Optimizing Sequential Pattern Mining Within Multiple Streams

Daniel Töws, Marwan Hassani, Christian Beecks, Thomas Seidl

Data Management and Data Exploration Group
RWTH Aachen University
Germany

{toews, hassani, beecks, seidl}@cs.rwth-aachen.de

Abstract: Analyzing information is recently becoming much more important than ever, as it is produced massively in every area. In the past years, data streams became more and more important and so were algorithms that can mine hidden patterns out of those non static data bases. Those algorithms can also be used to simulate processes and to find important information step by step. The translation of an English text into German is such a process. Linguists try to find characteristic patterns in this process to better understand it. For this purpose, keystrokes and eye movements during the process are tracked. The *StrPMiner* was designed to mine sequential patterns from this translation data.

One dominant algorithm to find sequential patterns is the PrefixSpan. Though it was created for static data bases, lots of data stream algorithms collect batches and use the algorithm to find sequential patterns. This batch approach is a simple solution, but makes it impossible to find patterns in between two consequent batches. The *PBuilder* is introduced to find sequential patterns with a higher accuracy and is used by the *StrPMiner* to find patterns.

1 Introduction

When translating an English text into German, a lot of cognitive efforts have to be made. The e-cosmos project is a project in the humanities, which tries to find characteristic patterns in human behavior while being confronted with linguistic challenges. As a part of this project test subjects, with German as their native language, were given the task to translate an English text into German. While the subjects translated the text, their eye movement and keystrokes were collected. The linear representation in Figure 1 helps to understand the process and gives small insight into the cognitive process. The pauses marked by the stars, help to find challenges for the subjects but they provide no information on what the subject is doing during this time, aside from not using the keyboard.

To find those hidden information sequential pattern mining algorithms can be used over the resulting data stream. Sequential pattern mining is a special case of the frequent item set mining, where patterns have to be frequent subsequences of the stream. Each pattern

wid[*12.672]ersetzt•sich•*******weit********⌫⌫*******⌫⌫⌫⌫
⌫⌫⌫⌫⌫⌫⌫⌫⌫⌫⌫⌫⌫⌫⌫stru⌫äu*bt•sich•gegen•weiteres*•⌫⌫n••Dr⌫
⌫⌫Druck

Figure 1: The *Linear Representation* shows part of the translation process. The stars mark pauses in which the subject did not use the keyboard for at least one second.

has to appear a certain number of times within one batch to count as a sequential pattern.

Stream data environments provide data sets that change constantly and provide new information every second. Because of this, each item can only be looked at once and then has to be forgotten.

For this, the algorithm will be split into an online and an offline part. The online part handles the arriving data itself. If the data needs simplification, or preprocessing of any kind, this will be performed by the online part. The detailed analysis will then be done by the offline component. For example some algorithms use the online component to collect data batches, so that the offline component can perform the algorithm for many items in parallel.

Additional challenges arise when looking at multiple streams at once, as patterns can be part of one or multiple streams. In this case the eye movement and keystrokes provide two different streams of data which can be analyzed to help understand the process of text translation. Using the data stream approach to evaluate the information step by step, reconstructs the whole process and gives deep insight for each session.

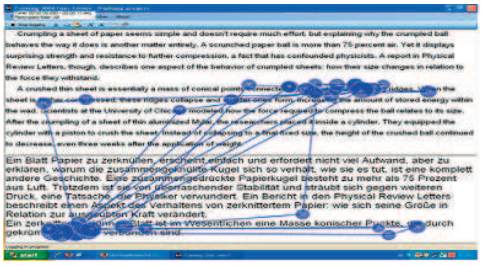


Figure 2: During the experiment eye gazes and keystrokes are recorded. The top half shows the original text and the bottom half the translated version. Fixations are marked in blue.

