QUALITY-AWARE MINING OF DATA STREAMS

(Research-in-Progress)

Conny Franke **Michael Hartung** **Marcel Karnstedt** **Kai-Uwe Sattler**
Department of Computer Science and Automation, TU Ilmenau, Germany

Abstract: Due to the inherent characteristics of data streams, appropriate mining techniques heavily rely on window-based processing and/or (approximating) data summaries. Because resources such as memory and CPU time for maintaining such summaries are usually limited, the quality of the mining results is affected in different ways. Based on selected mining techniques, we discuss in this paper relevant quality measures for analysis results. Furthermore, we describe extensions to two specific stream mining algorithms allowing (1) to estimate resource consumptions (mainly memory space) based on user-specified quality requirements and (2) to determine the output quality for changes in the available resources.

1 INTRODUCTION

Stream mining has recently attracted attention by the database as well as the data mining community. The goal of stream mining is a fast and adaptive analysis of data streams, i.e., the discovery of patterns and rules in the data. Two important tasks are frequent itemset mining and clustering. Frequent itemset mining aims to identify combinations of items that occur frequently in a given sequence of transactions. Clustering means to partition data objects so that similar objects (wrt. a certain distance function) are in the same partition whereas different objects are associated with different partitions [6]. Typical applications of mining data streams are among others click stream analysis, analysis of records in networking and telephone services or analysis of sensor data.

The main challenge in applying mining techniques to data streams is that a stream is theoretically infinite and therefore in most cases cannot be materialized. That means that the data have to be processed in a single pass using little memory. Based on this restriction and the goals of data mining one can identify two divergent objectives: On the one hand the analysis should produce comprehensive and exact results and detect changes in the data as soon as possible. On the other hand the single pass demand and the problem of resource limitations allow only to perform the analysis on an approximation of the stream (e.g., samples or sketches) or a window (i.e., a finite subset of the stream).

However, using an approximation or a subset of the stream affects the quality of the analysis result: The mining model (in case of clustering the cluster representation such as center and diameter as well as the sets of assigned objects; in case of frequent itemset mining the set of frequent itemsets) differs from the mining model we would get if the "whole" stream or a larger subset is considered. This is particularly important because some of the proposed stream mining approaches support time sensitiveness (reducing the influence of outdated stream elements) by using weaker approximations for outdated elements. Thus, the mining quality for these elements is worse than for newer data. In contrast to examining the quality of the input data, in this work we investigate the quality of the analysis result influenced by the applied stream mining techniques. The problem of the quality of the mining model can be also considered in the opposite direction: Based on user-specified quality requirements one could derive resource requirements, i.e., the memory needed for managing stream approximations in order to guarantee the requested quality.

Based on this observation we argue in this paper that quality is an important issue in many applications of stream mining. We further propose two quality-aware mining approaches for clustering and frequent itemset mining showing that quality-awareness is basically orthogonal to the specific mining problem, even if the individual mining approach requires dedicated techniques for considering mining quality. The contribution of this paper is twofold:

- We discuss the problem of quality-awareness in stream mining.
- We present two specific mining approaches incorporating this quality-awareness.

The remainder of this paper is structured as follows. After a brief survey of relevant mining approaches for data streams in Section 2 we discuss relevant quality measures in Section 3. In Section 4 we present our extended approaches by adding quality measuring as well as a resource adaption based on user-specified quality requirements. Results of an experimental evaluation are reported in Section 5. Finally, we conclude the paper and discuss open issues for future work.

2 MINING IN DATA STREAMS

As stated in Sect. 1 the goal of clustering is to partition a set of data objects that similar objects are assigned to the same partition whereas different objects are in different partitions. For this purpose, two main classes of approaches can be distinguished. Hierarchical approaches build a hierarchical representation of all objects from which the cluster structure can be derived. In contrast, partition-based approaches create and "measure" partitions of data. Probably the most well-known approach is k-Means and its derivatives. The idea is to minimize the distance of the points P to the centers of the clusters $C_1...C_k$ for a given k, i.e., $\sum_{i=1}^{k}\sum_{P\in C_i} dist(P,C_i)$. This is achieved by an iterative refinement: starting from randomly chosen cluster centers, each point as assigned to its nearest center. Based on this, new centers are computed (representing the "mean point" of all assigned points) and this is repeated until a given stop criterion is reached. Due to these multiple passes the basic k-Means is not suitable for stream processing.

As a first solution (though originally not intended for data streams) the BIRCH approach [11] was proposed, which is based on a tree structure. The nodes of this tree represent a set of (sub-)clusters by maintaining so-called clustering features (e.g., the number of points associated with this cluster, the single and the quadratic sum of the point values). Furthermore, by exploiting additive properties sub-clusters (nodes) can be merged. In summary, this tree represents a summary of the dataset and is used for the actual (offline) clustering (by clustering the leaf nodes).

The CLUStream [1] approach extends BIRCH by adding time-awareness. CLUStream uses so-called micro-clusters, which are basically a combination of clustering features from BIRCH and additional time information. These micro-clusters are stored at fixed time intervals as part of snapshots. Snapshots are organized in a pyramidal time frame where more current snapshots are stored in shorter time intervals than older ones. Note that the original approach assumes that all available memory can be used for storing the snapshot pyramid. In Sect. 4.2.2 we will discuss how the amount of required memory can be adjusted and how this affects the result quality.

Frequent itemset mining deals with the problem of identifying sets of items occurring together in so-called transactions frequently. Basically, two classes of algorithms can be distinguished: approaches with candidate generation (e.g., the famous apriori algorithm) as well as without candidate generation. Here, only the latter ones are suitable for stream mining. Usually, these approaches are based on a prefix-tree-like structure. In this tree – the frequent pattern (FP) tree – each path represents an itemset in which multiple transactions are merged into one path or at least share a common prefix if they share an identical frequent itemset [7]. For this purpose, items are sorted as part of their transaction in their frequency descending order and are inserted into the tree accordingly. Again, the FP tree is used as a compact data summary structure for the actual (offline) frequent pattern mining (the FP growth algorithm).

In order to mine streaming data in a time-sensitive way an extension of this approach was proposed [4]. Here, so-called tilted time window tables are added to the nodes representing window-based counts of the itemsets. The tilted windows allow to maintain summaries of frequency information of recent transactions in the data at a finer granularity than older transactions. The extended FP tree, called pattern tree, is updated in a batch-like manner: incoming transactions are accumulated until enough transactions of the stream have arrived. Then, the transactions of the batch are inserted into the tree. For mining the tree a modification of the FP growth algorithm is used taking the tilted window table into account. The original approach assumes that there is enough memory available to deliver results in the given quality and no way is described how to proceed if the algorithm runs out of memory. In Sect. 4.3 we will discuss how the amount of required memory can be adjusted and how this affects the result quality.

