SS-ZG548: ADVANCED DATA MINING

Lecture-07: Incremental Mining and Stream Processing

Dr. Kamlesh Tiwari
Assistant Professor
Department of Computer Science and Information Systems Engineering,
BITS Pilani, Rajasthan-333031 INDIA

Feb 03, 2018 (WILP @ BITS-Pilani Jan-Apr 2018)

Recap: AR Mining Without Candidate Generation

Frequent Pattern tree (FP-tree) structure[1], which is an extended prefix tree structure for storing information about frequent pattern

Tr ID	=	{items}
T_1	=	{A,B,E}
T_2	=	{B,D}
T_3	=	{B,C}
T_4	=	{A,B,D}
T_5	=	{A,C}
T_6	=	{B,C}
T_7	=	{A,C}
T_8	=	{A,B,C,E}
T_9	=	{A,B,C}

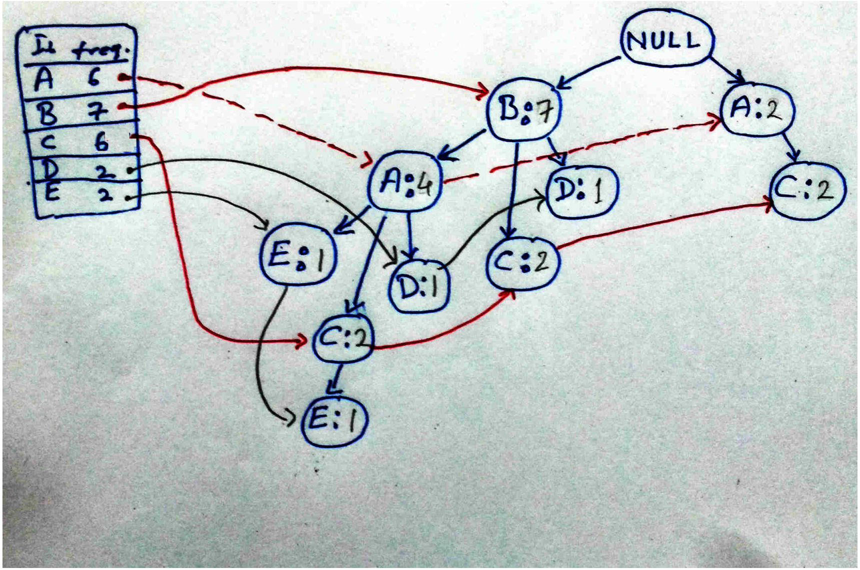
Item	Frequency	Priority
A	6	2
B	7	1
C	6	3
D	2	4
E	2	5

Tr ID	=	{items}
T_1	=	{B,A,E}
T_2	=	{B,D}
T_3	=	{B,C}
T_4	=	{B,A,D}
T_5	=	{A,C}
T_6	=	{B,C}
T_7	=	{A,C}
T_8	=	{B,A,C,E}
T_9	=	{B,A,C}

For 22% minimum support, min support count is 9*22/100 = 1.98

[1] Han, Jiawei and Pei, Jian and Yin, Yiwen, "Mining Frequent Patterns without Candidate Generation", in ACM Sigmod 29(2) pages 1-12, ACM 2000

Recap: FP-tree at work



Recap: FP-tree at work

Item	Conditional Pattern Base	Cond. FP-Tree
E	{B,A:1}, {B,A,C:1}	(<B:2,A:2>)
D	{B,A:1}, {B:1}	(<B:2>)
C	{B,A:2}, {B:2}, {A:2}	(<B:4,A:2>, <A:2>)
B	{}	(Null)
A	{B:4}	(<B:2>)

- **E:** {B,E}, {A,E}, {A,B,E}
- **D:** {B,D}
- **C:** {B,A,C}, {B,C}, {A,C}
- **A:** {B,A}

CATS Tree

- CATS Tree [2] (**C**ompressed and **A**rranged **T**ransaction **S**equences **T**ree) extends the idea of FP-Tree to improve storage compression and allow frequent pattern mining without generation of candidate itemsets.
- The proposed algorithms enable frequent pattern mining with different supports without rebuilding the tree structure
- Algorithms allow mining with a single pass over the database
- Handles insertion or deletion of transactions
- CATS Tree is a prefix tree and it contains all elements of FP-Tree including the header, the item links etc.