There are a lot of different algorithms that mine sequential patterns from data streams, but most of them use a batch approach. On the one hand the batch approach is a simple and fast solution to mine sequential patterns in a stream. On the other hand it has a room for errors. To avoid those errors, we will introduce two new algorithms. The PatternBuilder *PBuilder* mines sequential patterns for given data, without using the batch approach. The Streaming Pattern Miner *StrPMiner* uses the *PBuilder* to find sequential patterns in multiple streams and keeps track of their quality.

In section 2, this paper will discuss the related work that has been done in the past. Section 3 looks at the preliminaries of sequential pattern mining. Also it will highlight the problem

with the batch approach. In section 4 two new algorithms are introduced to mine sequential patterns. The *PBuilder* will then be tested and the results will be discussed in section 5. The paper concludes with a summary and a look to future directions in section 6.

2 Related Work

Optimizing sequential pattern mining is an important task in the streaming data mining field which lead to a lot of different algorithms. A base algorithm for many approaches [MDH08],[SEM11],[WC07], [CCPL14] is the *PrefixSpan* algorithm [PHMA$^+$01]. The *PrefixSpan* algorithm was designed for a static data environment. Because of this it can use the A priori assumption that every element contained in a frequent sequential pattern, also has to be frequent. In a given data base, the *PBuilder* will first look for frequent elements which are sequential patterns of length one. Following the A priori assumption, patterns can not be frequent, if they are created with elements or smaller patterns, that are non frequent. Using this, the *PrefixSpan* recursively creates new sequential pattern candidates, by combining a sequential pattern with a frequent element. All new created patterns will be counted and if they are frequent, *PrefixSpan* will search recursively for even longer patterns.. All algorithms using the *PrefixSpan* in a stream environment collect data in a batch instead of evaluating each item alone.

Since the streaming approach allows to only look at data once, algorithms have to make compromises in order to provide fast results. [MDH08] proposes two algorithms with different pruning strategies, the *SS-BE* and *SS-BM* algorithm. These algorithms restrict memory usage but are able to find all true sequential patterns and allow an error bound on the false positives. The patterns are saved in a new designed tree structure, the T_0 tree. Each pattern will only be saved in the tree, if it occurred frequently in the recent time. Both of those algorithms use the *PrefixSpan*.

In a static data set, all information needed for the algorithm are provided from the beginning, while in the streaming approach new data arrives every second, thus, patterns that where not frequent in the beginning may become frequent later on. Yet it is impossible to save every pattern and its information. The *FP-stream* [GHP$^+$03] solves this issue by saving information in different time granularities. The newer the information, the more accurate it will be displayed. Another way to solve the memory problem is by using a sliding window model, in which only the most recent data is being looked at. The *MFI-TransSW* algorithm [LL09] optimizes this concept. The algorithm works in three steps: window initialization, window sliding and pattern generation.

Previously described algorithms only provide solutions for one stream. In cases of multiple streams in parallel, the *MSS-BE* algorithm [HS11] is an idea to find sequential patterns in an multiple stream environment, where pattern elements can be part of different streams.

The algorithms mentioned above either provide solutions for frequent pattern mining or find sequential patterns by using batches.

3 Preliminaries: Optimizing Sequential Pattern Mining

A stream S is defined by an infinite ordered set $S = \{(s_1, t_1), (s_2, t_2), (s_3, t_3), ...\}$, with s_i being the item and t_i the arrival time of the item. A sequential pattern of S is defined as [HS11]:

Given a sequence of items $p = \{a_1, a_2, ..., a_n\}$. The sequence $p' = b_1, b_2, ..., b_m$ is a *subsequence* of p, if there exist integers $i_i < i_2 < ... < i_m$ such that $b_1 = a_{i_1}, b_2 = a_{i_2}, ..., b_m = a_{i_m}$. The *support* of a subsequence p in S is defined as the count of p divided by the number of items that have already arrived in the stream. A subsequence of S is a *sequential pattern* when its support is above a given threshold. In short, a sequential pattern in S is a frequent subsequence in S.

