Frequent Itemset Mining Revisited

April 21, 2004

Lu Hongjun，*HKUST*
Email: *luhj@cs.ust.hk*
URL: *http://www.cs.ust.hk/~luhj*

Outline

- Background
- Previous work
- SSP: scalable mining of frequent itemsets
- CFP-tree: storing and querying frequent itemsets
- Summary

Frequent Itemset Mining: Motivation

- Frequent itemset: pattern that occurs frequently in the database
- Applications
  - Market-basket data analysis
  - Web log mining
  - DNA sequence analysis
- Foundations of many data mining tasks
  - Association rule, correlation
  - Associative classification
  - Sequential patterns, temporal or cyclic association, partial periodicity patterns, episodes
  - Iceberg cube computation

Frequent Itemset Mining: Problem Statement

- Given
  - A set of items $I = \{a_1, a_2, \ldots, a_n\}$
  - A transaction database $D = \{t | t \subseteq I\}$.
  - ***support*** $(p) = \| \{t | p \subseteq t\} \|$.
  - minimum support threshold *min_sup*
- Output
  - Every itemset *p* such that *support(p)/||D|| >= min_sup*
- *Apriori property (anti-monotone property)*
  - If an itemset is not frequent, then none of its superset can be frequent

Frequent Itemset Mining: An Example

TID	Transactions
1	a, b, c, f, m, p
2	a, d, e, f, g
3	a, b, f, m, n
4	a, c, e, f, m, p
5	d, f, n, p
6	a, c, h, m, p
7	a, d, m, s

Transaction database

40%

Frequent Itemsets
c:3, d:3, p:4, f:5, m:5, a:6
cp:3, cm:3,ca:3, pf:3, pm:3, pa:3, fm:3, fa:4, ma:5
cpm:3, cpa:3, cma:3, pma:3, fma:3
cpma:3

Frequent Itemset Mining: Challenges

- Challenge
  - The size of search space is exponential to the number of items in the database
- Typical approaches
  - Candidate generate-and-test approach
  - Filter-and-refine approach
  - Vertical Mining approach
  - Pattern growth approach

Candidate Generate-and-test Approach

- Basic Apriori algorithm [VLDB'94]
  - scan database and count frequent 1-itemsets
  - In subsequent iterations
    - Pairs of frequent k-itemsets are joined to form candidate (k+1)-itemsets
    - Scan database to verify candidate (k+1)-itemsets and generate frequent (k+1)-itemsets.
- Drawbacks
  - Scan database multiple times
    - equal to the maximal length of frequent itemsets
  - Generate and test a large number of candidate itemsets
    - Subset matching is expensive

Speeding up Apriori algorithm

- **DHP** [SIGMOD'95]
  - prune candidate itemsets using hashing
  - trim both number of transactions and number of items in each transaction after each iteration
- **DIC** [SIGMOD'97]
  - count support for an itemset shortly after all of its subsets are proved to be frequent rather than wait until next database scan
- In worst case, the number of database scan is still equal to the maximal length of frequent itemsets

Filter-and-refine Approach

- Framework
  - In the filter phase, generate candidate itemsets
  - In the refine phase, scan database to verify the validity of each candidate itemsets
  - Usually scans database only twice
- Drawbacks
  - Generate and test a large number of candidate itemsets
  - The number of candidate itemsets generated can be larger than that of the basic Apriori algorithm

Filter-and-refine Algorithms

- Partition [VLDB'95]
  - Partition the database into disjoint partitions such that each partition can be mined in main memory. All the itemsets that are frequent in at least one partition form the candidate itemsets.
- Sampling [VLDB'96]
  - In the first pass, pick a random sample to compute frequent itemsets along with a negative border.
  - In the second pass, generate all frequent itemsets.
- BBS (**B**it-sliced **B**loom-filtered **S**ignature file) [ICDE'02]
  - In the filter strategy, the candidate patterns are obtained by scanning BBS instead of the database.
  - Tuning the size of BBS for optimal performance is critical