[2] Cheung, William and Zaiane, Osmar R, "Incremental mining of frequent patterns without candidate generation or support constraint", in Seventh International Database Engineering and Applications Symposium, pages 111–116, IEEE, 2003

CATS Tree

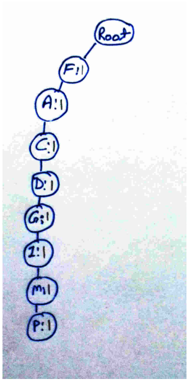
CATS Tree	**FP-Tree**
Contains all items in every transaction	Contains only frequent items
Sub-trees are locally optimized to improve compression	Sub-trees are not locally optimized
Ordering of items within paths from the root to leaves are ordered by local support	Ordering of items within paths from the root to leaves are ordered by global support
CATS nodes of the same parent are sorted in descending order according to local frequencies	Children of a node are not sorted

CATS Tree

Consider database as

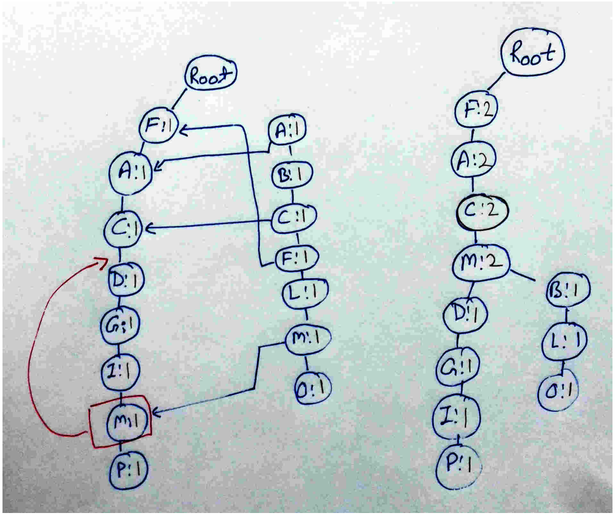
TID	Transaction
T1	F,A,C,D,G,I,M,P
T2	A, B, C, F, L, M, O
T3	B, F, H, J, O
T4	B, C, K, S, P
T5	A, F, C, E, L, P, M, N

