formed.
- The results must be emitted more often than the slide (not used in this application).

2.2 User-defined incremental window aggregate functions

The windowing mechanism in EPIC supports incrementally evaluated user defined aggregate functions [1][8]. These are defined by associating *init()*, *add(),* and *remove()* functions with a user defined aggregate function:

- *init() -> Object o_new* creates a new *aggregation object, o_new*, which is used for accumulating changes in a window.
- *add(Object o_cur, Object e) -> Object o_nxt* takes the current aggregation object *o_cur* and the current stream element *e* and returns the updated aggregation object *o_nxt.*
- *remove(Object o_cur, Object e_exp) -> Object o_nxt* removes from the current aggregation object *o_cur* the contribution of an element *e_exp* that has expired from a window. It returns the updated *o_nxt*.

A user defined aggregate function is registered with the system function: aggregate_function(Charstring agg_name, Charstring initfn, Charstring addfn, Charstring removefn) -> Object

For example, the following shows how to define the aggregate function *mysum()* over windows of numbers:

```
create function initsum() -> Number s as 0;
create function addsum(Number s_cur, Number e)
         -> Number s_nxt as res + e;
create function removesum(Number s_cur, Number e_exp)
         -> Number s_nxt as s_cur – e_exp;
```

These functions are registered to the system as the aggregate function *mysum()* by the function call:

```
aggregate_function('mysum','initsum','addsum','removesum');
```

After the registration *mysum()* can be used in CQs as:

```
select mysum(w)
from Window w
where w in fewtWindowize(s, 10, 2, 1);
```

In this simple example the aggregation object is a single number. It can also be arbitrary objects, including dictionaries (temporary tables) holding sets of

rows, which is used in the Challenge implementation to incrementally maintain complex spatio-temporal aggregations.

2.3 Solution outline

In Figure 1 the *Event Reader* node reads the full-game CSV file and produces the *Game* stream consisting of events for both balls and players. The *Event Reader* then scales the time stamps by subtracting the start time. It also transforms the position, velocity, and acceleration values to metric scales. To avoid the *Event Reader* becoming a bottleneck it is implemented as a foreign function in C. To speed up the communication we use binary representation of all events communicated between query processing nodes, while the input and output log files use the CSV format.

The *Interrupt Reader* node produces the *Interrupt* stream, which contains referee interruptions, by reading and transforming the provided game interruptions files.

The *DEBS Splitter* node merges the two input streams based on the time stamps in the streams and produces parallel input streams for the different queries. It also filters out those event stream tuples of the *Game* stream that are in-between game interruptions. The nodes *Q1 Front End*, *Q2/Q4 Ball Hitter*, and *Q3 Front End* receive parallel data streams required for the four Grand Challenge queries Q1-Q4. Q2 and Q4 share some downstream computations executed by *Q2/Q4 Ball Hitter node*.

In EPIC the *splitstream()* system function provides customizable distribution and transformation of stream tuples. The user can provide customizable splitting logic as a *distribution query* over an incoming tuple that specifies how a tuple is to be distributed, filtered and transformed.

The distribution query for the *DEBS Splitter* in Listing 1 is passed as an argument to *splitstream()*.

```
1 select i, ev from Integer i
2 where (i = 0 and isPlayer(ev)) or
3       (i = 1) or
4       (i = 2 and isPlayer(ev));
```

Listing 1. DEBS Splitter distribution query

The result of the query are pairs *(i, ev)* specifying that an incoming event *ev* is to be sent to output stream number *i*. In the DEBS splitter distribution query three output streams enumerated by *i* are specified. They produce the corresponding streams *Q1 Input*, *Q2/Q4 Input*, and *Q3 Input*. The Boolean function *isPlayer(v)* returns true if *v* is a player sensor reading.

To speed up the processing, shared computations are made in separate nodes. In Figure 1 the *Q1 Front End* and the *Q3 Front End* nodes perform

stream preprocessing and reduction for queries 1 and 3, respectively, while the *Q2/Q4 Ball Hitter* node detects hits to the ball needed by queries 2 and 4.

2.3.1 Query Q1: Running Analysis

Figure 2 shows the topology of Q1. The aggregated running statistics for different time windows are computed in parallel based on the common current running statistics produced by the *Q1 Front End* node. The stream containing player sensor readings is sent to the *Q1 Front End* node (see Listing 1), which produces the running statistics. The running statistics is then broadcasted to four other nodes to compute the aggregated running statistics of different time window lengths.

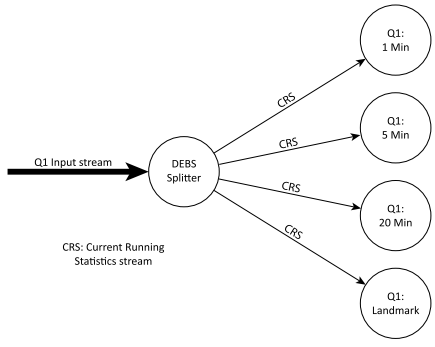


Figure 2. Query 1 data stream flow

2.3.1.1 Incremental maintenance of running statistics

In order to make the result more reliable for the current running statistics, we first create a *1 s* tumbling window and then calculate the statistics for each player over that window. The window length *1 s* was chosen experimentally to produce stable results. Both running and aggregate statistics utilize user defined aggregate functions to maintain arrays of the two types of statistics for each player.

2.3.1.2 Current running statistics

For each incoming player sensor reading in the current *1 s* window, the following statistics tuple for each player is incrementally maintained in an array:

(ts_start, ts_stop, pid, left_x_start, left_y_start, left_x_stop, left_y_stop, right_x_start, right_y_start, right_y_stop, right_y_stop, sum_speed, count)

The time stamp *ts_start* stores the first time when a sensor reading of player *pid* arrives to the current window, while *ts_stop* stores the last sensor reading. The elements *left_x_start*, *left_y_start*, *right_x_start,* and *right_y_start* are the position readings of the left and right foot of the player at time

ts_start, while *left_x_stop*, *left_y_stop, right_x_stop,* and *right_y_stop* are the corresponding foot position readings at time *ts_stop*. To incrementally calculate the average velocity the elements *sum_speed* and *count* are also included. *ts_start*, *left_x_start*, *left_y_start*, *right_x_start,* and *right_y_start* are updated only when the first sensor reading of the player *pid* arrives to the window, while all the other elements are updated every time a sensor reading of *pid* arrives. Here, no remove function is needed for the aggregation, since we are maintaining a stream of tumbling windows where the statistic will be re-initialized every time the window tumbles.