3 QUALITY MEASURES FOR STREAM MINING

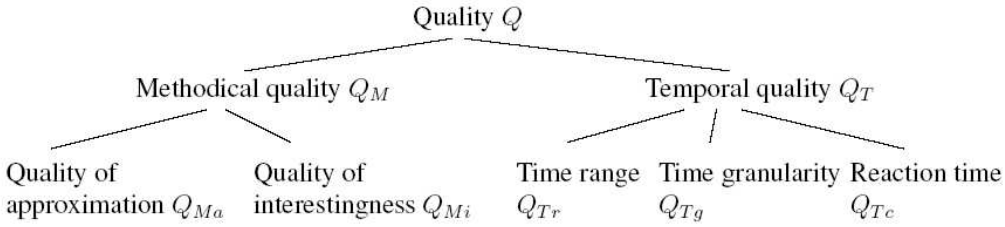


Figure 1: Different quality measures

In this section we briefly describe the different quality measures we examine in the context of stream mining. In the following sections, when describing the implemented mining algorithms, we will introduce how these measures are integrated into stream processing. We distinguish several different classes of quality measures, which are categorized in Fig. 1.

All Q_{T*} are identical for different mining problems and symbolize concrete measures, while Q_{M*} represent classes of quality measures that are always specific to the investigated problem and the applied algorithm(s). In the context of clustering these measures involve the clustering quality, e.g., sum of square distances (SSQ), diameter and other standards to evaluate the final result of a clustering. One traditional measure for the problem of frequent itemset mining is the ratio between a value ε and the support σ, which reflects the maximal deviation from the defined support each of the itemsets finally identified as frequent could possess. For both mining applications examined in this work, special interesting measures (Q_{Mi}) have been proposed in the literature (e.g., [9]). In the context of frequent itemsets the support is one such interesting measure, other relevant objective measures are not considered here.

As the algorithms proposed in this work always take time sensitiveness into account, we define time as another important quality measure. Q_{Tr} describes how far we can look back into the history of the processed stream and Q_{Tg} how exact we can do this, which means which time granularity we can provide. Q_{Tc} corresponds to one of the main challenges of stream mining: the actual time we need in order to register changes in the stream. As change detection may happen internally without reference to user queries, Q_{Tc} depends only on the update interval in both applications. As we will see, for clustering this is the interval between inserting snapshots into the pyramidal frame, for frequent itemset mining this is the size b of a batch. These temporal quality measures must not be confused with temporal factors that influence the methodical quality (see Fig. 2).

For the remainder of this paper, if we refer to all quality measures as a whole, we will use the symbol of the superclass Q and the general term ‘quality’. Some of the introduced measures depend on influencing factors, others represent an independent value. In Fig. 2 we summarize the meanings of the investigated quality measures and which factors act as parameters, if any.

Q	Output		Factors
	Clustering	Frequent itemsets	
Q_{Ma}	SSQ	ε/σ	queried time interval
Q_{Mi}	-	σ	-
Q_{Tr}	maximal "look back time"		-
Q_{Tg}	minimal granularity		queried "look back time"
Q_{Tc}	minimal time till detection of changes		update interval u

Figure 2: Values of quality measures and influencing factors

Furthermore, we appoint which measures are chosen for exemplarily representing the classes Q_{Ma} and Q_{Mi}. In the original algorithms, no conceivable measure for Q_{Ma} depends on the queried interval. In the following sections we will show that the interval's influence results from the implemented adaptivity to resource changes.

As introduced before, in the context of stream mining we have to process the stream data while adhering to limited resources available. Thus, we propose resource-awareness in conjunction with quality awareness as one of the main requirements – and challenges in parallel. We interpret resource requirements, we focus on memory requirements, as a counter piece to quality. Thus, we have to determine correlations between quality and the needed memory. Fig. 3 illustrates how these dependencies are integrated into the operational flow of stream processing. All available resources are managed adaptively by the Data Stream Management System (DSMS). The amount of memory needed to achieve a claimed quality is requested from and provided by the DSMS, as long as it is available. If quality constraints are provided at the operator side the needed resources will be allocated in this way. If not enough resources are available this is signalized by according changes in the methodical and temporal qualities achievable. This is also necessary if resource changes occur, which may happen, for instance, when new operators are processed and query graphs are modified accordingly. Two ways of putting resource and quality-awareness into practice get evident:

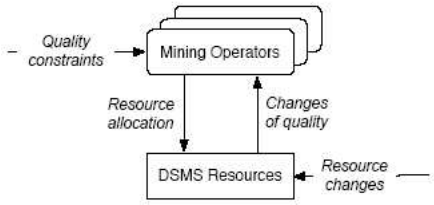


Figure 3: Mapping of quality

1. Claim for specific quality requirements and deduce the needed resources to achieve it.
2. Limit the maximal memory available for processing and deduce the achievable quality.

Thus, in principle we have to determine two (theoretical) kinds of functions:

1. $r : Q \rightarrow R$ - maps claimed quality to the resources needed, and
2. $r' : R \rightarrow Q$ - maps provided resources to the achievable quality, as an inverse function to r.

More detailed, r is one function $r(args_x, Q_x(args_x), args_y, Q_y(args_y), \ldots)$ taking all claimed quality measures $Q_x, Q_y, \ldots$ and their factors as input, but we write $r(Q)$ for short. In contrast, r' represents a bundle of inverse functions $r'_x, r'_y, \ldots$, each corresponding to one quality measure $Q_x, Q_y, \ldots$. Moreover, as we do not state the distribution of the stream elements as input factor for any function, r and r' differ with different stream characteristics.

4 Quality-Aware Mining Operators

4.1 Operators for Data Streams

We implemented all the introduced algorithms and techniques using a DSMS called PIPES [8]. Rather than a monolithic DSMS, PIPES is an infrastructure that, in conjunction with the comprehending Java library XXL [3], allows for building a DSMS specific to concrete applications with full functionality.

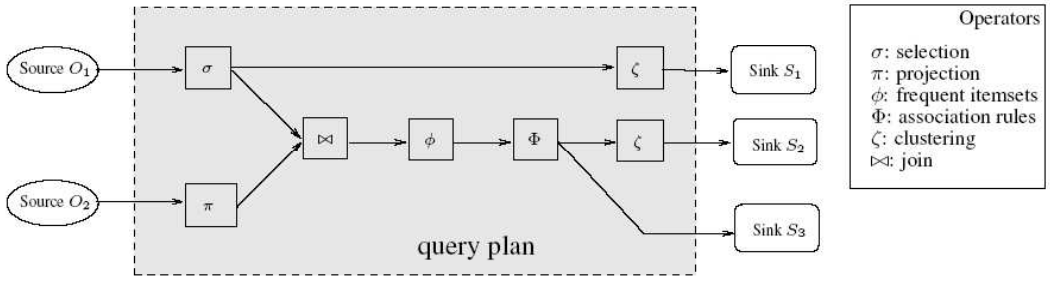


Figure 4: Example query plan

Usually, DSMSs manage multiple continuous queries specified by operator graphs, which allows for reusing shared subqueries. PIPES adapts this concept and introduces three types of graph nodes: sources, sinks and operators, where operators combine the functionality of a source and a sink with query processing functionalities. The resulting query graphs can be built and manipulated dynamically using an inherent publish and subscribe mechanism. This offers, among others, the possibility to adaptively optimize the processing according to resource-awareness. PIPES provides the operational run-time environment to process and optimize queries as well as a programming interface to implement new operators, sources and sinks.