Following the A priori principle, given two subsequences $p = \{p_1, p_2, ..., p_n\}$ and $p' = p \backslash \{p_n\}$, it holds that $supp(p') \geq supp(p)$ due to the anti-monotonicity property. Thus, if p is a sequential pattern, p' is also a sequential pattern. An association rule is an implication of the form $p_1, p_2, ...p_{n-1} \Rightarrow p_n$. The confidence of an association rule is defined as $conf(p_1, ..., p_{n-1} \Rightarrow p_n\} = \frac{supp(p)}{supp(p')}$.

Given multiple streams $S_1, S_2, ..., S_n$, sequential patterns can be generated with either of the streams, or as a mixture of multiple streams. Sequential patterns that only contain items of one specific stream are called intra-stream sequential patterns. Sequential patterns that contain items of different streams are called inter-stream sequential patterns [HS11]. For inter-stream sequential patterns, the rules above have to be adapted. The sequence must not be a subset of one set but a combination of subsets from different sets.

To provide different views on the data, three different window concepts are used by the *StrPMiner*. The algorithm works with the Landmark Window, the Sliding Window and the Damped Window concept. In the Landmark Window, a point in time is defined as the *landmark*. All data is then collected starting from the *landmark*. This concept allows to look at big parts of the data. The Sliding Window concept uses a fixed window size and slides it over the data. Thus, only a snapshot of the data will be monitored at any given time. An advantage is that old patterns will be forgotten eventually, which leaves only current information. The Damped Window uses a similar concept as the Landmark Window. In contrast to the Landmark Window, the Damped Window uses weights to reflect the age of an object. New items will be more important than old ones. This allows a compromise between the Landmark Window and the Sliding Window concept.

3.1 The Batch Approach and its Problems

A good solution to find sequential patterns in a streaming environment is the batch approach. It allows to use the Apriori principle, since each batch provides a static data set. However it also leaves a room for errors.

Given a support threshold of 2, meaning a pattern has to appear two times within one batch

to be counted as frequent, a batch size of 3 and following sequence:

$$(A, B, C, A, C, C, A, D, C, A...) \tag{1}$$

with A, B, C, D being items of a stream. The online component would cut the data stream in following batches:

1. (A, B, C) 2. (A, C, C) 3. (A, D, C) 4. (A, ...) 5. ...

In this case, no pattern would be frequent. Looking at the whole data without cutting it into batches would reveal that the pattern C, A appears three times, which is over the support threshold of 2. Thus, would lead to a frequent pattern. Additionally, all items except for C in the second batch, would be pruned away, although the item A and C appear in every batch. This leads to two reasons for errors through the batch approach:

1. Patterns that appear between batches will not be found.
2. Items and patterns that do not appear often in one batch will be pruned, although they are frequent in the whole data set.

The *StrPMiner* was designed to avoid the batch approach because of these two reasons which result into false statistics for sequential patterns.

4 The StrPMiner

Since the *PrefixSpan* algorithm only scales well when the candidates for sequential patterns can be pruned, the *StrPMiner* reverses the idea of the *PrefixSpan* and uses a new algorithm called the Pattern Builder (*PBuilder*). This allows the *StrPMiner* to work on each data item step by step as it comes in.

For the given data and application the order of the items is important. To provide this focus, the definition of sequential patterns had to be changed slightly. As stated previously, a sequential pattern is a frequent subsequence. A subsequence is, in short, an ordered list of elements taken from the database. Yet for the e-cosmos project, the definition of a subsequence had to be stricter. We redefine a subsequence, and also a sequential pattern, as only allowed to be a list of ordered items which follow each other directly: p is considered a subsequence of q if $p = (p_1, p_2, ..., p_n)$, $q = (q_1, q_2, ..., q_m)$ and there exist integers $i_i < i_2 < ... < i_m$ such that $p_1 = q_{i_1}, p_2 = q_{i_2}, ..., p_m = q_{i_m}$ *and* for all k, l with $l, k < m$ and $l = k + 1$ there exists no item q_j with $k < j < l$. This restriction was made, since the focus of the patterns should lie in the order of the items.