With the statistics above, the current running statistics for a given player is calculated as the Euclidian distance between the average position of the first and last update during the time window.

2.3.1.3 Aggregate running statistics

We have chosen to log the result tuple of Q1 in CSV format every *1 s* since the current running statistics are not emitted more often than once per second. Four FEW time windows were defined for aggregating running statistics with lengths 1 minute, 5 minutes, 20 minutes, and the entire game. All windows slide and emit results every *1 s*. FEW is critical for early emission while the first windows are being formed.

Aggregate running statistics over the window are incrementally maintained in an array similar to current running statistics.

The stream from the *Q1 Front End* node contains the elements *ts_start, ts_stop, player_id, intensity, distance,* and *speed*. The difference *ts_stop – ts_start* is used to incrementally maintain the duration of a player being in the corresponding running *intensity* class. Analogously, the moving distance is maintained for the corresponding intensity classes by incrementally associating the incoming distance with the right intensity.

2.3.2 Query Q2: Ball Possession

Figure 3 shows the data flow of queries Q2 and Q4 combined. The *Q2/Q4 input* stream consists of player, ball, and interrupt sensor readings. The *Q2/Q4 Ball Hitter* computes the *Ball Hitter* and the *Ball* streams. The *Ball Hitter* stream contains ball hitter events, which occur when a player *pid* at timestamp *ts* hits the ball. The *Ball* stream contains Ball Hitter events interleaved with ball sensor readings. The *Q2/Q4 Ball Hitter* node emits the *Ball* stream to the *Shot on Goals* query processing node, which executes the final stages of query Q4. The *Ball Hitter* stream contains only ball hitter events and is sent to the *Player Possession* node, which calculates and broadcasts the same *Player Ball Possession* stream to four *Team Possession* query processing nodes. The *Team Possession* nodes log every *10 s* statistics of team ball possessions for the two teams with the different window lengths: 1 minute, 5 minutes, 20 minutes, and a landmark window of the entire game.

As an alternative, we also measured reporting team possessions every *1 s* resulting in the same latency and throughput.

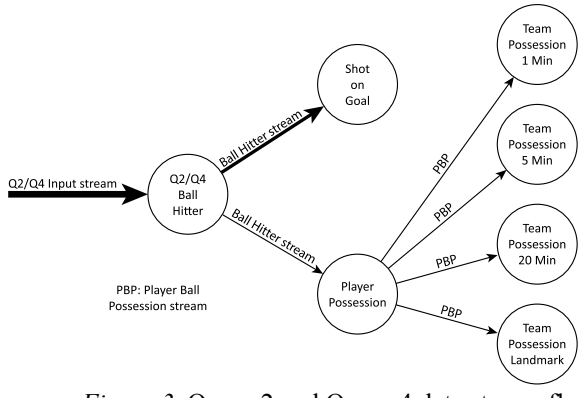


Figure 3. Query 2 and Query 4 data stream flow

2.3.2.1 The Q2/Q4 Ball Hitter query processing node

In order to compute a stream of ball hitters, we maintain acceleration of the ball *ballacc*, its position *bx*, *by*, *bz*, the shortest distance from a player to the ball *sdist*, and the player *pid*.

For every input ball sensor reading, the *Q2/Q4 Ball Hitter* node incrementally updates the ball acceleration and the ball position accordingly. When a player sensor reading arrives, it incrementally maintains *sdist*.

A ball hitter event is emitted when both the following criteria hold:

- C1: The ball acceleration reaches a predefined threshold: $ballacc > 55\ m / s^2$.
- C2: The shortest distance *sdist* is within the player's proximity: $sdist < 1\ m$.

There are 36*200 player sensor readings per second. In addition, after being hit, the ball acceleration remains high for a while, in particular before the ball leaves the player's proximity. Therefore, the two conditions C1 and C2 will hold for a short period of time within which several ball hitter events could be reported for the same actual ball hit by the player. To avoid generating false *ball hitter* events, we employ a *dropping policy* to drop player sensor readings occurring significantly later than the last report time. The dropping policy is expressed by the following query condition over a player sensor reading *v*: *ts(v)-lrts > epsilon;*

Here, *lrts* is the latest timestamp when a ball hitter event was reported, and *epsilon* is the minimum time period between two reports. Because Q4 is more sensitive to the *ball hitter* events, we have empirically tuned this parameter to *0.2 s* to get the best possible accuracy of Q4.

2.3.2.2 The Player Possession query processing node

The *Player Possession* node emits the *Player Ball Possession (PBP)* stream consisting of the variables *fts*, *pid*, and *hits*, which state that the player *pid* possessed the ball *hits* times, starting from first time the player hits the ball, *fts*.

The *Player Possession* node increases the variable *hits* if a ball hitter event *bhe* is from the same player *pid*. Otherwise, it will emit ball possession events for player *pid* and then reset the variables. The total possession time is the interval between the timestamps *bhe* and *fts*.

2.3.2.3 The Team Possession query processing nodes

There are *four Team Possession* nodes, each with different window length: 1 minute, 5 minutes, 20 minutes, and a landmark of the whole game. For the received *Player Ball Possession* stream they compute team possession statistics as follows:

Incrementally calculate the sum of the ball possessions of all players in each team when a corresponding player ball possession arrives.

When a report is logged, the following two percentages are calculated:

$$P_A = \frac{sumTeamA}{sumTeam\ A + sum\ TeamB}$$

$$P_B = \frac{sumTeamB}{sumTeam\ A + sum\ TeamB}$$

Here FEW windows are used to frequently report while the first windows are being formed. For example, the results must be regularly delivered every 10 s while the team possession landmark window is being formed.

2.3.3 Query 4: Shot on Goal

The Shot on Goal node receives three different kinds of events in the Ball stream:

- A ball hitter event marks a shot and contains a time stamp and the *pid* of the shooting player.
- A ball event contains the current ball sensor reading.
- An interrupt event indicates a game interruption. It is good practice to reset the shot detection when an interruption occurs.

Q4 shares detection of a ball hit with Q2. However, the logic for detecting a shot is slightly different for the two queries: Q2 is specified stricter than needed for Q4. To share computations this stricter logic is also used for Q4.

