Kaal – a Real Time Stream Mining Algorithm

Rajanish Dass
Indian Institute of Management, Ahmedabad
rajanish@iimahd.ernet.in

Varun Kumar
Yahoo! India R&D
varunk@yahoo-inc.com

Abstract

Finding frequent patterns in a data stream has been one of the daunting tasks since its inception. Mining data streams are allowed only one look at the data, and techniques have to keep pace with the arrival of new data. Furthermore, dynamic data streams pose new challenges, because their underlying distribution might be changing. Most importantly, the stream mining algorithm must be fast enough to adapt itself to slow as well as very fast data streams. In this paper, we have introduced a new stream mining algorithm called Kaal – Sanskrit word for time – that is significantly better than existing classical algorithms. Further, Kaal is capable of adapting well to variable batch sizes. The batches are decided by a fixed time quanta, any number of transactions coming in that time interval constitutes that batch. Previous stream mining algorithms demand fixed batch sizes, which in real world scenario becomes difficult to realize or fail to provide periodic real-time results.

1. Introduction

In last few last decades, data mining techniques have become the key tool to analyze and understand the data and the underlying information it holds. Typical data mining tasks include association mining, classification and clustering. These techniques help find interesting patterns, regularities and anomalies in the data. However traditional data mining techniques can not directly apply to the data streams. Finding frequent patterns in a data stream has been one of the daunting tasks since its inception. Frequent pattern mining in static data base has been extensively researched and many implementations are done to squeeze down the time requirements. Still the traditional static data mining tools cannot be directly applied to data streams. There are two difficulties in using an Apriori-like algorithm in a data stream environment. Frequent itemset mining by Apriori is essentially a set of join operations as shown in [1]. However, join is a typical blocking operator [1-3] which means that output comes in bursts and not continuously, which cannot be performed over stream data since one can only observe at any moment a very limited size window of a data stream. Mining data streams are allowed only one look at the data, and techniques have to keep pace with the arrival of new data. Furthermore, dynamic data streams pose new challenges, because their underlying distribution might be changing [4, 5].

For data stream applications, the volume of data is usually too huge to be stored on permanent devices or to be scanned thoroughly more than once. Summarizing, it is necessary to store information related to frequent items, as well as those of infrequent ones. If the currently infrequent items were not stored and these items become frequent later, it would be impossible to figure out their correct overall support [6], however, it will be unrealistic to hold all streaming data in the limited main memory. Lastly the stream mining algorithm must be fast enough to adept itself to slow as well as very fast data streams. This paper efficiently addresses all the above problems. Kaal is significantly faster than existing classical algorithms [7]. Further, Kaal is capable of adapting well to variable batch sizes which is not accounted in most recent works also [17] [18]. The batches are decided by a fixed time quanta, any number of transactions coming in that time interval constitutes that batch. Previous stream mining algorithms demand fixed batch sizes, which in real world scenario becomes difficult to realize or fail to provide periodic real-time results.

1.1. Real World Applications of Stream Data Mining

Some motivating examples for stream data mining are: Transactional records of a store, across the globe or a geographic region. The sales researchers will be interested to know which items are frequently bought
together, which item set were bought maximum number of times during a certain historical period and several such queries.

A typical ISP (internet service provider) would be interested in finding data like how much traffic went on a link in a day from a given set of IP addresses, how much of traffic between two routers was similar and what were the variations of data traffic in a day. The above queries are very useful in rerouting users to backup servers if the primary servers are overloaded and also finding denial service attacks. All these questions can be well addressed by stream mining.

There are several emerging applications in sensor monitoring where a large number of sensors are distributed in the physical world which generate streams of data that need to be combined, monitored and analyzed. Terrestrial, atmospheric and ocean surface data collected by satellites is huge and mining such data gives enormous information about weather conditions, which helps in many ways.

Stock market analysis requires a typical real-time framework for providing healthy decision. The stock indices constitute a huge data stream which requires instant processing keeping in view the historic data. A very fast mining algorithm can suffice such needs. Further the fast growth rate of the algorithm (Figure [14]) provides us with a good picture of the frequent patterns very quickly.