CATS Tree with T1



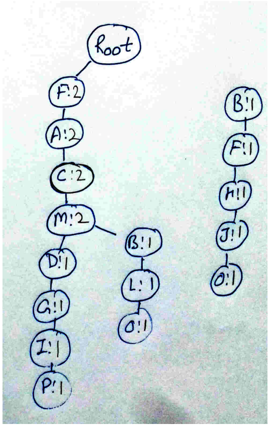
CATS Tree

With T2 $=\{A, B, C, F, L, M, O\}$

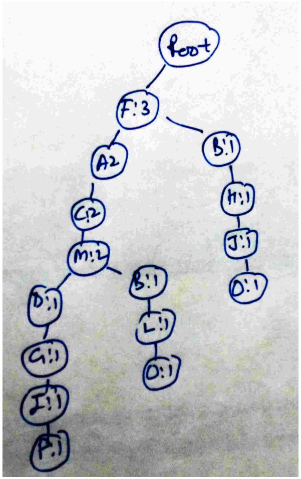


CATS Tree

With T3 =$\{$B, F, H, J, O$\}$

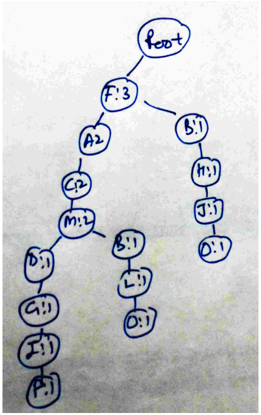


CATS Tree with T3

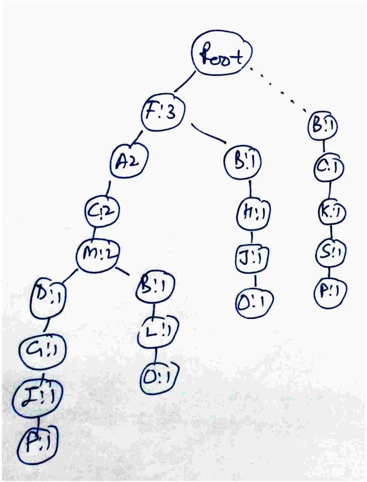


CATS Tree

With T4 ={B, C, K, S, P}

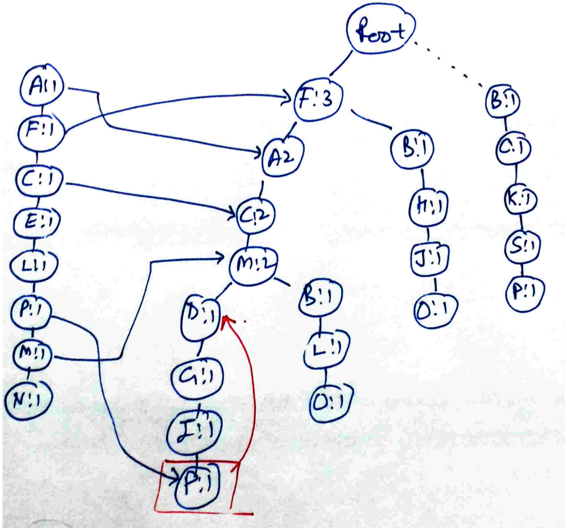


CATS Tree with T4



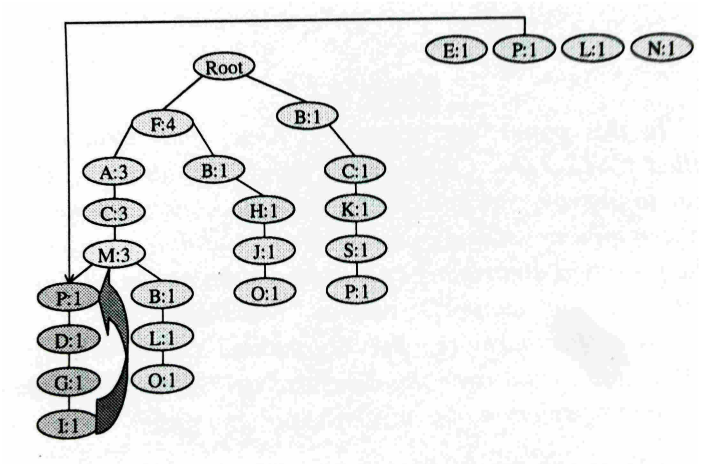
CATS Tree

With T5 ={A, F, C, E, L, P, M, N}



CATS Tree

With T5 ={A, F, C, E, L, P, M, N}

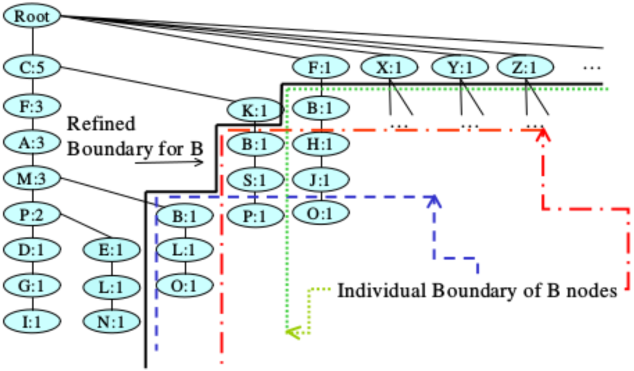


CATS Tree properties

1. The compactness of CATS Tree measures how many transactions are sharing a node. Compactness decreases as it is getting away from the root. This is the result of branches being arranged in descending order.
2. No item of the same kind could appear on the lower right hand side of another item. If there were items of the same kind on the right hand side, they should have been merged with the node on the left to increase compression. Any items on the lower right hand side can be switched to the same level as the item, split nodes as required if switching nodes violates the structure of CATS Tree. After that they can be merged with the item on the left.
3. Because of the above properties, a vertical downward boundary is formed beside each node and a horizontal rightward boundary is formed at the top of each node to make a step-like individual boundary.