Vertical Mining

- Each itemset is associated with a tid list or tid bitmap
  - tid list: list of transaction ids containing that itemset
  - support counting is performed by tid list/bitmap join, which is more efficient than subset matching
- drawbacks:
  - Constructing and maintaining a large number of tid list/bitmap
  - Not scale well with respect to the number of transactions
- Optimizations
  - Tid bitmap compression
  - Use diffset to reduce size

Pattern Growth Approach

- Basic idea
  - Grows a frequent itemset from its prefix to avoid candidate generation and test
  - Using divide-and-conquer methodology
- Framework
  - Find all frequent items in the database, I = {a, b, c, d}
  - Divide the search space into disjoint sub-spaces:
    - Frequent Itemsets containing *a*
    - Frequent itemsets containing *b* but no *a*
    - Frequent itemsets containing *c* but no *a*, *b*
    - Frequent itemsets containing *d* but no *a*, *b*, *c*
  - Accordingly, the database is divided into partitions (**conditional database**) after removing infrequent items
    - All transactions containing a
    - All transactions containing b (item a is eliminated)
    - All transactions containing c (items a and b are eliminated)
    - All transactions containing d (items a, b and c are eliminated)
  - Mine each conditional database recursively

Pattern Growth Approach---Key Factors

- Total number of conditional databases
- Size of individual conditional database

} Depends on item search order

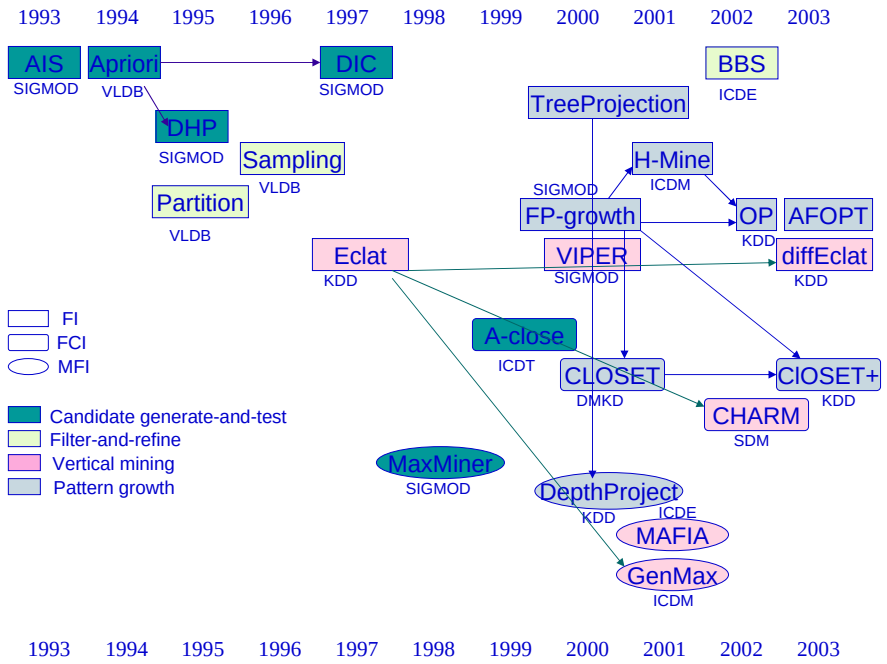
- Conditional database representation format
  - Tree-based structure: low traversal cost but high construction cost
  - Array-based structure: low construction cost but high traversal cost
- Conditional database construction strategy
  - Physical: expensive but save traversal cost
  - Pseudo: cheap but incur high traversal cost
- Conditional database traversal strategy
  - Top-down
  - Bottom-up

Pattern Growth Algorithms

algorithms	item search order	CondDB format	ConDB construction	tree traversal
Tree Projection	static lexicographic	array	adaptive	-
FP-growth	dynamic frequency	FP-tree	physical	bottom-up
H-mine	static lexicographic	hyper-structure	pseudo	-
OP	adaptive	adaptive	adaptive	bottom-up
PP-mine	static lexicographic	PP-tree	pseudo	top-down
AFOPT	dynamic frequency	adaptive	physical	top-down
CLOSET+	dynamic frequency	FP-tree	adaptive	adaptive