Another fastest evolving application of stream mining is in the area of security. It ranges from network security, e-transaction security (eg. ATMs), counter terrorism etc. Security applications also require very fast real-time processing. An outlier pattern discovered long after the threat will be of no good use.

2. Previous Work

There has been several works on stream data mining. Most of the other algorithms in stream mining only talk about mining frequent itemsets in recent data. Many of them [8-11] fail if we ask for knowledge about frequent itemsets in the stream at a past time or in a particular interval of time. Algorithms like [7, 9, 12, 13] process the transactions in batches or buckets of fixed size. In any real world stream mining application, fixing batch size is not realistic. If we only process transactions batch wise, then we are restricting stream to only have fixed multiples of transactions in fixed intervals of time and it is not going to be the scenario, as the algorithm has no control over the stream, and number of transactions in a fixed interval of time can not exactly be the multiples some fixed batch size. So, to process transactions in batch wise, the batch size should not be fixed as it is more realistic.

Previous work [9] studied the landmark model, which mines frequent patterns in data streams by assuming that patterns are studied from the start of the stream up to the current moment. The landmark model stays undesirable since the set of frequent patterns usually are time sensitive and in many cases, changes of patterns and their trends are more interesting than patterns themselves [7].

3. Problem Definition

Find the complete set of frequent patterns, and their trend in a data stream, assuming that one can only see the set of transactions in a limited size window at any moment.

3.1. Definitions

Depending upon the time granularity required a time slice is fixed and data collected in one time slice is grouped as a batch B.

- B is the block of data collected over a time period T
- Frequency of an Itemset I is the number of transactions in B in which I occurs
- Support = frequency/(number of transactions in B)
- \( \sigma = \) Minimum support
- \( \epsilon = \) Maximum support error (usually taken to be 10% of support)

Thus, the algorithm divides patterns into three categories :-

- An Itemset is frequent if \( \text{support} \geq \sigma \)
- An Itemset is sub-frequent if \( \epsilon \leq \text{support} \leq \sigma \)
- Otherwise its infrequent

3.2 Auxiliaries of the Algorithm

3.2.1 Main Data Structure Used

The summary data structure, as shown in Figure [1], used for maintaining the frequency of the frequent itemsets over the time is similar to that of the FP-stream [7]. This is an extension of a trie structure, with the logarithmic tilted-time window embedded in each node. An additional column is added to maintain the size of the batch. It helps the algorithm adjust to varying data stream rates.

The embedded array structure has three columns (implementation wise just two) tilted-time window number, support (frequency), and batch size. The support of corresponding tilt-window is the grouped frequencies according to the logarithmic scale. The batch size is the number of transactions in the batch (to deal with variable batch size).
3.2.2 Logarithmic Merging

*Kaal* uses logarithmic time merging as suggested by[8]. In Logarithmic time merging we accumulate the results in fixed time quanta, then as time passes we merge the older results averaging them. Thus we allot more storage space to the recent results and the less to the older ones, the beauty of the approach is that we store very historic data in small memory.

The inspiration for this approach is that we are often interested in recent changes at a fine granularity, but long term changes at a coarse granularity. Illustrating in an example, we collect data every 15 minutes (one quarter). All the transactions in one time quanta are treated as one batch. Suppose that ‘n’ batches have arrives, i.e n*15 minutes have passed. f(x,y) is one storage space for data summary from batch x to batch y.

For 8 batches of data we maintain:

<table>
<thead>
<tr>
<th>Level</th>
<th>Storage Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>f(n,n)</td>
</tr>
<tr>
<td>1</td>
<td>f(n-1,n-1)</td>
</tr>
<tr>
<td>2</td>
<td>f(n-2,n-3)</td>
</tr>
<tr>
<td>3</td>
<td>f(n-4, n-7)</td>
</tr>
</tbody>
</table>

The above flow says that the newest batch ‘n’ is given 1 row in the table f(n,n), the second last batch ‘n-1’ is also given one row, but the batches ‘n-2’ and ‘n-3’ are averaged and given one storage row. Similarly the further 4 batches ‘n-4’th to ‘n-7’th are merged.