The operation of Q4 is straightforward; it is iteration over the *Ball* stream to keep track of the state of a shot:

1. Wait for the next ball hitter event.
2. Check ball events until the ball has travelled one meter.

3. Return ball events as long as the ball is approaching the opposite team's goal.

The calculation of the ball direction uses basic linear algebra over the ball sensor readings.

Gravity is accounted for to an extent. The expected time for the ball to travel to the goal line is multiplied twice with the acceleration constant g, and added to the height of the goal bar. The actual ball trajectory is not considered, but the current calculation should be an adequate approximation.

Using the Q2 requirements for detecting a ball hit has the drawback that some events are not detected, such as the header at 12:19 in the second half our example Game stream, since the ball is more than one meter away from any sensor. Whether that is technically a "shot" is questionable.

Curve balls need special attention. For example, at 26:07 in the first half there is a curve ball goal. In this case the direction of the ball is pointing outside the goal posts, while the ball later curves inwards and comes to rest inside the goal.

To handle curve balls we have introduced a state pending, indicating that a shot is not yet dismissed, but could later become a shot on goal. The model adds two meters of margin on both sides of the goal posts and the shot is considered pending if it points in the direction of the margin area.

Bounces are considered as long as the direction of the bounce is within the negative distance of the goal bar plus gravity. While the instructions do not account for bounces at all, this limit should add some correctness to the algebra.

Shots that are bounces, which we detect, are not included in the provided list of shots on goal. In the second half of the game there are four shots on goal that are bounces. They are at 4:11, 19:39, 24:36 and 29:29. Setting the bounce threshold to zero, i.e. not considering bounces creates a result in accordance to the specification. Viewing the video makes it apparent that the specification is not correct in this regard.

2.3.4 Query 3: Heat Map

In Query 3 a grid on the field is formed where the cells are numbered in row order, for example from 0 to 6399 in a 64 X 100 grid. Given the position of a player *(x,y)*, the function *cell_id(x,y,grid_size)* returns the corresponding cell number for a given grid size. Query results for lower resolution grids are computed by aggregating the results for the higher resolution grids. Thus we incrementally maintain the results only for the highest resolution.

Note that the results of longer windows cannot be built on top of the results from a shorter window. This is due to the *1 s* stride parameter in all the queries. For example, the 5 minute window can't be built on top of the results produced by the 1 minute window, since the 5 minute window needs to remove the contributions made to the statistics by the expired elements, i.e.

the elements with the time stamp *ts – 300 s*, where *ts* is the current time stamp. Those elements are too old to be in the 1 minute window. Nevertheless, the definition of longer windows in terms of shorter ones could have been utilized if the stride was one minute instead of the one second stride in the Challenge specification.

2.3.4.1 Q3 Front End

Figure 4 shows the dataflow diagram for query Q3. The *Q3 Front End* node produces the *One Second HeatMap (OSHM)* stream by forming *1 s* tumbling windows over the incoming tuples. Thereby incremental user defined aggregate functions are used to maintain statistics per second in a table *heamap1s(pid, cell_id, ts, cnt)* local per window. Here *ts* is the latest time stamp player *pid* has been present in the cell identified by *cell_id* cell identifier in the highest resolution grid (64 X 100). *cnt* is the total number of sensor readings for player *pid* in the cell in the current window.

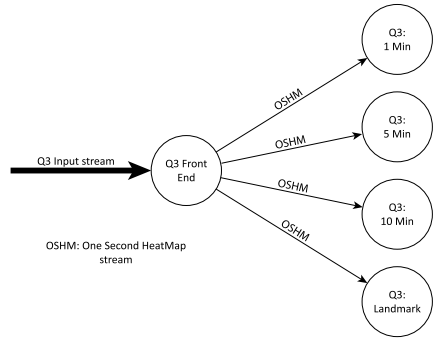


Figure 4. Query 3 data stream flow

The *OSHM* stream is produced by emitting all the rows accumulated in the table during the past second.

The *Q3 Front End* significantly reduces the stream volume by summarizing it. It receives 200 tuples per second from 36 sensors, in total 7200 tuples/second. It emits at maximum the total number of cells all the players have been present in the highest grid resolution during one second, which is about 70 tuples per second, i.e. a factor 10 reductions in stream flow.

2.3.4.2 Q3 query nodes

The *OSHM* stream is broadcasted to four Q3 query nodes *Q3 1 Min*, *Q3 5 Min*, *Q3 10 Min*, and *Q3 Landmark*. These nodes run parallel CQs over time windows with lengths 1, 5, 10 minutes, and whole game, respectively. The windows are formed by the FEW window specification *fewtWindowize(oshm, length, 1, 1),* where *length* is 60s, 300s, 600s and the whole game duration, respectively. The stride and the emit frequency are both *1 s*.

The emit frequency is needed so that sub-windows are emitted while the window is being formed the first time.

Similar to *Q3 Front End*, the Q3 query nodes incrementally maintain user defined aggregates by updating the following local tables inside each window as the input stream elements arrive:

```
heatmap(pid, cell_id, ts, cnt)
sensor_count(pid, total_cnt)
```

In table *heatmap*, the attribute *cell_id* is the cell player *pid* has been present in, *ts* is the latest time player *pid* was in the cell, *cnt* is the number of times the player has been present in the cell. To enable translation of *cnt* into percentages per cell, the Q3 query nodes also maintain *total_cnt* per player, which stores the total number of position reports in all cells for a given player during the window in question.

Since Q3 query nodes only maintain the statistics for the highest resolution in a given window length, at reporting time they compute lower resolutions by aggregating grid cells per player to fill the bigger cells in the higher resolutions.

The Q3 query nodes log the output CSV streams to files. Since each Q3 query nodes cover all grid settings in a given window size, the produced log files contains output stream elements for more than one grid setting. We use the following grid identifiers to tag streams per grid: *6400* for 64 X 100, *1600* for 32 X 50, *400* for 16*25, and *104* for 8 X 13 grid setting.

The size of these log files is huge (ca 400,000 rows/s) since they cover all movements between grid cells over several very long windows. Here it becomes important to use SSD as storage medium, which is fast at writing big blocks in parallel, while disk arm movements for writing different log files has been observed to slow down the entire system throughput with a factor of around two.

3 PERFORMANCE

The performance of our implementation is measured based on both throughput and delay. The throughput was measured as the total execution time per query and for all queries in parallel over the entire game. The latency was measured by propagating the system wall clock of the entry time of the latest event contributing to each result tuple. The delay was calculated by subtracting the propagated entry time from the wall time when a result tuple is delivered. The throughput is measured per query while the latency is measured per output stream.