The aimed resource-awareness mainly arises from implementing the mining techniques implemented as operators in PIPES. But why do we implement them as operators, rather than, for instance, building specialized DSMS for clustering and frequent itemset mining? The answer is, we want to be able to freely choose among any combination of these mining techniques between themselves, with other mining algorithms and, generally, with all operators implemented in PIPES. Fig. 4 pictures a small example to illustrate this approach (each operator is pictured by its corresponding algebra symbol). The stream data produced by two sources O_1 and O_2 is processed in three ways: sink S_1 receives all clusters determined (ζ) for O_1 after a preprocessing filter step. S_2 and S_3 receive association rules determined (Φ) on joined data from O_1 and O_2 – this implies finding the frequent itemsets (ϕ). S_3 works on the determined rules directly, while S_2 receives them after applying another clustering step (ζ) in order to identify interesting rules by grouping the related ones (similar to the approach in [10]). All of our mining operators take three dynamically adjustable parameters:

1. For clustering the number of final clusters *k*, for frequent itemsets the size *b* of a batch.
2. The queried time interval $[t_s, t_e]$.
3. An output interval *o*.

For frequent itemset mining, *b* represents the finest granularity of observed time and is equal to the internal update interval *u*. Thus, *o* should be a multiple of *b*. In the context of clustering only $o > u$ must hold. The resulting output is a data stream consisting of one stream element per passed output interval, containing the *k* clusters, the frequent itemsets, respectively, found in $[t_s, t_e]$. In correspondence to the aimed resource-awareness a user can decide between two possibilities to initialize the operator by providing:

1. Claimed qualities.
2. A memory limit.

In the first case, the amount of memory is calculated by adapting parameters inside the operator to achieve the claimed quality. For the second case, the adapting technique tries to find the ideal parameter settings to achieve optimal quality results, if possible, while adhering to the given memory limit. In this case, the user must define which of the supported qualities is prioritized and/or weight them accordingly.

4.2 Stream Clustering

In order to calculate the required amount of memory to solve the data stream clustering problem in the defined global quality Q, we need a stream clustering algorithm that allows us to change parameters (like

number of micro-clusters etc.) for manipulating the memory usage. With a look to related work, the algorithm of [1], called CLUStream, seems to be the ideal basis for our work. On the one hand, we have the number of micro-clusters, which has an influence on memory M and on methodical quality Q_M. Aggarwal et. al. showed that an increasing number of micro-clusters results in a better quality of the calculated clusters at the end. The first result was, that a proportion between q (the number of micro-clusters) and k (the number of end-clusters) fewer than three ends in a quite poor quality. Secondly, a proportion over ten outcomes with a quality that is only a few better than with proportions of eight or nine, but with a significantly higher amount of memory needed to store the micro-clusters and snapshots.
On the other hand, the size and the form of the snapshot-pyramid takes also amount of memory and has influence on the temporal qualities like Q_{Tg} and Q_{Tp}. In [1], no constraints on memory for storing the pyramid were made, what means that enough memory was available (main memory and disk) to do the work. As disc access is an expensive operation, holding the pyramid only in (limited) main memory is a valuable claim (also done in [2]). Moreover, disk usage also needs to get optimized and even disk space may be rare. At this point our new approach for estimating the amount of memory attaches.
The primary aim is to calculate the amount of memory in order to describe the function $r(Q)$ from Sect. 3. Parameters that influence the amount of memory are:

- k: number of end-clusters
- d: dimensionality of the data room
- $[t_s, t_e]$: time interval for calculating the clusters
- T: time since the beginning of the stream when the determination of the end-clusters starts
- Q_{Ma}: quality on clusters (in relation of the SSQ achievable with maximal q/k, equals 10 in our tests)

Secondary, we will present an extended version for storing snapshots in the pyramidal time frame. The old static pyramid has the disadvantage that at the beginning of storing snapshots, the amount of used memory is quite low compared to the overall amount of memory. That is why we dynamically adjust the width of the pyramid to achieve a better memory usage and to improve the temporal qualities Q_{Tg} and Q_{Tp}.

4.2.1 Estimating the amount of memory

Firstly, we calculate the number of micro-clusters q to be used within the algorithm. With the knowledge about reasonable values for q/k provided by the tests of Aggarwal et. al. (the interval (4, . . ., 10)) we can estimate q with the help of k and the quality Q_{Ma}, which shall be achieved. For poorer Q_{Ma} (e.g., aimed SSQ is double times the best SSQ) we can set q to $2 \cdot k$, on the other side for better Q_{Ma} (e.g., 0-5% away from the optimum) q has to be set between $4 \cdot k$ and $10 \cdot k$.
After estimating the number of micro-clusters q, we need to know how much memory one micro-cluster costs. Counting the costs in number of floating point numbers (FPN), one micro-cluster in the online component weights $(2 \cdot d+1+cLRU)$. For each dimension we have to store the first and the second moment plus the number of points within the cluster (that is why $2 \cdot d+1$ FPN) and constant information $cLRU$ for the LRU-strategy in the algorithm. Thus, the amount of memory for q micro-clusters in the online-component is determined by $q \cdot (2 \cdot d+1+cLRU)$ FPN. Remember this is only one part of the global amount of memory.
The other part is used for storing the snapshots within the pyramidal time frame. The proposed clustering operator for DSMS only uses main memory to store information, i.e., the pyramidal time frame is only stored in main memory and not on disk. Every snapshot S with a clear timestamp t contains q micro-clusters. The memory usage of micro-cluster in this situation is determined by $(2 \cdot d+1)$ FPN. The LRU-strategy information is not needed for offline execution of snapshots, i.e., it is not added to the memory usage. On the other side, one more FPN has to be stored to reflect the timestamp of the entire snapshot. If the pyramidal time frame contains $\#S$ snapshots the overall memory usage of the pyramidal time frame in FPN can be calculated as: $\#S \cdot (q \cdot (2 \cdot d+1) + 1)$. The overall global memory usage, which consists of the

online-component and the pyramidal time frame, is the sum of the two parts: $q\cdot(2\cdot d+1+cLRU) + \#S\cdot(q\cdot(2\cdot d+1) + 1)$.