The Window needs 4 units for current 4 quarter, 1 unit for past 4 quarter, 1 unit for past 8 quarter, 1 unit for past 16 quarter, as shown in Figure [2]. Thus for a year’s data 4×24 ×365 quarters Which needs 2N = 4×24 ×365 quarters => No. of units required = N = log2(4×24 ×365 ) = 17 Thus the summery structure needs just 17 frames (or memory rows in the tilt window table) to store a data of 1 year, with time frame of 15 minutes. With a maximum support error as suggested by[7].

3.2.3 Pruning Techniques

The Tilted window table for each node(pattern) can be pruned using the following methods:

**Tail Pruning:** We maintain only fl(t0), fl(t1), fl(t2)...fl(tm-1) and drop the tail sequence fl(tm), fl(tm+1)...fl(tn). We drop this tail sequence when the following condition holds true:

- There exists an l, for 1 ≤ i ≤ n, fl(ti) < σ|Bi| and
- There exists an l’, for 1 ≤ m ≤ l’ ≤ n, Σl’i=1 fl(ti) < ε Σi=1 |Bi|

(The above condition hints that we are taking frequent as well as sub-frequent (ε ≤ support ≤ σ) itemsets.) as suggested by [7].

**Type I Pruning:** If fl(B) < ε|B| then none of superset need to be examined. So the mining of B can prune its search and not visit superset of I. (Inspiration from Apriori).

**Type II Pruning:** If all the I’s tilted-time window table entries are pruned then any superset will also be dropped. As suggested by [7].

For Heuristics model of kaal, refer [14] for step I. Taking an example to show how *Kaal* works:

Taking a data set collected over a period of time, the stream is grouped into batches each collected over time T. (The batches collected can be seen to be of different sizes as the stream rate varies)

4. Algorithm Kaal

<table>
<thead>
<tr>
<th>Steps</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize the allowable execution time</td>
<td></td>
</tr>
<tr>
<td>Initialize the support for the first phase to be ε. Initialize the summary trie to empty.</td>
<td></td>
</tr>
<tr>
<td>Collect the transactions over a defined period of time(over the lowest time slice of Tilted-time window). We call this set of transitions a batch B.</td>
<td></td>
</tr>
</tbody>
</table>

**Step 1. Processing current Batch**

Let the initial search frontier contain all 3-length candidate patterns. Let this search frontier be stored as a global pool of candidate patterns. Initialize a set called Border Set to null. Order the candidate patterns of the global pool according to their decreasing length(resolve ties arbitrarily). Take a group of most promising candidate patterns and put them in a block b of predefined size.

**Expand (b)**

Expand (b: block of candidate patterns)
If not last_level
then
begin
Expand1(b)
end

**Expand1(b):**

1. Count support for each candidate pattern in the block b by intersecting the diff-set list of the items in the database.
2. When a pattern becomes frequent, remove it from the block b and put it in the list of frequent patterns along with its support value. If the pattern is present in the Border Set increase its sub itemset counter. If the sub itemset counter of the pattern in Border Set is equal to its length move it to the global pool of candidate patterns.
3. Prune all patterns whose support values less than the calculated support. Remove all supersets of these patterns from Border Set.
4. Generate all patterns of next higher length from
the newly obtained frequent patterns at step 3. If all immediate subsets of the newly generated pattern are frequent then put the pattern in the global pool of candidate patterns else put it in the Border Set if the pattern length is > 3.

5. Take a block of most promising b candidate patterns from the global pool.

6. If block b is empty and no more candidate patterns left, output frequent patterns and exit.

7. Call Expand (b) if enough time is left in to expand a new block of patterns, else obtain the frequent patterns and return.