Mining Frequent Closed/Maximal Itemsets

- The complete set of frequent itemsets can be very large on dense datasets
  - If a length-100 itemset is frequent, then all of its 2^{100}-1 subsets are frequents. *Infeasible!*
- Solution: mining frequent closed/maximal itemsets.
  - An itemset is **closed** if all of its supersets are less frequent than it
  - An itemset is **maximal** if none of its supersets is frequent
  - The number of frequent closed/maximal itemsets can be substantially smaller than the number of frequent itemsets



Previous work---summary

- Candidate generate-and-test algorithms
  - Scan database multiple times
  - Generate and test a large number of candidate itemsets
- Vertical mining algorithms
  - Not scalable well with respect to the number of transactions
- Pattern growth algorithms
  - Construct and traverse a large number of conditional databases
  - Existing algorithms mainly focus on optimizing in-memory performance
- A recent comparative study (FIMI'03 workshop) shows that few existing algorithms can scale-up to very large databases with millions of transactions.

Our Work -- Overview

- SSP: a scalable algorithm for mining frequent itemsets from very large databases with millions of transactions
  - Partitioning database according to search space
  - Specially designed for out-of-core mining
  - Taking memory constraints into consideration in algorithm design
  - Managing memory in fully dynamic fashion
- CFP-tree: a compact disk-based structure for storing and querying frequent itemsets
  - Stores only frequent closed itemsets
  - Supports three basic types of queries
    - Queries with minimum support constraints
    - Queries with item constraints

Partition Algorithm [VLDB'95]

- Basic idea
  - If we divide the database into several disjoint partitions, then a frequent itemset must be frequent in at least one partition.
- Algorithm
  - Partition the database into disjoint partitions such that each partition can be mined in main memory.
  - All the itemsets that are frequent in at least one partition form the candidate itemsets. The whole database is scanned to find the exact set of frequent itemsets
- Pros & cons
  - \+ Scan database only twice
  - \- Duplicate computation cost
  - \- It is very hard to accurately estimate the amount of memory consumed by the mining algorithm when partitioning the database.

SSP: Search Space based Partitioning

- It is based on the pattern growth approach, and partitions the database according to the search space of the frequent itemset mining problem.
- Different partitions share data but do not share frequent itemsets.
  - \+ The frequent itemsets mined from each partition are final, therefore we do not need to scan the whole database to verify their supports.
  - \+ We need to keep only data in memory.
  - \- The total size of the partitions can be much larger than the size of the database
- Main issue: utilize the data overlap among partitions to reduce I/O cost.

SSP Algorithm---framework

SSP Algorithm (l, D_l, min_sup)

1. Scan D_l to count frequent items, and sort them in descending frequency order, denoted as F= {a_1, a_2, …, a_n}
2. for (every item $a \in F$)
   $D_{l \cup a} = \varnothing$;
3. For (every transaction $t \in D_l$) //construct conditional database
   1. Remove infrequent items from t, and sort remaining items according to their orders in F;
   2. Let a be the first item of t, insert t into $D_{l \cup a}$.
4. for (j=1;j<=n;j++)
   1. Output $s = l \cup a_j$;
   2. SSP(s, D_s, min_sup);
   3. for (every transaction $t \in D_s$) //push-right step
      1. $t = t$ - {a};
      2. Let a' be the first item of t, insert t into $D_{l \cup a'}$.