The experiments are run on a VMware virtual machine with Windows Server 2008 R2 x64, running on a laptop with the following specifications: Dell Latitude E6530, CPU: Intel Core i7-3720QM @2.60 GHz, RAM: 8 GB, Hard Disk Device: ST500LX003-1AC15G, OS: Windows 7 64-bit.

Figure 5 illustrates the throughput of the individual queries as well as all queries running together. Queries Q1, Q2, and Q4 take around 5 minutes to finish separately, while Q3 takes considerably longer time, which is mainly due to intensive report computations in the Q3 query nodes. To investigate the log writing time, *Q3* and the *all queries* columns have a watermark indicating how much time it takes to execute them without logging to disk, showing that this takes around 35 % of the Q3 alone time and 25 % of all queries together. We also investigated whether it would be favorable to parallelize the logging of the result stream for Q3 query nodes, but that turned out to be slower in our current environment.

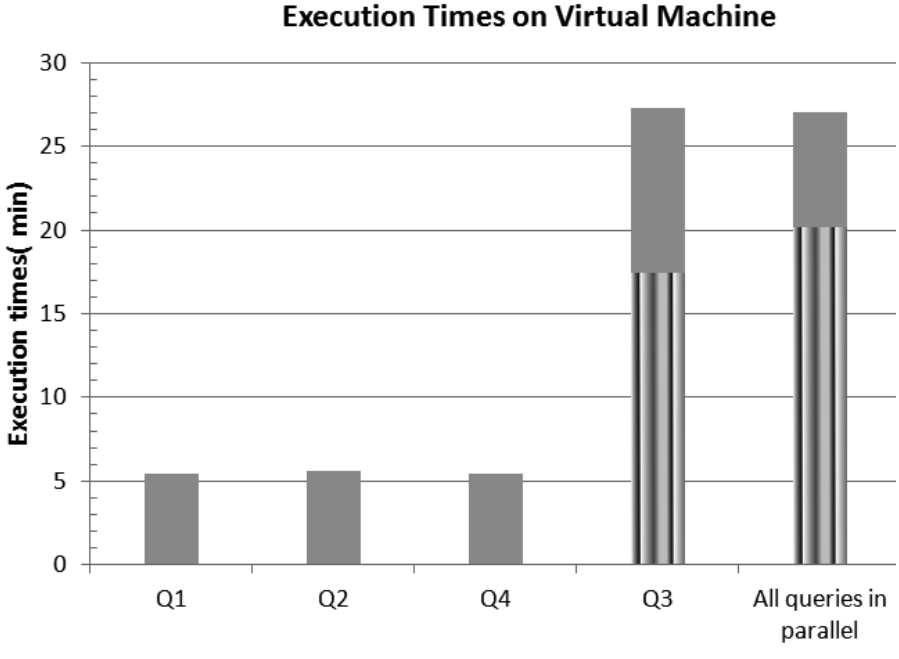


Figure 5. Performance

Since all queries run in parallel according to the dataflow diagrams, running all of them together takes approximately the same time as running the slowest one, Q3.

Figure 6 shows the average delay per output stream while running all queries together. Notice that Q2 and Q4 are time critical queries since they immediately report real-time phenomena. By contrast Q1 and Q3 report delayed statistics aggregated over time.

The VMware virtual machine containing our implementation of the Grand Challenge can be downloaded from http://udbl2.it.uu.se/DEBS/. There is also a zip archive that can be run on any Windows machine.

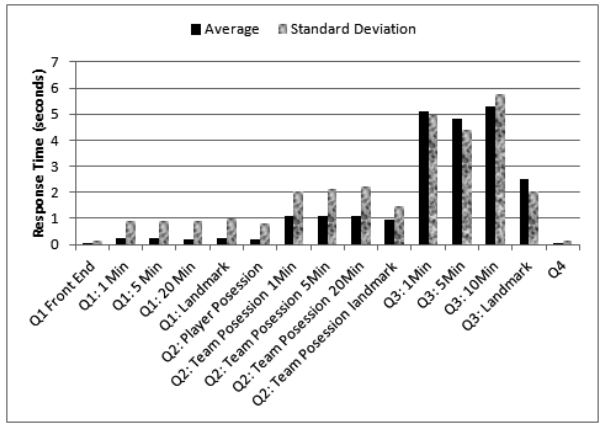


Figure 6. Delays

4 RELATED WORK

In the stream processing community, there has been a lot of work for developing query languages over data streams [5]. [7] introduced a formal specification of different kinds of windows over data streams and provided a taxonomy of window variants. The notation of report (emit) frequency was proposed in SECRET [2] without any actual implementation. SECRET is a descriptive model to help users understand the result of window-based queries from different stream processing engines. Esper [4] also allows a report frequency but does not have user defined window aggregate functions. Furthermore Esper's sliding window model is different from FEW because the slides are triggered by window content changes rather than explicitly specified time periods.

To efficiently calculate the aggregate result over long windows with small strides, [6] and [1] use delta computations to reduce the latency and the memory usage. The focus of [8] is to extend a DSMS with online data mining facilities by user defined aggregate functions over windows. The implementation described in this paper shows that EPIC is general enough to define very complicated user defined aggregations as functions while in [1] and [8] the aggregates are defined as updates.

5 CONCLUSIONS

We have addressed the Grand Challenge by expressing continuous queries in a high level language that supports incremental evaluation of aggregate functions over windows and frequently emitting windowing. We meet the real-time requirements of the real-time queries on a virtual machine running on a laptop. The extensibility of the query engine was used for supporting high throughput and low latency of time critical operations.