The *StrPMiner* handles arriving data from multiple streams at once, compresses the data and passes it to the *PBuilder*. The *PBuilder* then uses this data to create sequential pattern candidates. After this, the *StrPMiner* saves the candidates in the T_0 tree structure and keeps track of those candidates and their corresponding statistics. Currently those are the count, support and confidence value. This approach allows full accuracy, and flexibility

in the output, as the support threshold can be changed at every output request. This is not possible when using the *PrefixSpan*, as the threshold has to be set before calculation starts. Additionally the *StrPMiner* is able to jump at any point in time of the calculation process.

4.1 The *PBuilder*

The *StrPMiner* assumes that at each point in time, only one item can arrive. If more than one item arrives at one point in time, the algorithm orders them and calculates each step for one item after another. This allows the *StrPMiner* to minimize the amount of calculation that has to be done with each new arriving item by only updating the statistics that are affected by the new item.

Following this concept, the *PBuilder* is only creating patterns that contain the newly arrived item. Additionally, since it is the last arrived item, all created patterns will end with this item. Given an item A as the newly arrived item, the *PBuilder* starts with this item as the first pattern, the postfix. After this, the algorithm recursively adds older items as a prefix to the previously created postfix. To ensure that the *StrPMiner* only finds direct sequential patterns, the prefix is a direct predecessor of the postfix Figure 4 visualizes the *PBuilder*. Also Algorithm 1 shows a peseudocode of the *PBuilder*

The *PBuilder* saves the relevant items in a list. The list contains all relevant items, without differing between the streams. With this concept, the *PBuilder* can create all inter- and intra-stream patterns without any specifications.

For each created pattern, the *PBuilder* algorithm calls the update function of the T_0 tree.

PBuilder
Data: DataSnapshot, currentPattern
//DataSnapshot contains the latest compressed items and is limited by maxRuleLength. The newly arrived item is at the last position
Result: The new patterns that can be created with the new item
int index = DataSnapshot.length;
//create patterns until maxRuleLength is reached
while *currentPattern.length* $\leq$ *DataSnapshot.length* **do**
    //add the next item to the pattern
    currentPattern = DataSnapshot.get(index-currentPattern.length) + currentPattern;
    //update the tree with the new pattern
    updateTree(currentPattern);
end

Algorithm 1: The *PBuilder* explained with pseudo code

4.2 Maximum Pattern Length as a Solution for Exponential Growth

In contrast to a static database, where all information are available from the beginning, the streaming approach does not have any information on how future items and their frequency might look like. This means that any item and pattern that is currently not frequent in a stream, can become frequent at any later point in time. The support of every pattern changes with every new arriving item. To ensure that at every time the user requests an output all sequential patterns are part of the output, every possible pattern and its information have to be saved. This causes an exponential increase of the calculation time, as with every new arriving item more patterns can be created. Additionally, the memory space will eventually collapse, as the amount of data that has to be saved also increases exponentially.

To stop the exponential growth, the *StrPMiner* introduces a parameter called *maxPatternLength*, as an upper Bound for the pattern length. This variable restricts the *PBuilder* to only look at the last *maxPatternLength* items. A *maxPatternLength* of five, will cause patterns to maximally contain five items, as only those are relevant to the algorithm. Given this bound, the calculation time in each step only scales with the size of the *maxPatternLength* parameter. Additionally this parameter bounds the maximum growth of the required memory space. On the one hand, as the parameter will not change over the time, the calculation time for each new arriving item will be constant. On the other hand this upper bound filters patterns, before they have been created. Sequential patterns that have a length higher than the given bound, will not be found. With this in mind, a careful selection of the upper bound is important, as it provides a trade off between the calculation time and accuracy.

4.3 Different Window Models

As previously mentioned, the *StrPMiner* uses the T_0 tree introduced by [MDH08]. For the algorithm slight adaptations were made, regarding the saved information. The *StrPMiner* saves the label of the item and the appearances of the pattern in each node. The count of each pattern is then determined by the number of time stamps saved in the corresponding node. An example is shown in Figure 5.