In order to achieve good methodical quality Q_M, at first the proportion of q/k has to be set correctly. The form and the size of the pyramidal time frame also influence the temporal qualities Q_{Tg} and Q_{Tp}. The pyramidal time frame at moment T need to be that large (in width) that clustering analysis for the time interval $[t_s, t_e]$ can be done in a good way. We assume the average width of the pyramidal time frame (number of snapshots S: $\#S$ divided by depth: $log_\alpha(T)$) as a parameter for adjusting or regulating the output quality of the stream clustering. Tests on synthetic data have shown that an average width above two guarantees good clustering quality on the output clusters (q/k set above three). A width under 1.75 – 2 reduces the quality, because an approximately good time interval $[s_1, s_2]$ of two snapshots for $[t_s, t_e]$ cannot be found within the pyramidal time frame. If the difference between timestamp of s_1 and t_s (or timestamp of s_2 and t_e) is too large, the approximate answer for the time interval $[t_s, t_e]$ is quite poor. A pyramidal time frame with greater average width reduces this problem and guarantees better temporal quality (Q_{Tg} and Q_{Tp}) on the output. Of course, a bigger pyramid in width results in higher memory usage. In our approach we try to set the average width above two to guarantee good results with higher memory usage and fewer than two for poorer quality results with lower memory usage. Assume we process a stream for T time units and use a dividing factor of α. The depth of the pyramidal time frame at T is given by $\lceil log_\alpha(T) \rceil$. Let w denote the estimated average width of the pyramidal time frame, than $\#S$ becomes $w\cdot\lceil log_\alpha(T) \rceil$. The memory usage for the pyramidal time frame in FPN can be determined by: $w\cdot\lceil log_\alpha(T) \rceil\cdot(q\cdot(2\cdot d+1)+1)$ and so the overall memory usage is described by $q\cdot(2\cdot d+1+cLRU)+w\cdot\lceil log_\alpha(T) \rceil\cdot(q\cdot(2\cdot d + 1) + 1)$. This formula is the main key to describe the proposed function $r(Q)$.

In summary, we can distinguish between adapting the q/k proportion and the average width w of the pyramidal time frame to get the following qualities on the output:

quality	$\frac{q}{k} < 3$ (too small)	$3 \leq \frac{q}{k} \leq 10$ (ideal)	$\frac{q}{k} > 10$ (too large)
$w < 1.75$ (too small)	low	low	low
$1.75 \leq w \leq 2$ (middle)	low	good (actual intervals)	good (actual intervals)
$w > 2$ (ideal)	low	good (all intervals)	good (all intervals)

The combination of q/k and w determines the overall output quality Q of the k clusters. If q/k and w is set for good quality the global quality will also be good. If one of the two parameters is set for a worse case the global quality will also be poorer.

To calculate the resulting quality (function $r'(R)$) by a given amount of memory R, we use the above formula for the global amount of memory converted to w: $w = (R - q\cdot(2\cdot d+1+cLRU)) / (\lceil log_\alpha(T) \rceil\cdot(q\cdot(2\cdot d+1) + 1))$. By varying the number of micro-clusters q from $10\cdot k$ down to k the corresponding average width w of the pyramidal time frame is calculated. If w is above 2 for a good choice of q ($3\cdot k$ to $10\cdot k$) good clustering quality can be achieved. Otherwise w or q/k are too small to guarantee near optimal results.

4.2.2 Dynamical pyramidal time frame

Besides estimating the memory usage for a given quality, we introduce the extended pyramidal time frame, which uses as much memory as possible in every moment of stream clustering. In any situation, we need to be able to dynamically adjust the width of the pyramidal time frame according to the actual time, depth and memory usage. The difference between [1] and our approach is the implementation of the pyramidal time frame. While in the original version memory is unbounded and provides a low memory filling factor at the beginning, our approach uses the whole memory at each moment to improve the temporal qualities Q_{Tg} and Q_{Tp} and to challenge with varying memory sizes. At the beginning the width of the pyramidal time frame is set very high to deal with the low depth. While the depth increases with time

the width has to be reduced in order to avoid an overfilling of the memory. Thus, at each moment of the clustering algorithm the memory is filled near optimal and better quality results can be provided.
In order to achieve the dynamic described above, we introduce a memory filling factor f defined as follows: $f := (actual\_memory\_usage) / (maximal\_memory)$. The actual memory usage can be calculated using the above memory formula or by incrementally adding and reducing memory allocations.
By manipulating the width of the pyramidal time frame, over 85% of the maximal memory is used at every time. For the manipulation of width we have to check and interpret the filling factor correctly. For f, we distinguish between three different intervals:

1. $f < 0.85$: memory utilization is too bad.
2. $0.85 \leq f \leq 1.0$: the pyramidal time uses enough memory.
3. $f > 1.0$: overfill situation.

For each situation an individual reaction is processed:

1. The width of the pyramidal time frame is increased by one.
2. The width of the pyramidal time frame is unchanged.
3. The width of the pyramidal time frame is reduced, as long as f is above 1.0.

While in the first two cases no deleting operations need to be processed, an overfill situation as in three causes deletion of snapshots. Reducing the actual amount of memory includes the following operations:

1. Decrease the pyramidal time frame width by one.
2. If an order i of the pyramid has more snapshots than the given width, delete the oldest snapshots, until the width is equal to the number of snapshots.
3. Check the actual filling factor: if $f > 1$ go to 1., otherwise exit.

The procedure for adjusting the width in combination with the filling factor f is processed every time a snapshot is added to the pyramidal time frame. The approach can also be used in environments with varying memory (for example memory falls from 10000 to 5000 or rises from 5000 to 10000). In these cases the pyramidal width is decreased with falling memory or increased with rising memory.

4.3 Frequent Itemset Mining

As frequent itemset mining algorithms on data streams usually produce approximate results, there may be some false positives in the resulting output. Therefore, we need an algorithm that guarantees an error threshold.
Additionally, the approach has to be time-sensitive. The FP-Stream approach in [4] is capable to satisfy these requirements. Asked for the frequent itemsets of a time period $[t_s, t_e]$, FP-Stream guarantees that it delivers all frequent itemsets in $[t_{s'}, t_{e'}]$ with frequency $\geq \sigma \cdot W$, where W is the number of transactions the time period $[t_{s'}, t_{e'}]$ contains. $t_{s'}$ and $t_{e'}$ are the time stamps of the used tilted time window table (TTWT) and correspond to ts and te, depending on Q_{Tg}. The result may also contain some itemsets whose frequency is between $(\sigma - \varepsilon) \cdot W$ and $\sigma \cdot W$.
Our first goal was to find out how much memory the algorithm needs in order to deliver results in a certain quality. We conducted an extensive series of tests for the algorithm's memory requirements in different parameter settings. Secondly, we extended the approach from [4] so we can cope with limited memory, resulting in the algorithm's resource- and quality-awareness.