**Step 2. Update the summary structure:**

Mine itemsets out of the current trie using the following steps:

For each itemset I, check if I is in the summary trie structure

If I is in summary trie add fI(B) to the tilted-time window table of I, conduct tail pruning. If table is empty, then stop mining supersets of I (Type II pruning) If not empty then continue mining the super sets of I.

If I is not present in structure and if fI(B) ≥є|B|
then insert I into the FP-Stream. Otherwise, stops mining supersets of I (Type I pruning)

**Step 3. Scan the summary trie (DFS):**

For each itemset I, check if I was updated

If not then insert 0 in I’s tilt window
Tail prune I’s table
For the leaf, if tilt-window table is empty then drop the leaf. Search the sibling of the leaf
Returning to parent node check if may be dropped.
Collect new batches and process them similarly.

<table>
<thead>
<tr>
<th>Table 1. The algorithm kaal</th>
</tr>
</thead>
</table>

**Step I:** Collect the transactions over a defined period of time (that which will be the lowest time slice of Tilted-time window). The set of transitions is shown as batch B1.

**Step II:** Construct a two dimensional lower triangular matrix M using the procedure Create_matrix explained in Figure [3] from [15].

**Step III:** Let the absolute support for batch processing be 2, corresponding to c. Cells of the Matrix M are visited to find F(1) and F(2) [where F(n) is the set of frequent pattern of length n]

F(1) = {a(9), b(7), c(7), d(11), e(9)}……(1)

F(2) = {ab(5), ac(6), ad(8), ae(6), bc(4), bd(7), be(4), cd(6), ce(6), de(8)}……(2)

Matrix M is shown in Figure [4].

**Step IV** Two 2-length patterns are merged if their first elements match. Thus

Newly merged patterns = {abc, abd, abe, acd, ace, ade, bcd, bce, bde}…………(3)

**Step V**. Find if all the subsets of new merged patterns are frequent. If all its 2-length subsets are not present, then the pattern is pruned (using the support monotonicity property [3], else the pattern becomes a candidate-pattern and it is moved to the global-pool of candidate patterns C( ). The global-pool of candidate patterns is sorted on length and any tie between two same length patterns is resolved arbitrarily.

C( ) = {abc, abd, abe, acd, ace, ade, bcd, bce, bde, cde}………(4)

**Step VI** Block size is chosen according to the density of the current data batch and the free memory available. (Currently we have implemented using only by the batch density). Batch density is estimated as:

\[
\text{Density} = \frac{\text{Number of 2 length patterns in the current batch}}{\sum f(a,b)}
\]

where \(\sum f(a,b)\) denotes the maximum possible pairs for the given data set, i.e if every transaction has all the items. The computation is done as \(nC2\), where \(n\) is the number of unique items.

Batch size of one means that it is effectively a depth first search and large block size b is equivalent to breadth first search. If the density of the data in the batch is high we try to go for breath first search.

In the sample dataset we take the block size to be 5 treating the density as average.

This means as the 3-length candidate patterns are pushed into the global pool, 5 of these patterns namely, abc, abd, abe, acd and ade will be put in the next block b.
Step VII From the diff-sets of the two-length patterns we calculate the diff-sets of the three length patterns as follows:
If \(d(ab)\) and \(d(ac)\) represents the diff-set of \(ab\) and \(ac\) respectively, then we can get \(d(abc) = d(ac) - d(ab)\) [as suggested by Zaki (Zaki 2004)] and the frequency of the pattern \(abc\) can be found from \(freq(abc) = freq(ab) - |d(abc)|\).

We now check the frequency of these patterns by intersecting the diff-set tid-lists of the items.
\[b = \{abc (3), abd (5), axe (2), acd(5), ace(5)\} \ldots (5)\]
Here none of the frequency is below the support threshold so none gets pruned.
\[F(3) = \{abc (3), abd (5), axe (2), acd(5), ace(5)\} \ldots (6)\]