SSP Algorithm--- Features

- item search order: ascending frequency order
  - The most infrequent item has the largest candidate extension set, with the increasing of frequency, the number of candidate extensions decreases
    - Balances the size of conditional databases thus ensures that every time a small conditional database is pushed right
    - Balances the size of the sub search spaces thus ensures that the memory consumption for mining the conditional databases cannot be large
- Conditional database representation format: adaptive
  - Sparse : array
  - Dense : prefix-trie
  - Extremely dense: bucket counting
- Conditional database construction strategy
  - Physical construction

SSP Algorithm --- Main Issue

- A transaction can belong to multiple conditional databases, but at any time it can belong to one and only one conditional database.
  - The total size of the conditional database is can be much larger than the original database. On average, $L_{avg}/2$ times larger, where L_{avg} is the average transaction length.
  - The space needed for holding all the conditional databases cannot be larger than the original database.
- If all the conditional databases cannot be held in memory, a transaction may be write to and read from disk many times, which incurs high I/O cost.
- Main issue: reducing I/O cost by utilizing the transaction sharing among conditional databases
  - SSP-naïve
  - SSP-static
  - SSP-dynamic

SSP- Naïve algorithm

- It is realistic to assume that a single conditional database and all its descendant conditional databases can be held in memory
  - The amount of main memory available nowadays is very large.
  - Ascending frequency order: ensures that a single conditional database is much smaller than the original database, and the size of its descendant conditional databases also shrinks quickly.
  - If a single conditional database can fit in memory but there is no enough space for holding all of its descendant conditional databases, we can use the pseudo-construction strategy.
  - If a single conditional database cannot be held in main memory, we can recursively apply the out-of-core algorithm on the conditional database.

SSP- naïve Algorithm

- Basic idea:Keep one conditional database in memory at one time.
  - When constructing new conditional databases from the original transaction database, we keep only the first conditional database in memory and write all the others on disk.
  - When the mining on a conditional database in memory is finished, some of the transactions in it will be written to other conditional databases.
  - One optimization is that if a transaction belongs to the next conditional database to be mined, we keep it in memory.

SSP- ntatic Algorithm

- It is a waste of memory to keep only one conditional database in memory given the large amount of memory available nowadays.
- Solution
  - keep adjacent conditional databases into memory as more as possible---How many?
- Observation:
  - If we keep conditional databases $D_{a_{i+1}}$, …, $D_{a_{i+m}}$ in memory, the space required for storing these m conditional databases is smaller than the total size of these m conditional databases because of transaction sharing.
- To accurately calculate the borders between partitions, we use a matrix C, called *differential matrix*, to record the differences among conditional databases.

SSP- dynamic Algorithm

- SSP-static Algorithm
  - requires extra I/O cost to partition the database.
  - uses the total length of the transactions in a conditional database as the estimation of the size of the conditional database, which is a a rather loose upper bound when the conditional database is dense.
- SSP-dynamic algorithm
  - Adopt a lazy writing strategy to fully utilize memory: writes conditional databases on disk only when new structures are to be created but there is no memory available

SSP-dynamic algorithm

- The search space is traversed in depth-first-order, therefore the exact access order of the conditional databases is known.
  - put the conditional databases that will be accessed last on disk to release memory for new data.
  - Maintain an active stack *S* to trace the conditional databases that have not been processed yet at every level.
    - Conditional database dumping order: from bottom to top, from right to left.
- It is inefficient to release memory for every transaction to be inserted.
  - Solution: estimate the size of memory required for future mining when releasing memory.

SSP Algorithm---Summary

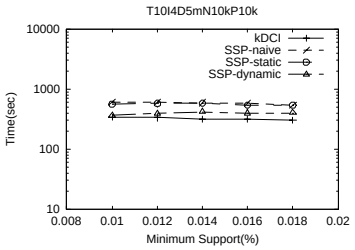
- SSP-naïve
  - Utilizes the overlap between adjacent conditional databases to reduce I/O cost.
- SSP-static
  - Utilizes the overlap between adjacent partitions to reduce I/O cost.
  - Requires an additional database scan to compute differential matrix
- SSP-dynamic
  - Uses a lazy-writing strategy to guarantee the full utilization of the memory.
  - Dynamic in nature

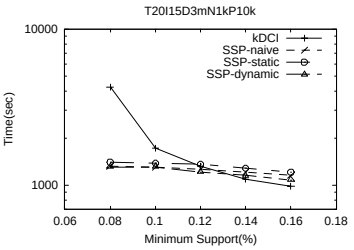
Experiment Setting

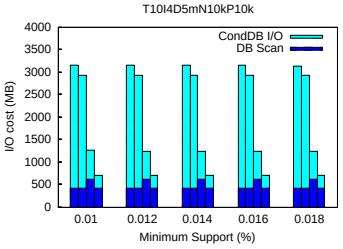
- Test environment
  - 1.0GHz Pentium III, 256MB memory
  - Linux mandrake
- Datasets: generated by IBM dataset generator