4.3.1 Estimating the amount of memory

Firstly, we estimate the amount of memory a pattern tree needs within a given parameter setting. For estimating the overall memory requirements of a pattern tree, we need to know the number of nodes in a pattern tree, and the amount of memory each individual node needs. In [4] an upper bound is given for the size of a TTWT by $2 \lceil \log_2(N) \rceil + 2$, where N is the number of batches seen so far. This is because the algorithm uses a logarithmic TTWT to the basis 2 and is designed with one buffer value between each two values except the two most recent. In our experiments the actual number of entries averaged over all TTWTs in a tree was always less than 70% of this upper bound. Besides the size of the TTWT, each node

in a pattern tree needs some fixed sized memory c for storing the itemset it represents and information about its parent node and child nodes.
The number of nodes in a pattern tree depends on several facts. On the one hand there are the values of the algorithm's input parameters ε and b. The value of σ does not affect the number of nodes in a pattern tree, because the adding and dropping conditions in a pattern tree depend only on ε, though σ is a quality measure! On the other hand there are the characteristics of the underlying data stream like the average number of items per transaction and the number of distinct items in the stream. We have not yet found a concrete formula describing the maximum number of nodes in a pattern tree for specific parameter settings.
But, we conducted a series of tests showing that the maximum number of nodes remains constant over time while the algorithm processes an infinite series of transactions. Thus, for certain values of ε and b and specific characteristics of the underlying data stream we know the maximum number of nodes in a pattern tree. In general, we can state that a large pattern tree leads to mining results with better quality than a small pattern tree.
The fact that the overall space requirements stabilize or grow very slowly as the stream evolves was already shown in [4]. The authors investigated different values for σ and the average number of items per transaction. [5] additionally showed the same effect for varying values of ε. We extended these tests to different values of the number of distinct items in the stream and the size of b. With a constant input rate the size of b affects the number of transactions a batch contains. As we will show in Sect. 5 the value of b is the only one that has very little impact, as long as the number of transactions in a time interval of size b exceeds a certain threshold (depending on ε and some data stream characteristics). This is in order to fade out the effect of temporarily frequent itemsets that have no significance for the overall mining result. According to Sect. 3 we can determine r as: $number\text{-}of\text{-}nodes(\varepsilon,b) \cdot [2 \cdot \lceil \log_2(N) \rceil + 2 + c]$, representing a heuristic approximation of the resources actually needed.

4.3.2 Dynamic tree size adjustment

In analogy to Sect. 4.2.2 we introduce an extended approach of [4] that can cope with limited memory and uses the available memory as effective as possible. If there is not enough memory available for finding frequent itemsets in the claimed quality, we need to dynamically adjust the size of the pattern tree according to the actual memory conditions. At first we implemented the approach in [4] and examined its memory requirements for different parameter settings. After that, we considered a couple of possibilities to control the memory requirements of the pattern tree. Thus, we gained an extended approach that has several alternatives for controlling the tree size depending on the user's quality weighting. We used the same memory filling factor f as in Sect. 4.2.2. Depending on the filling factor, our approach takes action to reduce or increase the size of the pattern tree. The possibilities for manipulating the size of the tree are:

1. Adjust the value of ε while keeping σ constant, i.e., change the approximation quality Q_{Ma} of our mining results according to available memory.
2. Adjust the value of σ according to available memory while keeping ε/σ constant, i.e., keep the quality Q_{Ma} of the mining result constant but change the quality of interestingness Q_{Mi}.
3. Limit the size of each TTWT according to available memory, i.e., change the time range quality Q_{Tr}.
4. Adjust the value of b according to available memory, i.e., change the time granularity quality Q_{Tg}.

Since we can estimate the memory requirements of a pattern tree for a given set of parameters, we can also estimate the maximum number of nodes our pattern tree may not exceed in order to adhere to a certain memory limit. In each step we assume the size of a TTWT to be at its upper bound. The changes in this upper bound are estimated only at constant intervals (in our tests every 100 batches), because we want to avoid registering negligible changes of the maximum number of nodes after every batch.

Option 1: Adapting ε. A first option for manipulating the size of the pattern tree is to change the value of ε, i.e., change Q_{Ma}. Therefore, we estimate an ideal ε that results in a pattern tree with approximate as many nodes as possible, rather than taking the value of ε as an input parameter. However, the user may specify a lower bound for the value of ε, i.e., an upper bound for the quality of the mining result.
Since we are not yet able to calculate the maximum number of nodes for a given amount of memory exactly, we also cannot estimate an ideal value of ε. Our algorithm adjusts the value of ε depending on the filling factor f after every processed batch as follows:

- $f < 0.85$: Decrease ε by ten percent. Use this ε for all following batches.
- $0.85 \leq f \leq 1.0$: The value of ε remains fixed.
- $f > 1.0$: Increase ε by ten percent. Conduct tail pruning at the TTWTs of each node in the pattern tree and drop all nodes with empty TTWTs. Repeat these steps as long as $f > 1.0$.

In this approach we have to store the value of ε we used in each specific time interval, in addition to monitoring the number of transactions in each interval. If two TTWT frequency entries n_i and n_j are merged, we also have to merge the according ε values ε_i and ε_j in order to determine the achievable quality for this time period. We can average the two distinct values of ε as described by equation 1. w_i and w_j denote the sizes of time intervals t_i and t_j.

$$\varepsilon' = (\varepsilon_i w_i + \varepsilon_j w_j) / (w_i + w_j) \tag{1}$$

It was shown in [4] that for a fixed ε, if all itemsets whose approximate frequency is larger than $(\sigma - \varepsilon) \cdot W$ are requested, then the result will contain all frequent itemsets in the period $[t_{s'}, t_{e'}]$. Thus, in our extended approach all itemsets with approximate frequency $(\sigma - \varepsilon') \cdot W$ are returned. The value of ε' depends on the time intervals contained in $[t_{s'}, t_{e'}]$. If $[t_{s'}, t_{e'}]$ covers only time intervals where the same value of ε was used for all batches, then ε' is equal to this ε. If $[t_{s'}, t_{e'}]$ contains time intervals where the value of ε differs, the value of ε' becomes:

$$\varepsilon' = (\sum_{i=s'}^{e'} \varepsilon_i w_i) / W \tag{2}$$

In summary, in our first option we control the amount of memory used by varying the value of ε and so ε/σ, which reflects the quality of approximation Q_{Ma} of our mining result.