The sliding window model helps to provide another view on the data, as it only contains knowledge of recent data and forgets old data. This helps in cases, where the data changes drastically over the duration of the stream. The landmark window would still show old patterns even though they did not reappear for a long time. In general the whole algorithm works the same, as in the landmark window, except for an extra pruning step. For this the time stamp of the corresponding item and the patterns created with it have to be deleted from the T_0 tree, which is one path.

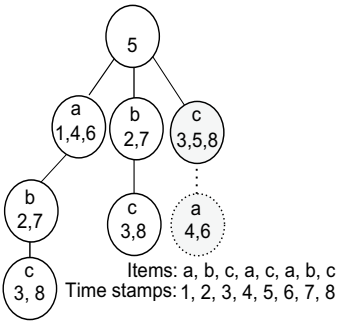
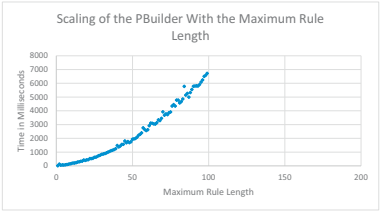
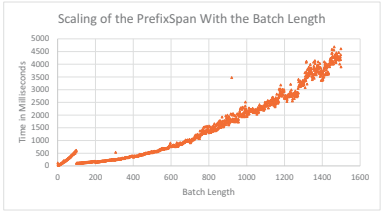


Figure 3: An example of the T_0 tree. The doted node represents the pattern (c,a).



(a) The *PBuilder* scales with the Maximum rule length parameter



(b) The *PrefixSpan* scales with the batch length

Figure 4:

5 Experimental Results

Because of the two reasons previously described, the *StrPMiner* uses a different algorithm to find sequential patterns, than the *PrefixSpan*. The *PBuilder* can handle each newly arriving item without using the batch approach.

This results in an algorithm that uses more memory space, as it has to save all patterns. While the calculation time of the *PrefixSpan* scales with the set support threshold and the batch length, the calculation time of the *PBuilder* is only dependent on the maximum pattern length.

For this experiment the *PrefixSpan* has been adapted, so that it will only create direct sequential patterns. Additionally this improves the run time of the *PrefixSpan*, as it will calculate less patterns.

As seen in Figure 6 the calculation time of both algorithms increases exponentially with the increase of their corresponding parameters. The scaling of the *PBuilder* may be worse, but in practical use this is no real problem. Rarely sequential patterns above a threshold of 1% have a length of more than 20. In the given translation data set, there are no patterns

with a length of more than 18, that have support over 1%. With the assumption that no sequential pattern has a length of more than 20, the *PBuilder* provides a 100% accuracy with a maximum pattern length parameter of 20.

	Datasize	Batchlength 50		Batchlength 250		Batchlength 1000	
Subject		max. error	avg. error	max. err	avg. err	max. err	avg. err
GT1	3885	57/498	35/545	50/406	23/545	89/406	27/545
GT2	3926	59/194	30/371	65/222	38/371	132/233	42/371
GT3	3913	53/455	30/473	47/298	27/473	78/455	17/473
GT4	5503	101/322	68/814	51/273	42/814	75/369	33/814

Table 1: The table shows the error rate on the count value over the Top 20 patterns with the highest support value, generated by the *PrefixSpan* and the *PBuilder*. The support threshold was set to 0.05. The notation x/y reads as, y being the number of patterns found by the *PBuilder* and x the difference between y and the number of patterns found by the *PrefixSpan*.

Table 1 and Table 2 show the error rate of the *PrefixSpan*. The 20 patterns with the highest support value of each algorithm were counted and compared. The ranking of the patterns only has minor differences. The accuracy of the *PrefixSpan* should increase with the size of the batch length less patterns can be lost between two batches. Nevertheless, Table 1 shows, that the batch length may be selected unluckily, so that some sequential patterns will be pruned because they are part of different batches. This may also provide mistakes in the count value of items, as an item may not be frequent in one batch, although it is frequent overall.