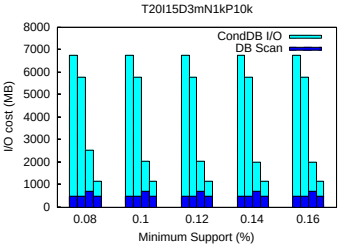
Datasets	size	#Trans	#Item	AvgTL	MaxTL
T10I4D5mN10kP10k	238MB	4,922,589	7692	36	10.14
T20I15D3mN1kP10k	223MB	2,719,116	994	61	21.82
T30I20D2mN5kP5k	277MB	1,933,520	4706	88	31.18
T40I10D2mN10kP10k	374MB	1,999,994	8912	84	39.95

Minimum Support

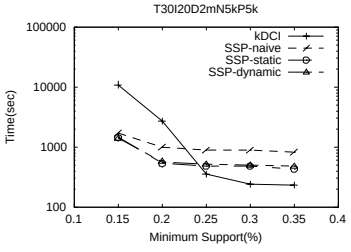


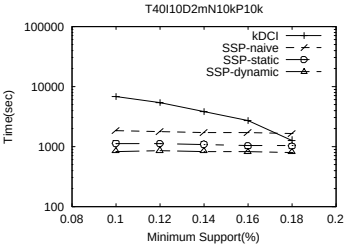


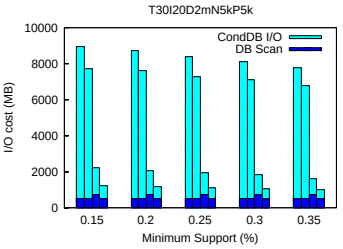


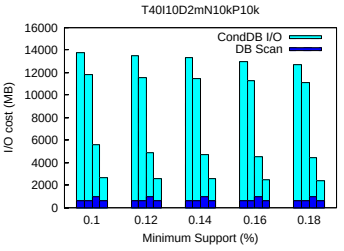


Minimum Support

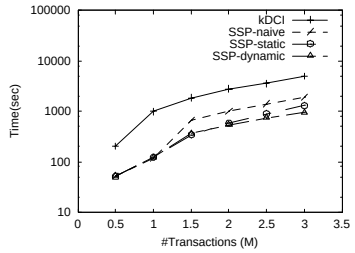


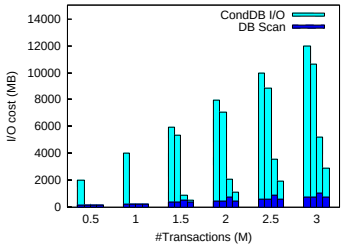


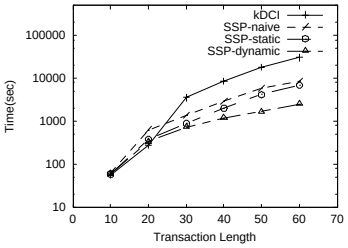


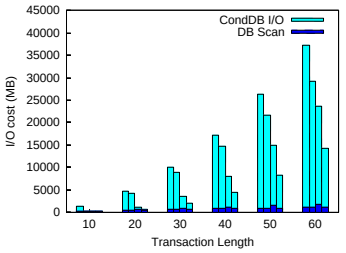


Dataset Generating Parameters

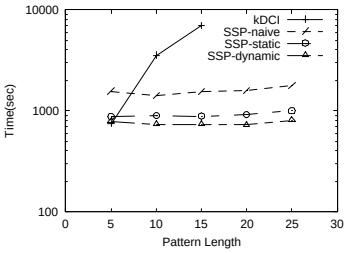


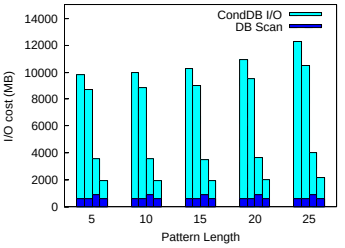


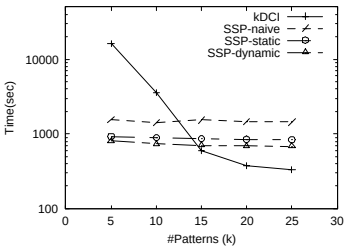


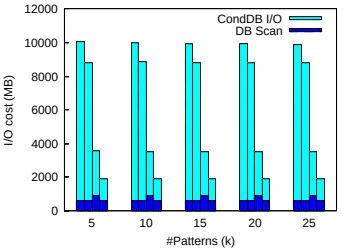


Dataset Generating Parameters

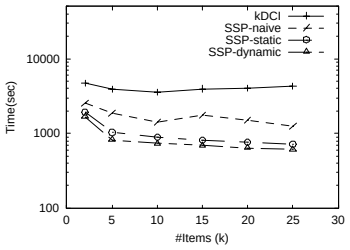


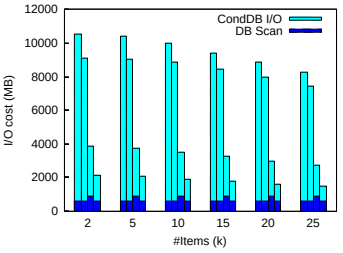




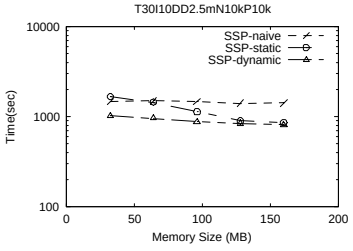


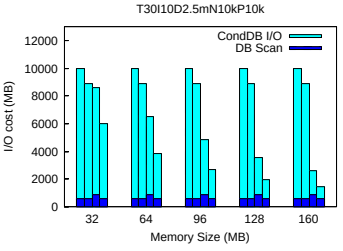
Dataset Generating Parameters





Memory Size





CFP-Tree---Motivation

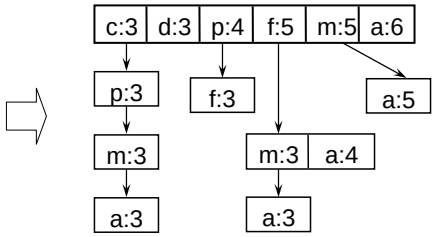
- Frequent itemset mining