Option 2: Adapting σ. The second option for adjusting the size of the pattern tree is to alter the value of σ. As σ does not influence the size of the tree directly, ε/σ remains the same. That is why the user does not provide a fixed value of σ, but rather claims for a certain ε/σ that should be guaranteed. Again, the user may also specify a lower bound for the value of σ. The handling of σ is analog to the handling of ε in the first option with the only difference, that ε has to be adjusted in parallel in order to keep ε/σ constant. The analogy also applies for determining the value of σ':

$$\sigma' = (\sum_{i=s'}^{e'} \sigma_i w_i) / W \tag{3}$$

The value of ε' again results from equation 2. Returned are all itemsets whose approximate frequency is larger than $(\sigma' - \varepsilon') \cdot W$. We guarantee to deliver all itemsets whose actual frequency is $\geq \sigma'$. Because ε'/σ' is kept constant as requested by the user, the quality Q_{Ma} meets the user's requirements. By adjusting the value of σ we alter the quality Q_{Mi} by varying the frequency an itemset must occur with in order to belong to the delivered set of results.
But, what is the difference between the first and the second option? In the first option we keep σ constant and change ε. Thus, we can request all itemsets with a minimum support of $(\sigma - \varepsilon') \cdot W$ and accept a poorer quality Q_{Ma}. In the second option, we modify σ but keep ε/σ constant. Thus, we also keep the user defined quality Q_{Ma} constant and return all itemsets with a minimum support of $(\sigma' - \varepsilon') \cdot W$. In this case, only the more interesting itemsets are found, in terms of σ as an interestingness measure from Q_{Mi}. An effect on memory usage is achieved in both options. Moreover, in both options the methodology of pruning TTWTs remains unchanged.

Option 3: Limiting the size of the TTWTs. The third option is to limit the size of each TTWT to a fixed value. This results in a restriction of how far we can look back into the history of the processed stream, because we limit the number of time intervals for which we store frequency information. Thus, we are not able to deliver results from a time period that includes batches lying farther back in history than the information we recorded. According to Sect. 3 this leads to an impairment of the time range quality Q_{Tr}. As the number of time intervals stored in one entry of a TTWT increases logarithmically, saving a large amount of memory demands for limiting the size of a TTWT drastically. In this way, the information of a considerable portion of the observed batches would get lost. Lowering the maximal size by small values, only a small amount of memory can be saved.

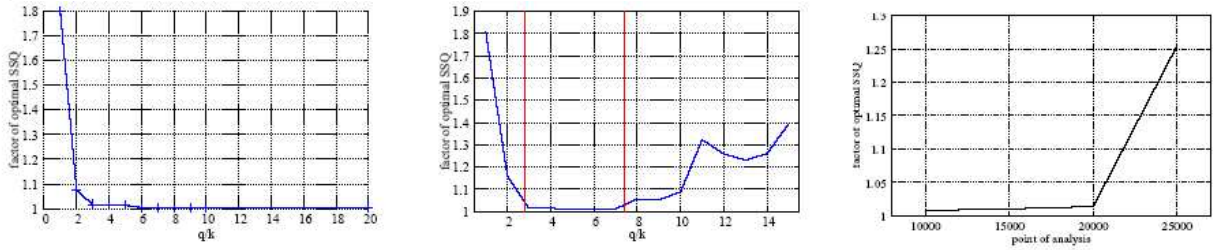
Option 4: Adapting b. The last option in order to limit the used memory is to adjust the value of *b*. Assuming a constant input rate, the size of *b* affects the number of transactions a batch contains. Increasing *b* leads to an impairment of the time granularity quality Q_{Tg}, as we reduce how exact we can look back.

Our experiments reveal that the number of transactions per batch does not affect the number of nodes in the pattern tree significantly. By increasing the value of *b* we can only reduce the total number of entries in a TTWT while processing a finite part of the stream. Considering infinite data streams, since the size of a TTWT grows logarithmically, the number of entries in a TTWT will converge to the same value for all choices of *b*. Therefore, this option will only have a significant impact on the required memory if it is combined with a limitation of the TTWT size. Combining both, we can reduce the granularity of a time interval and look back farther in the history of the data stream using the same number of TTWT entries.

5 EVALUATION

The purpose of the following evaluation is twofold: at first we will show how we achieve the aimed quality-awareness. To show this, we will evaluate how good the algorithms achieve a claimed quality and how exact the corresponding calculations are. In the context of the proposed stream mining approach quality-awareness comes along with the adaption to provided resources and the determination of needed resources concerning aimed quality measures. This is the second aim of this section: we will show that memory requirements are approximated satisfyingly and that they are met finally, and which conclusions we can draw to the achieved quality.

Quality-Aware Clustering. First of all we show that the assumptions made in Sect. 4.2 hold. Fig. 5(a) shows the results for *q/k* from [1] again, reproduced with our implementation. The diagram in Fig. 5(b) assumes that a final constant amount of memory is used. The variation of the proportion *q/k* influences Q_M.

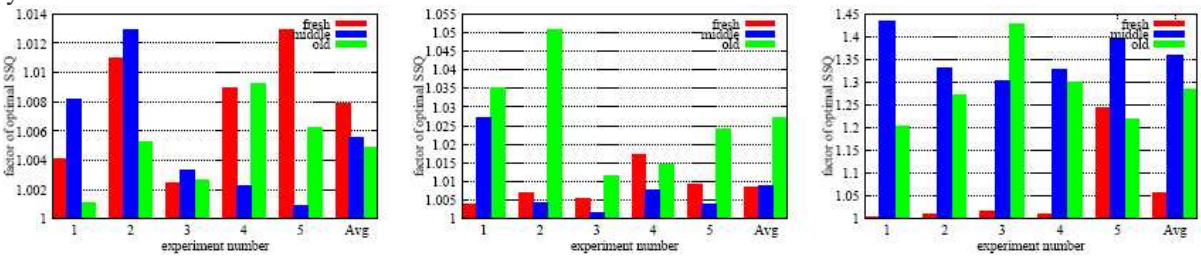


(a) q/k – Quality in CLUStream (b) Varying q/k, constant memory (c) Temporal quality

Figure 5: Observation of q/k in original CLUStream and with limited memory & temporal quality

With *q/k* too low (first vertical line) SSQ increases, oversized *q/k* (second vertical line) results in higher memory usage, which reduces the average width of the pyramidal time. The interval of *q/k* between the two above situations is characterized by good clustering results (0-5% beyond the optimum).

In order to evaluate the quality-aware clustering approach we use synthetic data sets with around 25000 data points, generated using Gaussian distributions for the elements of each cluster. As streams are usually evolving, at first we define the centers of the clusters and afterwards let them change every 5000th point in mean and variance. To measure the output quality we define a data point interval $[p_1, p_2]$, which is clustered with a standard clustering algorithm (k-Means, LBGU) to get the optimal cluster centers and SSQ. After calculating the optimal centers we start the stream processing to build the pyramidal time frame.