CATS Tree properties

Boundaries of multiple items can be joined together to form a refined boundary for a particular item. Items of the same kind can only exist on the refined boundary.



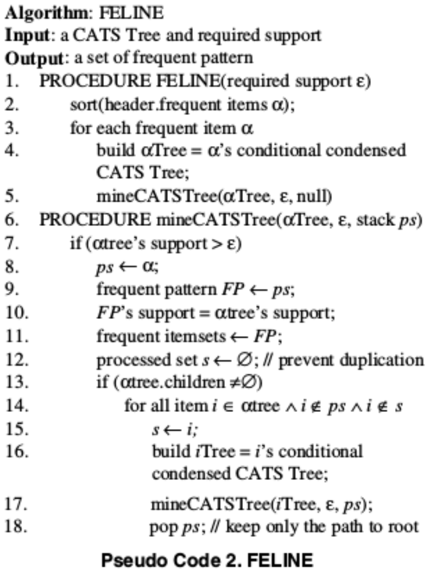
CATS Tree properties

1. New transactions are added at the root level. At each level, items of the transaction are compared with those of children nodes. If the same items exist in both the new transaction and that of the children nodes, the transaction is merged with the node at the highest frequency. The frequency of the node is incremented. The remainder of the transaction is added to the merged nodes and the process is repeated recursively until all common items are found. Any remaining items of the transaction are added as a new branch to the last merged node.
2. In general, it is impossible to build a CATS Tree with maximal compression and without prior knowledge of the data. Therefore the compression of a CATS Tree is sensitive to both ordering of transaction and items within the transactions.

CATS Tree frequent pattern mining

1. FrEquent/Large patterns mINing with CATS trEe (FELINE)
2. Unlike FP-tree, once the CATS Tree is built, it can be mined repeatedly for frequent patterns with different support thresholds without the need to rebuild the tree.
3. FELINE employs divide and conquer, fragment growth method to generate frequent patterns without generating candidate itemsets.
4. FELINE partitions the dataset based on what patterns transactions have. For a pattern called p, a ps conditional CATS Tree is a tree built from all transactions that contain pattern p. Transactions contained in a conditional CATS Tree can be easily gathered by traversing the item links of pattern p.

FELINE

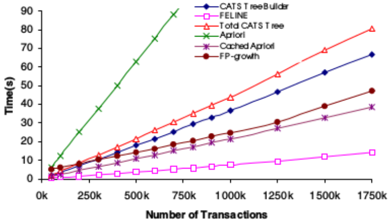


Algorithm: FELINE
Input: a CATS Tree and required support
Output: a set of frequent pattern

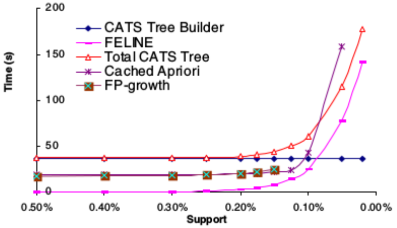
```
1.  PROCEDURE FELINE(required support $\varepsilon$)
2.      sort(header.frequent items $\alpha$);
3.      for each frequent item $\alpha$
4.          build $\alpha$Tree = $\alpha$'s conditional condensed
            CATS Tree;
5.          mineCATSTree($\alpha$Tree, $\varepsilon$, null)
6.  PROCEDURE mineCATSTree($\alpha$Tree, $\varepsilon$, stack $ps$)
7.      if ($\alpha$tree's support > $\varepsilon$)
8.          $ps \leftarrow \alpha$;
9.          frequent pattern $FP \leftarrow ps$;
10.         $FP$'s support = $\alpha$tree's support;
11.         frequent itemsets $\leftarrow FP$;
12.         processed set $s \leftarrow \varnothing$; // prevent duplication
13.         if ($\alpha$tree.children $\neq \varnothing$)
14.             for all item $i \in \alpha$tree $\wedge\ i \notin ps \wedge i \notin s$
15.                 $s \leftarrow i$;
16.                 build $i$Tree = $i$'s conditional
                    condensed CATS Tree;
17.                 mineCATSTree($i$Tree, $\varepsilon$, $ps$);
18.                 pop $ps$; // keep only the path to root
```