Step VIII We now merge the newly found frequent patterns in \(F(3)\) and test these newly merged patterns generated for the presence of their immediate subsets. Newly merged patterns = \{\(abed, abde, ace\}\} \ldots (7)
All immediate subsets of the pattern \(abcdef\) are not present in \(F(3)\). Hence we move the pattern \(abcdef\) to border set of length 4, \(BS(4)\), with a sub-itemset counter of 4.
\[BS(4) = \{abcdef\text{(sub-itemset = 3)}, abcede\text{(sub-itemset = 3)}, abde\text{(sub-itemset = 2)}, acde\text{(sub-itemset = 2)}\} \]
\…(8)
Patterns ade, bcd, bce, bde, cde are taken in the next block \(b\) from the global-pool of candidate patterns.
\[b = \{ade(5), bcd(4), bce(3), bde(4), cde(5)\}\ldots (9)\]
All these items have frequency greater than support greater than 2 and are hence frequent.
Thus from the new block \(F(3) = \{ade(5), bcd(4), bce(3), bde(4), cde(5)\}\ldots (10)
For each pattern in the current \(F(3)\), search \(BS(4)\) to see if any of the immediate supersets are waiting in the border set. Pattern \(abcdef\) is in \(BS(4)\) with sub-itemset counter = 3.

Hence increase the sub-itemset counter of \(abcdef\) and make it 4. The patterns \(abde, abce\) and \(acde\) also increase the sub-itemset count to 4. The pattern \(abcdef, abde, acde\) is of the highest length among the candidate patterns in the global-pool and is put in the next block \(b\).

Merge newly found klength frequent patterns with previously found k-length frequent patterns to make patterns of higher length. Newly merged patterns \(4 = \{bcde\}\ldots (11)\)
The patterns \(abcdef, abce, abde, acde\) go to the current block \(b\). After intersecting the diff-tid-list of these patterns,
\[F(4) = \{abcd(3), abce(2), abde(2), acde(4), bode(3)\}\ldots (12)\]

Step IX Construct a trie as illustrated in Figure [5] of the obtained frequencies \(F(1), F(2)\ldots F(5)\)

Similarly search the BS (5) with newly found F(4) patterns and merge the patterns in the newly found F(4)’s with themselves and also with previous F(3)’s to generate higher length patterns.
As no higher length patterns can be generated and the number of patterns in block \(b\) becomes zero and also the number of candidate patterns in the global pool of candidate patterns becomes zero, the algorithm stops executing here. Thus, the set of all frequent patterns are:
\[F(1) = \{a(9), b(7), c(7), d(11), e(9)\}\]
\[F(2) = \{ab(5), ac(6), ad(8), ae(6), bc(4), bd(7), be(4), cd(6), ce(6), de(8)\}\]
\[F(3) = \{abc(3), abd(5), abe(2), acd(5), ace(5), ade(5), bcd(4), bce(3), bde(4), cde(5)\}\]
\[F(4) = \{abcd(3), abce(2), abde(2), acde(4), bcde(3)\}\]
\[F(5) = \{abcde(2)\}\]

Whenever it is stopped before its natural completion, i.e the execution time expires, it outputs frequent patterns of various lengths it had obtained up to that point of execution time.

Step X Update the summary trie structure by the current trie. Since it’s the first structure summary trie is empty. Thus we mine the items out of the current trie in depth first manner. Following first item mined is “a” its not present in summary trie so it is inserted with a value 9 in the support column and 12 in the batch size column. The updated node for pattern a is shown in Figure [6].
The tail pruning condition is checked \( f(t_i) \) is not less than \( \epsilon |B_i| \), Thus no tail pruning.

As the table is not empty thus the supersets of “a” is mined. (No type II pruning)

Then following it all elements of \( F(1), F(2) \ldots F(5) \) are inserted. And a structure similar to the current trie is obtained.

**Step XI:** Now new batch is collected and the batch is processed through step II to Step IX

For further batches similar processing is to be done through step II to Step IX

Taking example of a particular pattern “acd”

**After batch 2**

freq(“acd”) = 2 as its already present in the structure its new value is inserted in the time window.

| Table 3. Embedded array for node “acd” after batch 2 |
|-----------------|---------|----------|
| t-win | Support | Batch-size |
| t0   | 2       | 10       |
| t1   | 5       | 12       |

The tail pruning condition is checked \( f(t_i) \) is not less than \( \epsilon |B_i| \).