(a) Best quality (b) Middle quality (c) Low quality

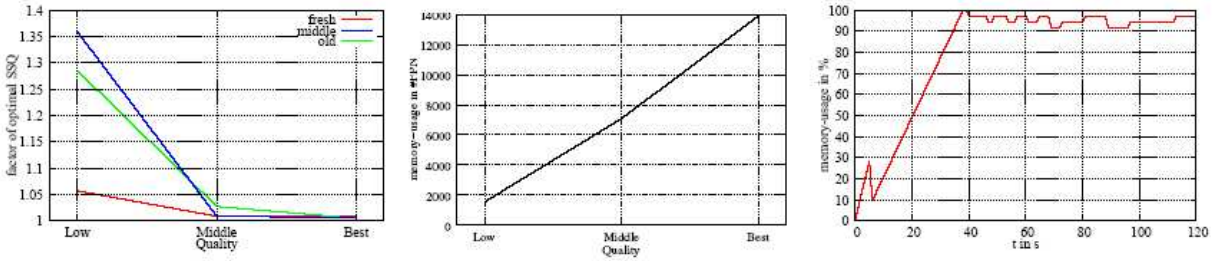
Figure 6: Comparison of different aimed qualities

At a maximal time T, the pyramidal time frame is asked for two snapshots to approximate the above given interval. The difference between these two snapshots is calculated and a local clustering algorithm is used to cluster the q micro-clusters into k final clusters. Now we can compare the optimal clusters, determined statically, with the clusters found with stream processing, by measuring the factor of approximation between both. Unless mentioned otherwise the algorithm uses the following parameter-settings: α=2, initPoints=1000, snapshotsaving-interval=1s and input rate=200 points/s. In order to evaluate time sensitiveness and its effect on the output quality we divided the synthetic data set into three different intervals, called Fresh, Middle and Old. The Fresh interval includes all points from position 16000-19000, the Middle interval all from 11000-14000 and the Old all points from 6000-9000. For each interval we did five measures for calculating the SSQ. Fig. 6(a)-6(c) show the quality results for low ($q/k=2, w=2$), middle ($q/k=5, w=4$) and best ($q/k=8, w=5$) quality.

The main conclusion from these experiments is the following: a better adjustment at the beginning (low, middle or best) results in better average quality on the output. This means that the chance to get best results (lower than 5% approximation of the optimum) can be achieved by using the best strategy at the beginning, which on the other side needs the highest amount of memory. But, also the middle variant produces good output quality in relation to the amount of memory it is using. The low variant produces the worst clustering quality, because the number of micro-clusters used is too low. Moreover, the average width of the pyramidal time frame cannot achieve a good temporal approximation. Another conclusion is as follows: the difference between fresh, middle and old intervals results in quality differences if not enough memory is used. Because of the deleting strategy of the pyramidal time frame old snapshots are deleted earlier than newer ones.

This is why the temporal approximation of the pyramid gets worse and snapshots for older intervals cannot approximate the given time interval satisfyingly (Fig. 7(a)). If the provided memory is high enough (middle and best strategy), also old intervals can be calculated with even good quality. Fig. 7(b) illustrates the differences in memory usage between the three qualities. In Fig. 5(c) we briefly show the SSQ dependence on time, by clustering the same data set all 5000 arriving points.

Finally, we investigate the optimal memory usage of our new dynamic pyramidal time frame. In Fig. 7(c) the memory usage for the middle strategy experiment is given over the time. In the first six seconds, a linear growing memory usage shows the storage of the 1000 initial data points for getting the initial micro-clusters.



(a) Varying quality (b) Varying memory (c) Memory of pyramidal time frame

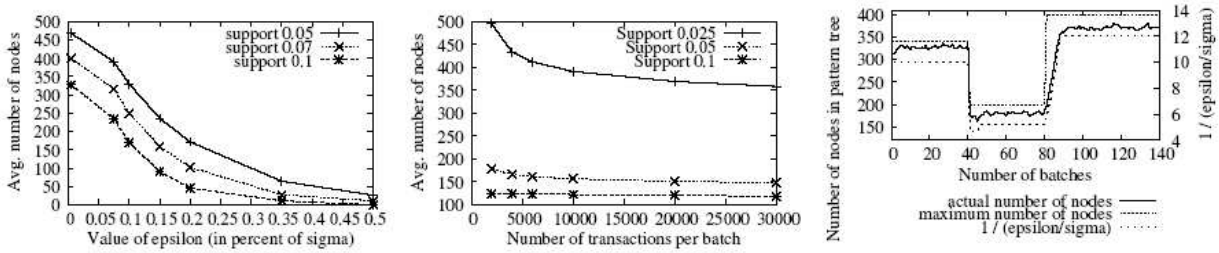
Figure 7: Comparison of different strategies and memory usage of dynamic pyramidal time frame

After that the initial points are used no longer, that is why the memory usage reduces. With increasing time the memory usage increases linear again, until the maximal usage of 100% is achieved. At this point, the dynamic pyramidal time frame reduces their width to free space for the following snapshots. The same procedure is done until the end of the stream. The percentage of used memory never drops below 85-90% and the memory utilization is nearly optimal until the end.

Quality-Aware Frequent Itemset Mining. Before evaluating our quality-aware frequent itemset mining approach we conducted a series of tests with the original FP-Stream algorithm. We figured out how the algorithm behaves for different parameter settings of ε, σ and b. We used synthetic data generated by the IBM market-basket data generator. In the first set of experiments 1M transactions were generated using 1K distinct items and an average transaction length of 3. All other parameters were set to their default values from the generator.

Since in the original approach a batch does not cover a constant period of time but a constant number of transactions, we set the size of a batch to 5000 transactions. Fig. 8(a) shows some of our experimental results with the original algorithm. We measured the average number of nodes in a pattern tree for three different values of σ. For each σ we run the algorithm with various ratios between ε and σ to simulate different quality demands. As expected the number of nodes decreases with rising ε and σ.

Fig. 8(b) shows results of another series of tests we conducted with the original algorithm. We processed the test data with different values of σ (always setting $\varepsilon = 0.1 \cdot \sigma$) and with various batch sizes.



(a) Varying support and epsilon (b) Varying support and batch size (c) Resource-awareness

Figure 8: Average number of nodes in a pattern tree

The number of nodes in the pattern tree does not decrease significantly when the batch size is highly raised. We also processed test data having more average items per transactions (5, 10) and/or more distinct items (5K, 10K) and we additionally tried different batch sizes (1000 to 30000). The main conclusion is always as stated above. One remarkable thing we noticed is that for small batch sizes the number of nodes in a pattern tree is far from being constant. For example, when processing testing data with 1M transactions, 1K distinct items and average transaction length of 3 we set $\sigma = 0.025$, $\varepsilon = 0.1 \cdot \sigma$ and the batch size to 1000 transactions. The number of nodes in the resulting pattern tree oscillated between under 1000 to nearly 5000. When processing the same data set with higher values of σ, ε or batch size this range got significantly smaller. For a support of 0.07 the difference between the minimum

and the maximum value of the number of nodes was 45. Remainders of this effect can be seen in Fig. 8(b) for $\sigma = 0.025$ by the sharp decrease of the number of nodes for low batch sizes.