Pseudo Code 2. FELINE

CATS Tree Performance

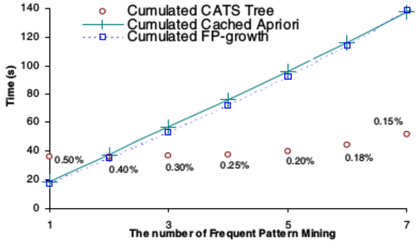


Scalability of CATS Tree with respect to number of Transactions with single run

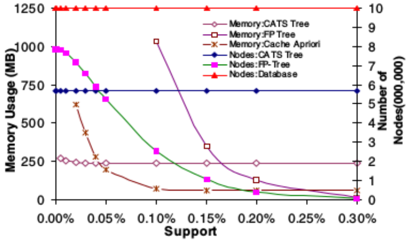


Scalability of CATS Tree with respect to support with single run

CATS Tree Performance



Build once, mine many with CATS Tree: scalability with multiple runs



Memory Comparison

Canonical-order Tree (CanTree)

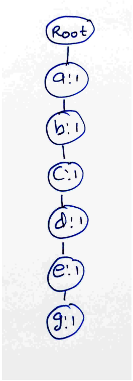
- **Can**onical-order **Tree** (CanTree) [3] captures the content of the transaction database and orders tree nodes according to some canonical order.
- CanTree can be easily maintained when database transactions are inserted, deleted, and/or modified. For example, the CanTree does not require adjustment, merging, and/or splitting of tree nodes during maintenance.
- No rescan of the entire updated database or reconstruction of a new tree is needed for incremental updating.

[3] Leung, Carson Kai-Sang and Khan, Quamrul I and Li, Zhan and Hoque, Tariqul, “CanTree: a canonical-order tree for incremental frequent-pattern mining”, in Knowledge and Information Systems, pages 287–3116, Springer, 2007

CanTree in action

Consider transaction database as

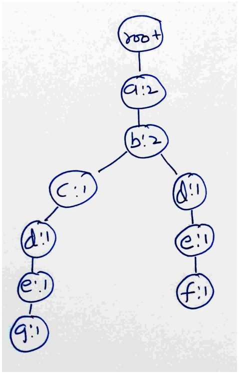
ID	Transaction	
T1	a, d, b, g, e, c	←
T2	d, f, b, a, e	
T3	a	
T4	d, a, b	
T5	a, c, b	
T6	c, b, a, e	
T7	a, b, c	



CanTree in action

Consider transaction database as

ID	Transaction
T1	a, d, b, g, e, c
T2	d, f, b, a, e $\leftarrow$
T3	a
T4	d, a, b
T5	a, c, b
T6	c, b, a, e
T7	a, b, c

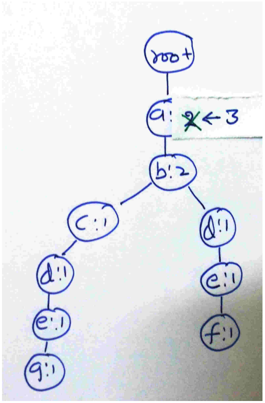


CanTree in action

Consider transaction database as

ID	Transaction
T1	a, d, b, g, e, c
T2	d, f, b, a, e
T3	a
T4	d, a, b
T5	a, c, b
T6	c, b, a, e
T7	a, b, c

$\leftarrow$

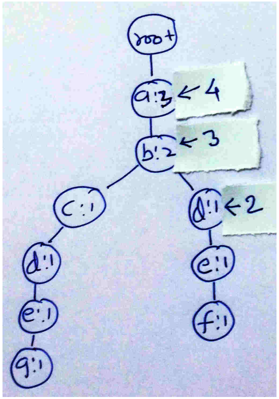


CanTree in action

Consider transaction database as

ID	Transaction
T1	a, d, b, g, e, c
T2	d, f, b, a, e
T3	a
T4	d, a, b
T5	a, c, b
T6	c, b, a, e
T7	a, b, c