Thus no tail pruning.

As the table is not empty thus the supersets of “acd” is mined.

**After batch 3**

freq(“acd”) = 3 as its already present in the structure its new value is inserted in the time window.

\( f(2,2) \) is shifted to next frame and \( f(3,3) \) is collected in new frame.

\( f(3,3) \) \( f(2,2)[f(1,1)] \)

| Table 4. Embedded array for node “acd” after batch 3 |
|-----------------|---------|----------|
| t-win | support | Batch-size |
| t0   | 3       | 12       |
| t1   | 5+2=7   | 12+10=22 |

The tail pruning condition is checked \( f(t_i) \) is not less than \( \epsilon |B_i| \).

Thus no tail pruning.

As the table is not empty thus the supersets of “acd” is mined.

**After batch 4**

freq(“acd”) = 4 as its already present in the structure its new value is inserted in the time window.

\( f(2,1) \) is shifted to next frame. \( f(3,3) \) is shifted to next frame and \( f(3,3) \) is collected in new frame.

\( f(4,4) \) \( f(3,3) \) \( f(2,1) \)

| Table 5. Embedded array for node “acd” after batch 4 |
|-----------------|---------|----------|
| t-win | support | Batch-size |
| T0   | 4       | 12       |
| T1   | 3       | 12       |
| T2   | 7       | 22       |

The tail pruning condition is checked \( f(t_i) \) is not less than \( \epsilon |B_i| \).

Thus no tail pruning.

As the table is not empty thus the supersets of “acd” is mined.

**After batch 5**

freq(“acd”) = 1 and was not included in the current trie structure. However in the update stage of summary trie, a value 0 is inserted into the table as the pattern was not updated in this cycle.

\( f(5,5) \) \( f(4,4)[f(3,3)] \) \( f(2,1) \)

| Table 6. Embedded array for node “acd” after batch 5 |
|-----------------|---------|----------|
| t-win | support | Batch-size |
| t0   | 0       | 12       |
| t1   | 3+4    | 12+12    |
| t2   | 7       | 22       |

The tail pruning condition is checked \( f(t_i) \) is not less than \( \epsilon |B_i| \).

Thus no tail pruning.

As the table is not empty thus the supersets of “acd” is mined.

**After batch 6**

freq(“acd”) = 2 as its already present in the structure its new value is inserted in the time window.

\( f(2,1) \) is shifted to next frame. \( f(3,3) \) is shifted to next frame and \( f(3,3) \) is collected in new frame.

\( f(6,6) \) \( f(5,5) \) \( f(4,3)[f(2,1)] \)

| Table 7. Embedded array for node “acd” after batch 6 |
|-----------------|---------|----------|
| t-win | support | Batch-size |
| t0   | 2       | 9        |
| t1   | 0       | 12       |
| t2   | 14      | 46       |
The tail pruning condition is checked if \( t_i \) is not less than \( \epsilon |B_i| \). Thus no tail pruning.

As the table is not empty thus the supersets of “acd” is mined.

After batch 7:
freq(acd) = 3 as its already present in the structure its new value is inserted in the time window.
\( f(2,1) \) is shifted to next frame. \( f(3,3) \) is shifted to next frame and \( f(3,3) \) is collected in new frame.
\( f(7,7) \) \( f(6,6) \) \( f(5,5) \) \( f(4,3) \) \( f(2,1) \)

The tail pruning condition is checked if \( t_i \) is not less than \( \epsilon |B_i| \). Thus no tail pruning.
As the table is not empty thus the supersets of “acd” is mined.

After batch 8:
freq(acd) = 6 as its already present in the structure its new value is inserted in the time window.
\( f(2,1) \) is shifted to next frame. \( f(3,3) \) is shifted to next frame and \( f(3,3) \) is collected in new frame.