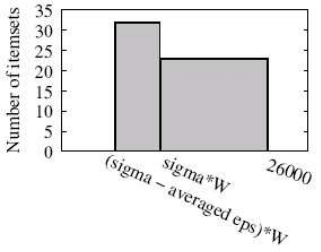
For evaluating the quality-aware frequent itemset mining approach, we again generated data sets with 1M transactions using 1K distinct items and an average transaction length of 3. Our algorithm consumed the stream of transactions from a source with a constant output rate of 3 seconds. The value b of the finest granularity of time was set to 15000 seconds, so each processed batch contained 5000 transactions. The value of σ was set to 0.05.

Firstly, we wanted to demonstrate that our approach can cope with changing memory conditions. Instead of limiting the actual amount of available memory we limited the number of nodes that the pattern tree may have. One could calculate the actual amount of memory needed, since the maximum size of a TTWT can be estimated as described in Sect. 4.3.1.

Initially we limited the maximum number of nodes the generated pattern tree may have to 340. As we do not have a formula yielding the maximum number of nodes in a pattern tree for a given set of parameters, we had to choose an adequate value of ε for the algorithm. Our previous experiments showed, that $\varepsilon = 0.1 \cdot \sigma = 0.005$ would be a good value to start.

We started the algorithm and decreased the maximum number of nodes after 40 batches to 200 nodes. We then raised it again after 40 batches to a value of 400 nodes. Fig. 8(c) shows the algorithm's behavior. After 40 batches, it had to raise the value of ε several times to get its filling factor below 1. Then, after the next 40 batches, the algorithm lowered the value of ε after every processed batch until a tree size was reached that was close enough to the maximum according to the filling factor. Fig. 8(c) also displays the quality of the stream mining for every batch, i.e., the value of $1/(\varepsilon/\sigma)$.

With our second series of tests of our quality-aware approach we demonstrate the quality of our mining results. We started the algorithm using option 1 of our proposed techniques for adjusting the size of the pattern tree, i.e., we changed the value of ε when necessary.



The experimental settings are equal to these of the above test. Only this time we set $\sigma = 0.025$ and $\varepsilon = 0.2 \cdot \sigma$ to get any frequent itemsets at all. The experiments show that we estimated the overall quality of our mining results ranging over several batches (and thus several values of ε) correctly and that it is in fact suitable to average distinct values of ε as described.

First we ran our approach once to get the value of ε', which was approximately 0.00765. Then we used the original FP tree algorithm to get the actual set of frequent itemsets from our test data. We also got the actual frequencies of all itemsets having frequencies between $(\sigma - \varepsilon')$ and a little less and the required support. Then, we ran our quality-aware approach and asked for the set of frequent itemsets after the 120th batch. We set the time period $[t_s, t_e]$ to the whole period of time the stream was processed so far. Thus, the window we requested contained $W=600000$ transactions. After we received the result, we compared the output to what we knew from the FP-growth method. For each itemset our approach delivered, we looked up its real frequency and printed these information in the histogram shown by Fig. 9. All itemsets our approach output were inserted in frequency buckets, according to their real frequency, which we gained through using the FP tree algorithm.

Fig. 9 shows that the output contained no itemsets having frequency less than $(\sigma - \varepsilon') \cdot W$. The histogram also shows that exactly 23 itemsets having frequency of at least σ were delivered. These itemsets are the same as FP-growth delivered.

Summary. All in all the experiments met our expectations. The approximations of memory usage hold, the quality deduced from the available resources is close to the quality actually achieved. This holds for all examined quality measures, as far as our tests can show that. Of course, we have to do a couple of extended test series, including different parameter settings, varying stream characteristics and a deeper analysis of several (classes of) quality measures. With the results of these subsequent tests we could finally demonstrate the quality- and resource-awareness already achieved in this work.

6 CONCLUSION

In this paper, we argued that quality of analysis results is an important issue of mining in data streams. The reason is that stream mining can be usually performed only on a resource-limited subset or approximation of the entire stream, which affects different measures of data quality. Based on a discussion of such quality measures for stream mining, we have investigated and enhanced two specific mining approaches for clustering and frequent itemsets in order to estimate the quality depending on the current resource situation (mainly the available memory) as well as to allocate resources needed for guaranteeing user-specified quality requirements.

With this work, we have addressed only operator-locally parameters like the amount of memory available for this specific operator. In future work, we plan to take also global properties of the whole analysis pipeline into account, e.g., load shedding and windowing operators, which have an impact on the output quality, too.

REFERENCES

[1] Ch. C. Aggarwal, J. Han, J.Wang, and Ph. S. Yu. A Framework for Clustering Evolving Data Streams. In *Proceedings of VLDB 2003, Berlin, Germany*, pages 81–92, 2003.

[2] Ch. C. Aggarwal, J. Han, J. Wang, and Ph. S. Yu. A Framework for Projected Clustering of High Dimensional Data Streams. In *Proceedings of VLDB 2004, Toronto, Canada*, pages 852–863, 2004.

[3] J. V. d. Bercken, B. Blohsfeld, J.-P. Dittrich, J. Kr¨amer, T. Sch¨afer, M. Schneider, and B. Seeger. XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In *Proceedings of VLDB 2001*, pages 39–48, 2001.

[4] C. Giannella, J. Han, J. Pei, X. Yan, and P. S. Yu. Mining Frequent Patterns in Data Streams at Multiple Time Granularities. In Workshop on Next Generation Data Mining, 2003.

[5] C. Giannella, J. Han, E. Robertson, and C. Liu. Mining Frequent Itemsets over Arbitrary Time Intervals in Data Streams. Technical report, Indiana University, 2003.

[6] S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering Data Streams: Theory and Practice. *IEEE Transactions on Knowledge and Data Engineering, 15(3):515–528*, 2003.

[7] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. In *Proceedings of SIGMOD 2000, Dallas, USA*, pages 1–12, 2000.

[8] J. Krämer and B. Seeger. PIPES - A Public Infrastructure for Processing and Exploring Streams. In *Proceedings of SIGMOD 2004, Paris, France*, pages 925–926, 2004.

[9] P. Tan and V. Kumar. Interestingness Measures for Association Patterns: A Perspective. Technical report, University of Minnesota, 2000.

[10] H. Toivonen, M. Klemettinen, P. Ronkainen, K. Haton, and H. Mannila. Pruning and grouping discovered association rules. In *ECML-95 Workshop on Statistics, Machine Learning and Knowledge Discovery in Databases, Greece*, pages 47–52, 1995.

[11] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: An Efficient Data Clustering Method for Very Large Databases. In *Proceedings of SIGMOD 1996, Montreal, Canada*, pages 103–114, 1996.