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

REFERENCES

1. Bai, Y., Thakkar, H., Wang, H., Luo, C., and Zaniolo, C.: A Data Stream Language and System Designed for Power and Extensibility. *Proc. CIKM Conf.*, 2006.
2. Botan, I., Derakhshan, R., Dindar, N., Haas, L., Miller, R. J. and Tatbul, N. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. *Proc. VLDB Conf.*, 2010.
3. Botan, I., Fischer, P. M., Florescu, D., Kossmann, D., Kraska, T., and Tamosevicius, R. Extending XQuery with Window Functions. *Proc. VLDB Conf.,* 2007.
4. http://esper.codehaus.org/
5. Law, Y-N, Wang, H., and Zaniolo, C.: Relational Languages and Data Models for Continuous Queries on Sequences and Data Streams. ACM TODS 36, 2, (May 2011).
6. Li, J., Maier, D., Tufte, K., Papadimos,V., and Tucker, P. A. Semantics and evaluation techniques for window aggregates in data streams. *Proc. SIGMOD Conf.*, pp. 311 - 322, 2005.
7. Patroumpas, K. and Sellis, T. Window specification over data streams. *Proc. EDBT Conf.,* 2006.
8. Thakkar, H., Mozafari, B. and Zaniolo, C.: Designing an Inductive Data Stream Management System: the Stream Mill Experience. *Proc. 2nd International Workshop on Scalable Stream Processing Systems*, 2008.
9. Zeitler, E. and Risch, T.: Massive scale-out of expensive continuous queries, *Proc. of the VLDB Endowment*, ISSN 2150-8097, Vol. 4, No. 11, pp.1181-1188, 2011.

Paper III

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 given 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 very 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.

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 **Fel! Hittar inte referenskälla.**. 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 price 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.

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

Table 1. Characteristics of different data processing workloads

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 $A\ \{A_1 ... A_n\}$. We define $Q\ \{Q_1\ ...\ Q_n\}$ 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 con-

ventional 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.

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

Listing 1. simple aggregate queries over data streams

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.

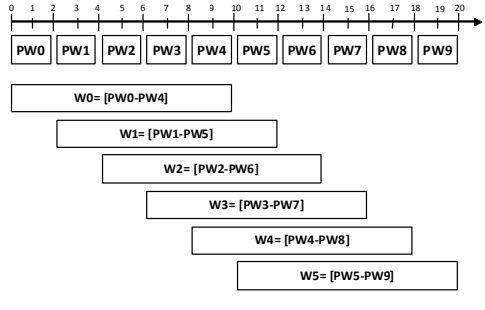


Figure 1. Sliding window maintenance

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.

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 data 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.

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

Listing 2. simple Gigascope ACQ

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.

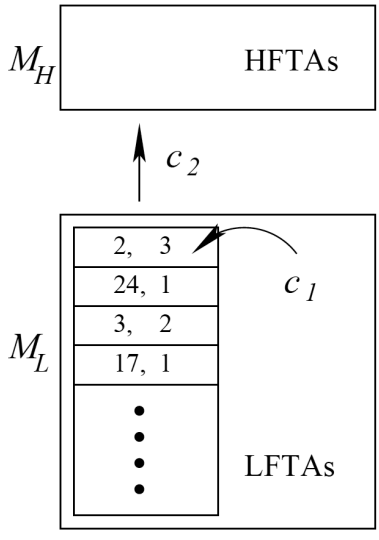


Figure 2. Single aggregation in Gigascope

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}.

```
/*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
```

Listing 3. A set of ACQs with varying grouping attributes

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

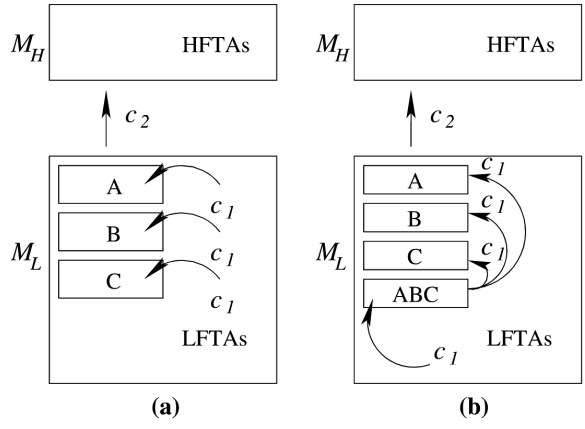


Figure 3. Processing multiple aggregates in Gigascope

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].

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.

```
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. A CCQ

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 ex-

plicit 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).

```
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;
```

Listing 5. Incremental user defined clustering function

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.

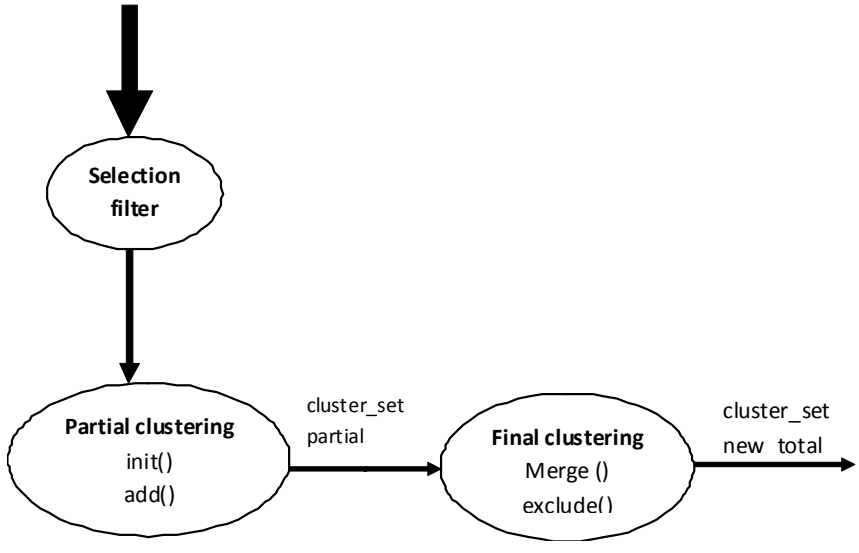


Figure 4. Data flow of a single CCQ

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.

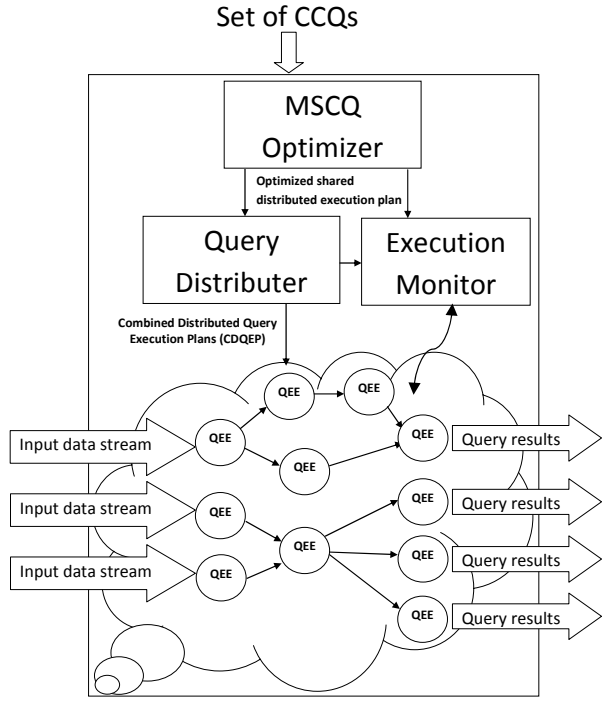


Figure 5. MSCQ system

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 new CDQEP every time a new 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 extensible, distributable multi-query optimization, specifically exploiting all sharing oppor-

tunities 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 stream 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 Information 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.

Paper IV

Framework for real-time clustering over sliding windows

Sobhan Badiozamany
Kjell Orsborn
Tore Risch
Department of Information Technology, Uppsala University, Sweden
Emails: Firstname.Lastname@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.

CCS Concepts
• **Information systems~Data stream mining**

Keywords
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.