←

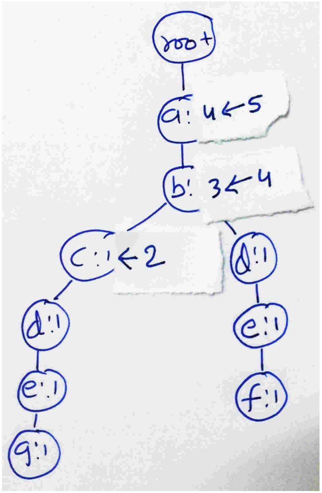


CanTree in action

Consider transaction database as

ID	Transaction
T1	a, d, b, g, e, c
T2	d, f, b, a, e
T3	a
T4	d, a, b
T5	a, c, b
T6	c, b, a, e
T7	a, b, c

$\leftarrow$

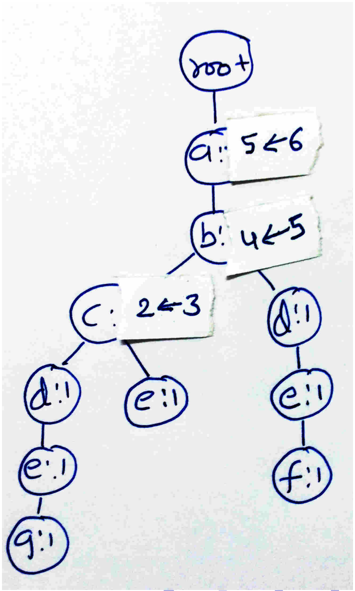


CanTree in action

Consider transaction database as

ID	Transaction
T1	a, d, b, g, e, c
T2	d, f, b, a, e
T3	a
T4	d, a, b
T5	a, c, b
T6	c, b, a, e
T7	a, b, c

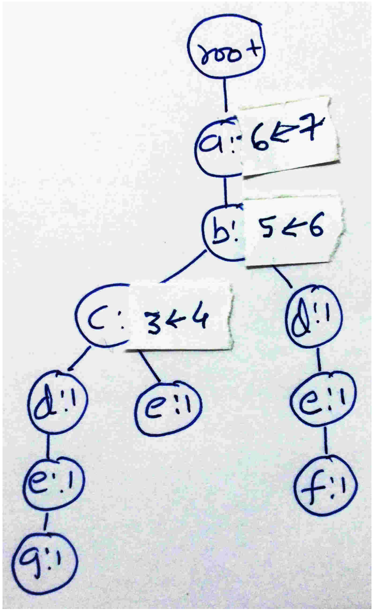
$\leftarrow$



CanTree in action

Consider transaction database as

ID	Transaction
T1	a, d, b, g, e, c
T2	d, f, b, a, e
T3	a
T4	d, a, b
T5	a, c, b
T6	c, b, a, e
T7	a, b, c ←



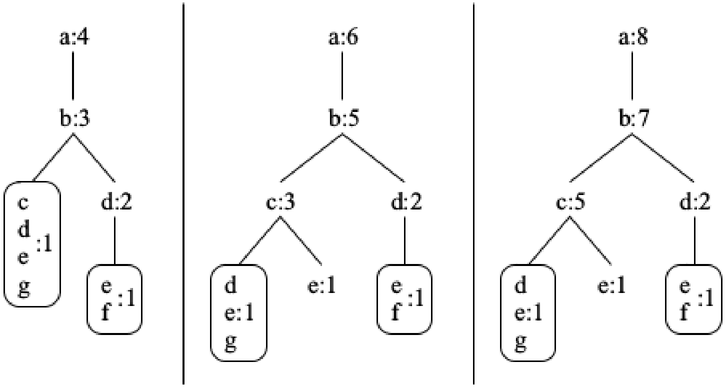
CanTree advantages

CanTree improves previous algorithms as

1. For our CanTree, items are arranged according to some canonical order that is unaffected by the item frequency. Hence, searching for common items and mergeable paths during the tree construction is easy. No extra downward traversals are needed during the mining process.
2. The construction of CanTree is independent of threshold values.
3. Since items are consistently ordered; any insertions, deletions, and/or modifications of transactions have no effect on the ordering of items in the tree. As a result, swapping of tree nodeswhich may lead to merging and splitting of tree nodes is not needed.