  - A time-consuming process
    - I/O intensive : scan database multiple times
    - CPU intensive : count supports for a large number of itemsets
  - A repeated process
    - Different database and/or applications require different parameters
    - Often no guidelines for choosing proper parameters
- Solution: mining once and using many times
  - CFP-tree: a compact structure for storing and querying frequent itemsets

CFP-tree---Overview

- A compact structure for storing frequent itemsets
  - Stores only frequent closed itemsets to reduce tree size
- Supports three basic types of queries
  - Queries with minimum support constraints
    - Find all itemsets with support no less than s%
  - Queries with item constraints
    - Find all frequent itemsets containing items $\{a_1, a_2, \ldots a_m\}$
  - Queries with both support and item constraints

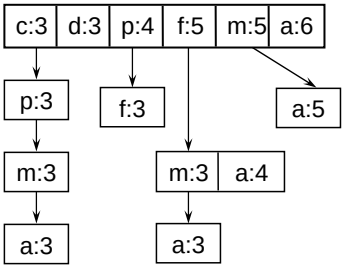
CFP-tree Structure

Frequent Closed Itmesets
d:3, p:4, f:5, a:6
pf:3, fa:4, ma:5
fma:3
cpma:3



- Items in a node are sorted in ascending order of their frequencies
- An entry stores:
  - Item id,
  - Support
  - A child pointer
  - A hash bitmap: to indicate whether an item appears in the subtree