\( f(8,8) \) \( f(7,7) \) \( f(6,5) \) \( f(4,1) \)

### Table 8. Embedded array for node “acd” after batch 7

<table>
<thead>
<tr>
<th>t-win</th>
<th>support</th>
<th>Batch-size</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>t1</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>t2</td>
<td>14</td>
<td>46</td>
</tr>
</tbody>
</table>

### Table 9. Embedded array for node “acd” after batch 8

<table>
<thead>
<tr>
<th>t-win</th>
<th>support</th>
<th>Batch-size</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>t1</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>t2</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>t3</td>
<td>14</td>
<td>46</td>
</tr>
</tbody>
</table>

The tail pruning condition is checked if \( t_i \) is not less than \( \epsilon |B_i| \). Thus no tail pruning.
As the table is not empty thus the supersets of “acd” is mined.

Similarly the table for other patterns is also generated. Checking at each step if the table is empty then it supersets are not mined (Type II pruning).

### 5. Empirical Findings

All the experiments are performed on 60 batches of average 50K per batch of average transaction length of 3, 7 and 10. The batches are generated by IBM synthetic data generator. Instead of dividing a T7I4D3M into 60 batches we have used 60 batches of T7I4D50K with small change in parameters to incorporate a variation in batches. All experiments were performed on 2GB RAM, 2Ghz Processor machine, Fedora 9 core.

Kaal takes almost equal time of (1.15 sec/batch) for all batches, thus the algorithm stabilizes and fits well with large stream (Figure [7]). Kaal stream mining algorithm can work for data rate of approximately (50,000 transactions /1.15 sec. * 1 sec) = 43500 Transactions/sec while FP-stream can handle (50,000 transaction /2.1 sec * 1 sec) = 24000 Transactions/sec. (of average length 7).

The faster processing rate is a major advantage since in stream mining applications the biggest challenge is to catch up with the stream rate which in most of the recent time applications is too fast. In case of FP-stream the approach is often to process only a sampled data, thus loosing the accuracy of the results, while with Kaal we can accurately process data as fast as 43500 transactions per second. This proves very useful in current hot applications like sensor data, web click stream.

Second statistics collected was, time taken by the three algorithms for various supports. Again the experiments were repeated for three different average transaction lengths as shown in figure [8], figure [9] and figure [10].

For data set with average transaction length equal to 3, high support experiments were not done as the number of frequent pattern generated grew too low (Figure [10]).

Experiments suggested that original Kaal algorithm performs drastically better after a certain support level which varies according to the average transaction length. This is because the preprocessing step takes a lot of time if the support level is chosen very low (like 0.25%), but for larger support values Kaal has a big advantage because after preprocessing step next process of diff-set computations is not much costly. Thus lower the support more costly it is to perform the preprocessing. Where as in FP-stream for all levels of support the same process is repeated, thus the average processing time stays constant. The heuristics version of Kaal performs better than FP-stream in all cases with respect to time because it has the inherent
advantage of not doing all the computations and moving by a heuristics approach [14]. The only disadvantage is that the frequency count becomes approximate. The findings reflect that Kaal’s relative performance becomes better as the density of the data set increases. This is because in FP-stream the denser is the tree construction and processing, whereas in Kaal only the preprocessing step gets effected and the further steps only deal with the found out frequent patterns which makes the summary tree lighter.

Average frequent pattern lengths for various supports are shown in Figures [11], [12] and [13].

Average Length of frequent patterns generated by all three algorithms is same in all cases.

Frequent pattern growth pattern: In stream mining algorithms it is very important for the processing rate to catch up with the stream rate, thus the algorithm must have the flexibility to be stopped after partial processing of each batch if required. Further Kaal gives a quick on the fly picture of the frequent patterns, which is very essential in most current applications. The experimental result averaged over various datasets can be easily inferred from the graph illustrated in Figure [14].

The experimental result averaged over various datasets is as follows:

The findings show that Kaal starts giving the frequent patterns as soon at the processing of each batch starts, while FP-Stream which relies on the basic framework of FP-Growth [16] takes almost 50% of total time in initial steps of input processing and tree construction. Thus in FP-Stream we can expect to see the nature of the frequent patterns only after the long pre-processing step.