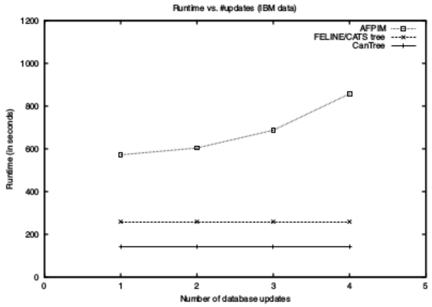
CanTries: a variant of CanTrees

Although the number of tree nodes is no more than the total number of items in the database, we can further reduce the size

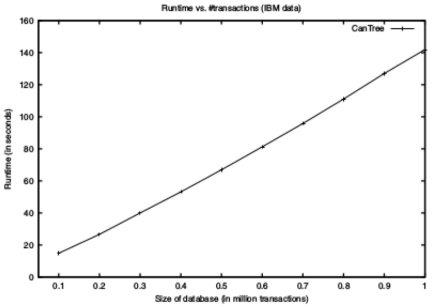


CanTree performance

IBM transaction database consists of 1M transactions with an average transaction length of 10 items and a domain of 1,000 distinct items.



Number of updates



Scale-up

Mining over data streams

- What is data stream
  - Data in rapid succession
  - Generally insufficient storage space to accommodate all values
  - Dynamic nature
  - Stale items
  - No re-scan
- Continuous update in model
- An example: report missing number

  I would tell you n-1 numbers from the first n natural numbers, without repetition, in an arbitrary order. Can you report the missing one? [Constraints are on memory and processing]

Frequent pattern mining over data streams

- Applications involves retail market data analysis, network monitoring, web usage mining, and stock market prediction.
- Using sliding window
- Efficiently remove the obsolete, old stream data
- Compact Pattern Stream tree (CPS-tree)
- Highly compact frequency-descending tree structure at runtime
- Efficient in terms of memory and time complexity
- Pane and window
- Insertion and restructuring

Frequent items over data stream

- Let identity of items be $\{1, 2, 3, ..., n\}$ and frequency of item i be f_i
- Assume general arrival model, (i, v), $v > 0$ represents arrival and $v < 0$ is departure. Sum of frequencies $m = \sum_i f_i$
- Item i is frequent of $f_i > m/(k+1)$ for some k. There can be at most k frequent items (why ? proof?) $m > k(k+1)$
- Any algorithm that finds all frequent and only frequent items requires $\log({}^nC_k)$ bits (how? $2^s \geq ({}^nC_k)$)

Find approx frequent items

Wish to output a list of items such that

- Every item in the list has frequency $f_i > (1 - \epsilon)\frac{m}{k+1}$
- All the items having frequency at least $(1 + \epsilon)\frac{m}{k+1}$ is in the list

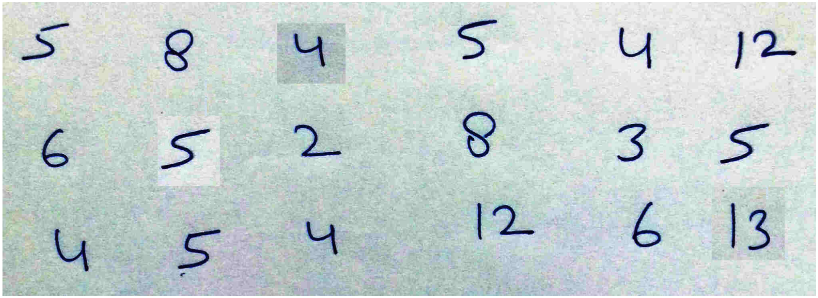
Output should satisfy above two properties with probability $(1 - \delta)$

Algorithm maintains a data structure A over the stream. Step to update an item *x* are explained below

```
1 IF ( A.ismember(x) ) A[x]++
2 ELSE A.insert(x)
3 IF ( A.size == k+1) THEN $\forall y \in A$
4             A[y]−−,
5             IF (A[y] ==0) A.delete(y);
```

In action: Frequent items

Take $k = 4$



Execution step by step:

Thank You!

Thank you very much for your attention!

Queries ?