```
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)
```

Listing 1. An example Continuous Clustering Query

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 $W_{2,12}$.

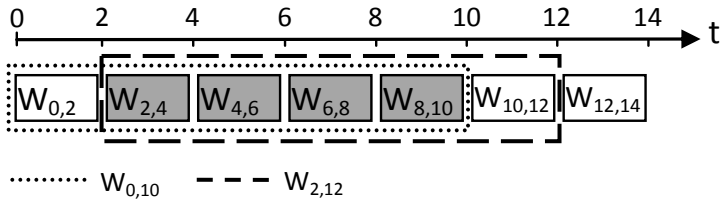


Figure 1. Data overlap in sliding windows

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:

$$COUNT(A \cup B) = COUNT(A) + COUNT(B) \text{ } \textit{(Incremental)}$$

$$COUNT(A \backslash C) = COUNT(A) - COUNT(C) \text{ } \textit{(Decremental)}$$

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] [9]. 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 [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 summariza-

tion 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 [13] 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] [14] 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.

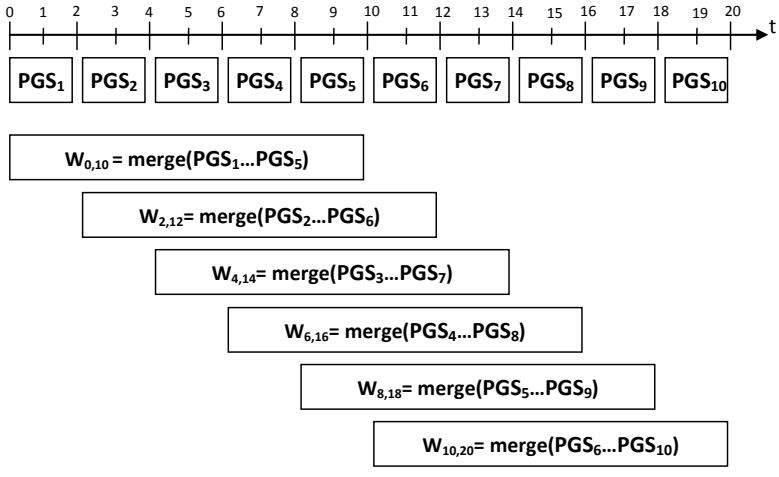


Figure 2. Final summarization with Repetitive Merge

2-phase decomposition of clustering algorithms

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 $DS=\cup_{1\leq i \leq n} DS_i$. 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:

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

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:

$$C(CS2 \setminus CS1) = EX(CS2, CS1)$$

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.

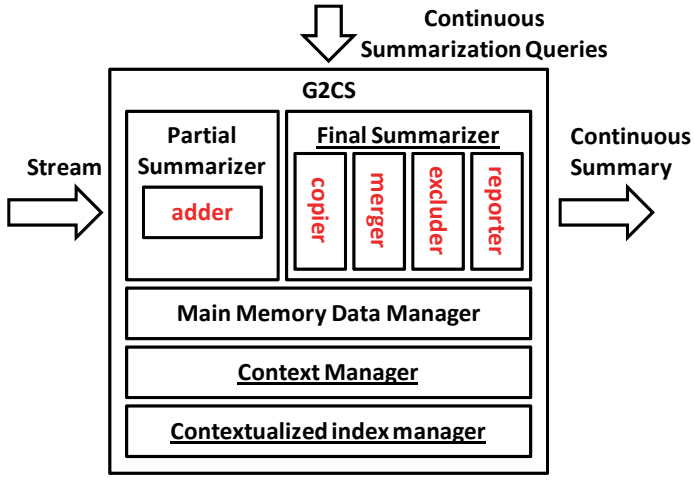


Figure 3. Generic 2-Phase Continuous Summarization

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 manage-

ment 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 [15] 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 in-

stances. In the G2CS framework information about clusters are stored in a *contextualized clustering table, CCT* with the schema:

CCT(cid, cxtid, a1,….an)

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-BIRCH(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.

```
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)
```

Listing 2. G2CS decomposition of clustering algorithms

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*.