	Datasize	Batchlength 50		Batchlength 250		Batchlength 1000	
Subject		max. error	avg. error	max. err	avg. err	max. err	avg. err
GT1	3885	39/367	21/545	13/211	5/545	10/211	2/545
GT2	3926	17/194	6/371	5/212	2/371	1/395	0.1/371
GT3	3913	28/361	14/473	8/361	3/473	13/213	2/473
GT4	5503	90/564	47/814	29/564	15/814	14/273	5/814

Table 2: Error rate as in Table 1. Here the support threshold is set to 0.01.

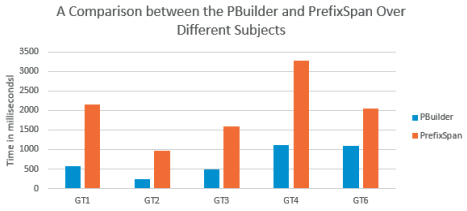


Figure 5: A visualization of the calculation time of both algorithms on different data sets. The maximum rule length is set to 20 and the batch length to 1000. The support threshold is 0.01.

Table 2 shows, that with the proper settings, the errors made by the *PrefixSpan* are neglectable. But using those settings increases the calculation time to such extent, that the *PBuilder* is much faster. This is visualized in Figure 7.

6 Conclusion and Future Directions

The *PBuilder* is an algorithm to mine sequential patterns out of a streaming data environment. Experiments have shown, that the calculation time of the *PBuilder* may be slower than the calculation time of the *PrefixSpan*, but it is still good enough. Additionally the *PBuilder* has a higher accuracy rate, which shows, that the *PBuilder* provides a good alternative to the *PrefixSpan*. The *StrPMiner* uses the *PBuilder*, which allows it to be more flexible than most current algorithms.

Fixations made by the eyes have a temporal extension. This means they start at a specific point in time and end at another, later, point in time. Those so called interval-based events and their temporal relations can not be found by the *StrPMiner*. In the future, we will upgrade the *StrPMiner* to the Interval Streaming Pattern Miner (*IStrPMiner*). The *IStrPMiner* will be able to find characteristics in the temporal relationships between different items. Those patterns are of the form: A is overlapping with B, or A happens during B.

Acknowledgments

Funded by the Excellence Initiative of the German federal and state governments.

References

[CCPL14] Yi-Cheng Chen, Chien-Chih Chen, Wen-Chih Peng, and Wang-Chien Lee. Mining Correlation Patterns among Appliances in Smart Home Environment. In *Advances in Knowledge Discovery and Data Mining*, pages 222–233. Springer, 2014.

[GHP+03] Chris Giannella, Jiawei Han, Jian Pei, Xifeng Yan, and Philip S Yu. Mining frequent patterns in data streams at multiple time granularities. *Next generation data mining*, 212:191–212, 2003.

[HS11] Marwan Hassani and Thomas Seidl. Towards a mobile health context prediction: Sequential pattern mining in multiple streams. In *MDM, 2011*, pages 55–57. IEEE, 2011.

[LL09] Hua-Fu Li and Suh-Yin Lee. Mining frequent itemsets over data streams using efficient window sliding techniques. *Expert Systems with Applications*, 36(2):1466–1477, 2009.

[MDH08] Luiz F Mendes, Bolin Ding, and Jiawei Han. Stream sequential pattern mining with precise error bounds. In *ICDM.*, pages 941–946. IEEE, 2008.

[PHMA+01] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *ICDE*, pages 0215–0215. IEEE Computer Society, 2001.

[SEM11] Amany F Soliman, Gamal A Ebrahim, and Hoda K Mohammed. SPEDS: A framework for mining sequential patterns in evolving data streams. In *Communications, Computers and Signal Processing (PacRim), 2011*, pages 464–469. IEEE, 2011.

[WC07] Shin-Yi Wu and Yen-Liang Chen. Mining nonambiguous temporal patterns for interval-based events. *KDE*, pages 742–758, 2007.