Total no. of frequent patterns generated by all three algorithms is same, for all supports. (The only difference is that the heuristics model does not give exact frequency for all patterns).

6. Conclusion

In this paper we have developed a new algorithm for frequent patterns mining in stream data. Extensive experimentations conclude that Kaal performs much better than other existing algorithms in the fields elucidated in the following paragraph.

Kaal finds frequent patterns from the beginning where as the other methods do not find frequent patterns until around 50% of the time has passed. The high rate of stream processing can be utilized for huge number of real time business applications and to provide security solutions (as mentioned previously in the section titled “Real World Applications of Stream Data Mining”). The time and space stability of the algorithm allows it to work consistently over very large stream as well. For very low support requirements Kaal heuristics model can be used giving approximate frequencies. The algorithm presented has capabilities for dealing with variable batch sizes as well, which is one of the major shortcomings of the present algorithms [7], [12] and [13]. Most importantly Kaal provides the flexibility to stop the batch processing even before 50% of total processing time, yet extracting over 70% patterns.

7. References

[10] J. Han, Pei, J., Yin, Y. and Mao R., "Mining Frequent Patterns without Candidate Generation.," presented at ACM SIGMOD, Dallas, TX, 2000.


8. Appendix
Table 2. Sample data set

<table>
<thead>
<tr>
<th>Ti d</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>B5</th>
<th>B6</th>
<th>B7</th>
<th>B8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>abcd</td>
<td>abd</td>
<td>abcd</td>
<td>abc</td>
<td>a</td>
<td>ae</td>
<td>abcd</td>
<td>e</td>
</tr>
<tr>
<td>2</td>
<td>acde</td>
<td>abc</td>
<td>acde</td>
<td>acd</td>
<td>ab</td>
<td>b</td>
<td>BCE</td>
<td>bcd</td>
</tr>
<tr>
<td>3</td>
<td>ade</td>
<td>acde</td>
<td>ace</td>
<td>ace</td>
<td>bce</td>
<td>a</td>
<td>ae</td>
<td>abcd</td>
</tr>
<tr>
<td>4</td>
<td>bcde</td>
<td>cde</td>
<td>ade</td>
<td>be</td>
<td>abd</td>
<td>d</td>
<td>abc</td>
<td>bde</td>
</tr>
<tr>
<td>5</td>
<td>bde</td>
<td>cde</td>
<td>ace</td>
<td>ad</td>
<td>a</td>
<td>abcd</td>
<td>abed</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>abd</td>
<td>de</td>
<td>ade</td>
<td>ae</td>
<td>acd</td>
<td>acde</td>
<td>abde</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>abd</td>
<td>ac</td>
<td>de</td>
<td>ace</td>
<td>abe</td>
<td>ade</td>
<td>abe</td>
<td>acde</td>
</tr>
<tr>
<td>8</td>
<td>abcd</td>
<td>abed</td>
<td>e</td>
<td>cde</td>
<td>bcd</td>
<td>d</td>
<td>bc</td>
<td>abce</td>
</tr>
<tr>
<td>9</td>
<td>de</td>
<td>cde</td>
<td>acde</td>
<td>bcd</td>
<td>abc</td>
<td>b</td>
<td>ab</td>
<td>abcd</td>
</tr>
<tr>
<td>10</td>
<td>acde</td>
<td>ae</td>
<td>ade</td>
<td>acd</td>
<td>ad</td>
<td>a</td>
<td>abe</td>
<td>abde</td>
</tr>
<tr>
<td>11</td>
<td>abcd</td>
<td>bce</td>
<td>bde</td>
<td>abc</td>
<td>abe</td>
<td>abce</td>
<td>cde</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>ace</td>
<td>bcd</td>
<td>abc</td>
<td>abe</td>
<td>abd</td>
<td>abe</td>
<td>abcd</td>
<td>e</td>
</tr>
</tbody>
</table>