```
1   CBIRCH-merger(Integer c_incoming, Integer c_res, Number r)
2   {
3     for each ROW mc in (
4     select cid, c_incoming, cm, ls, ss, c
5     from CCT-BIRCH where cxtid = c_incoming)
6     {
7       cf-near= nearest(c_res, mc.cm)
8       if can-absorb(cf-near, mc.cid, r)
9         update table CCT-BIRCH set
10          cm = cm + mc.cm,
11          ls = ls + mc.ls,
12          ss = ss + mc.ss,
13          c  =  c + mc.c
14        where cid = cf-near.cid;
15      else
16      insert into CCT-BIRCH values
17        (new_cid(), c_res, mc.cm,
18        mc.ls, mc.ss, mc.c);
19    }
20  }
```

Listing 3. C-BIRCH merger plug-in

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.

BIRCH is generally sensitive to the order of the data points, but as explained by the authors of BIRCH [15], 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 $W_{b,e}$.

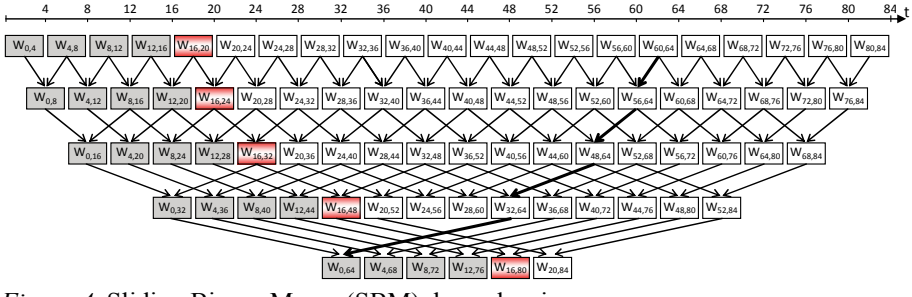


Figure 4. Sliding Binary Merge (SBM) dependencies

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·S*, while their strides are still *S*.

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

$TCRM = PR \cdot (CCRM + MC)$

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:

$TCRM = PR \cdot (AC + MC)$

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.

Memory footprint

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 $O(AC \cdot PR)$, 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*-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*:

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

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

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

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 [16] plugged-in to our main-memory database engine using MEXIMA [17]. 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.

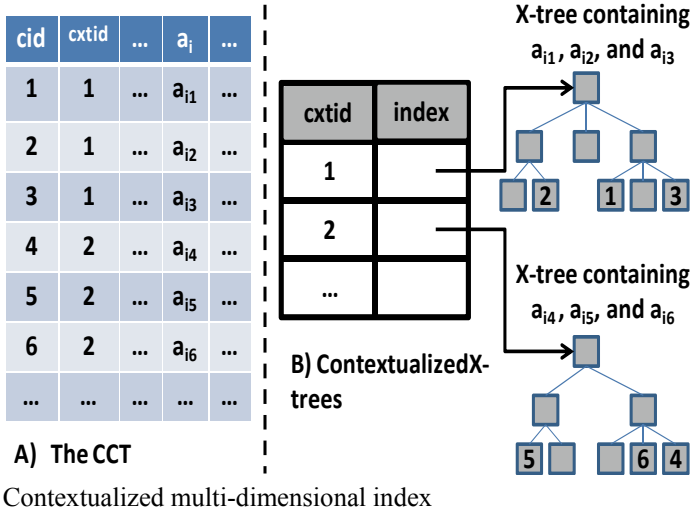


Figure 5. Contextualized multi-dimensional index

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 [17] 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.

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

Table 1. Sliding window maintenance alternatives

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.

The data streams

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 that 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 differential maintenance 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).

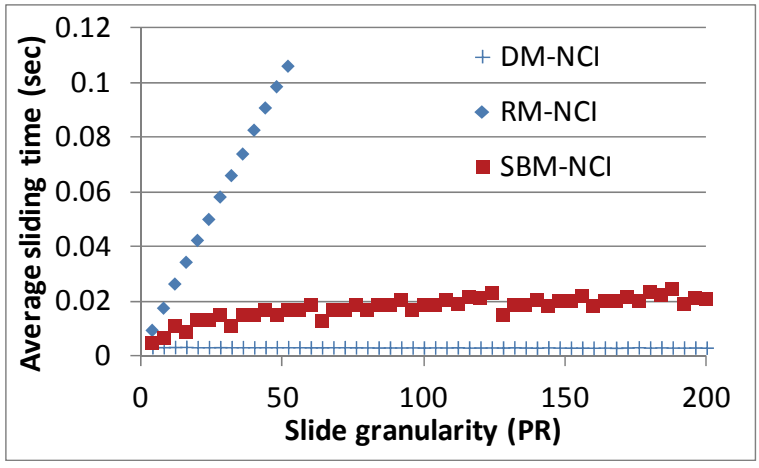


Figure 6. conventional GROUPBY over SDG stream.

4.2 Non-indexed C-BIRCH

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.

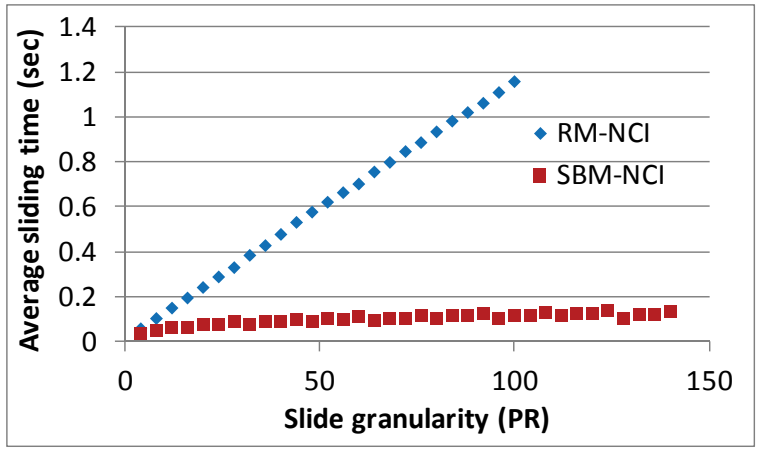
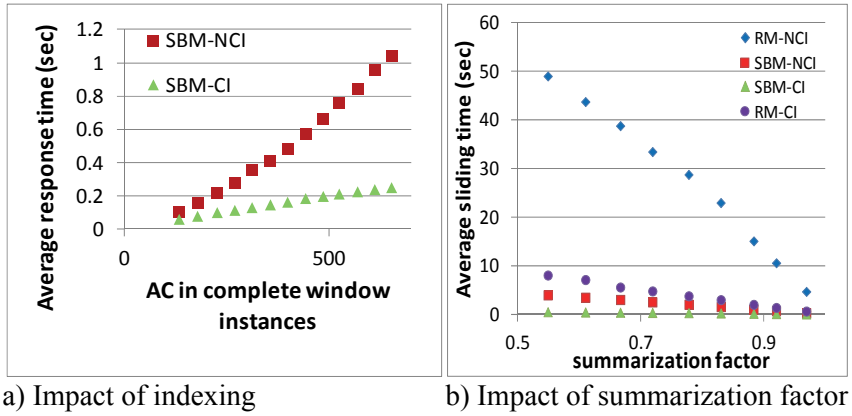


Figure 7. C-BIRCH over SDC stream.

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 the average sliding time of the 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.

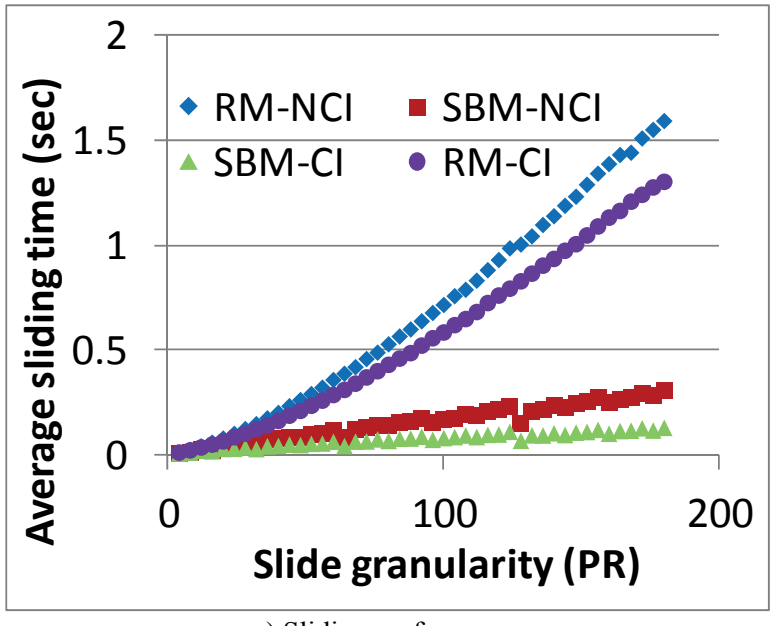
4.5 Contextualized C-BIRCH for Real Data

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

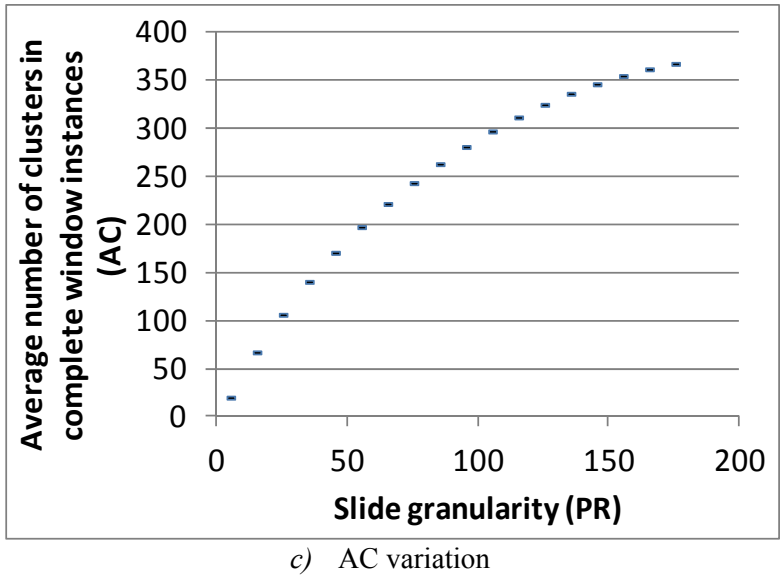
```
SELECT   Center(cid), Radius(cid), COUNT(cid)
FROM     DEBS (Range = R, Slide = 1sec)
CLUSTER BY X, Y, Z AS cid
USING    C-BIRCH(Radius = 2.0 meters)
```

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



c) 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.

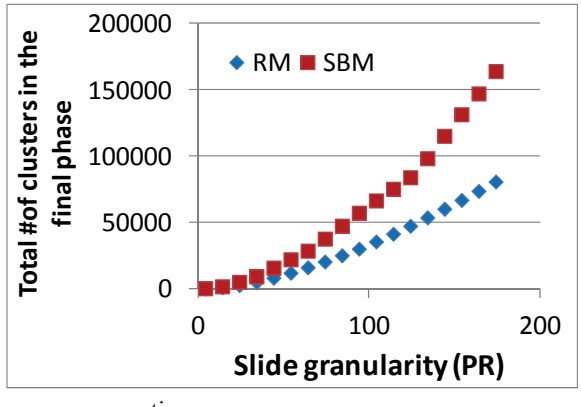


Figure 10. Memory consumption

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%.

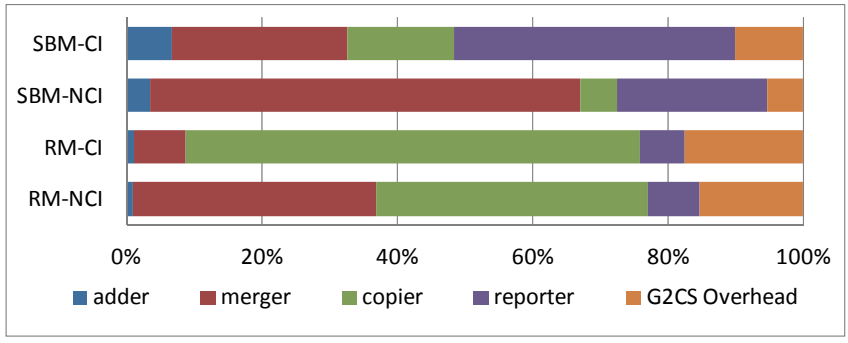


Figure 11. Workload break down for C-BIRCH variants

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 [18]. The clustering quality measure used in the experiment is the *Weighted Average Radius* (WAR) [15] 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.

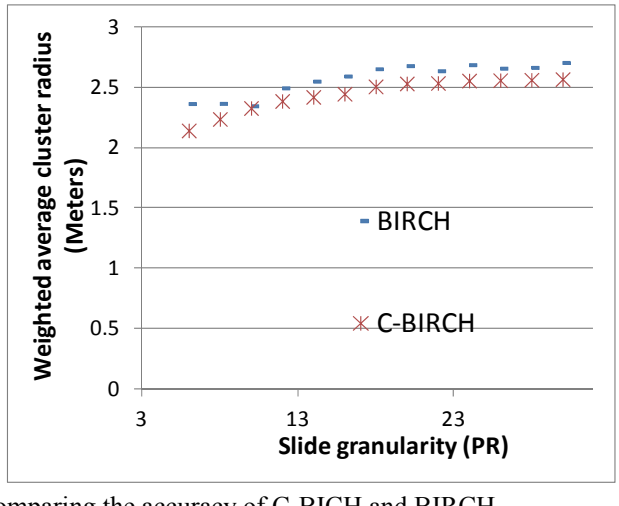


Figure 12. Comparing the accuracy of C-BICH and BIRCH

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 [15]). 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 [19] [9] 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. [9] 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 utilizes Hadoop map-reduce tasks 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 [15] [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 sum-

mary 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 [15] 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. [14] 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] 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.

[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] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[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.