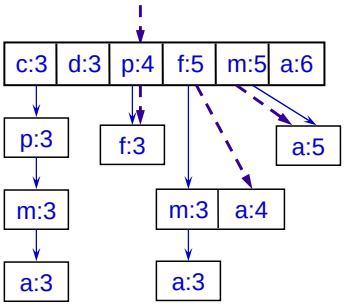
CFP-tree---Properties



- Apriori Property
  - The support of entry cannot be greater than its ancestors.
- Left Containment Property
  - The item of an entry *E* can only appear in the subtrees pointed by entries before *E* or in *E* itself

Query With Support Constraint

- Find all frequent patterns with support no less than **50%**
  - Apriori property : only entry ***p*** and **entries after *p*** need to be visited

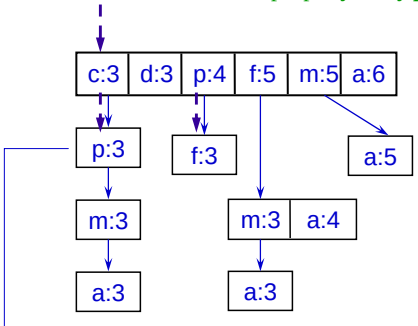


Frequent Patterns:

{ p } : 4
{ f } : 5
{ f, a } : 4
{ m, a} : 5
{ a } : 6

Query with Item Constraint

- Find all frequent patterns containing item ***f*** and ***p***
  - Left containment property***:*** only ***p*** and **entries before *p*** need to be visited



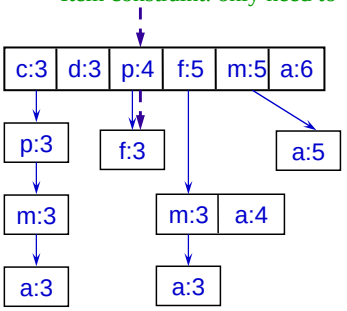
Frequent Patterns:

{ p, f } : 3

The corresponding bit of *f* in the hash bitmap is 0

Query with Both Constraints

- Find all frequent pattern with support no less than **50%** and containing items ***f*** and ***p***
  - Minimum support constraint : only need to visit entry ***p***, ***f***, ***m*** and ***a***
  - Item constraint: only need to visit entry ***c***, ***d*** and ***p***



No pattern satisfies both constraints

Construction Algorithm

- Pattern growth approach
  - Construct a *conditional database* D_p for each frequent pattern p such that all the patterns with p as prefix can be mined form D_p
  - For each conditional database D_p
    - First Scan : count frequent items
    - Second scan : construct a new conditional database for each frequent item
    - Mine each individual conditional database
  - The original transaction database D can be viewed as the conditional database of pattern $p=\varnothing$
- Conditional database representation
  - adaptive

Removing Non-closed Patterns

- A pattern ***p*** is closed iff two conditions hold
  - There is no previously mined pattern which is a proper superset of ***p*** and has the same support as ***p***.
  - All the items in D_p has a lower support than ***p***.
- Two pruning conditions to save mining cost
  - If condition 1 does not holds, then none of the patterns mined from D_p can be closed. (Use CFP-tree to do the checking)
  - If there exist an item ***i*** in D_p such that ***i*** appears in every transaction of D_p, then the patterns containing ***p*** but no ***i*** can be ignored. (Easy)

A Running Example

- Count frequent items and create the first CFP-tree node
- Construct conditional databases for frequent items

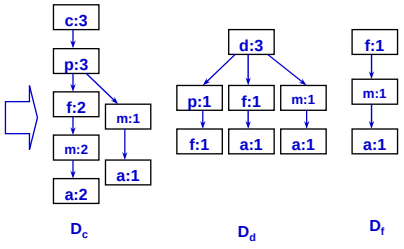
ID	Transactions
1	a, b, c, f, m, p
2	a, d, e, f, g
3	a, b, f, m, n
4	a, c, e, f, m, p
5	d, f, n, p
6	a, c, h, m, p
7	a, d, m, s

40%

1	c:3	d:3	p:4	f:5	m:5	a:6

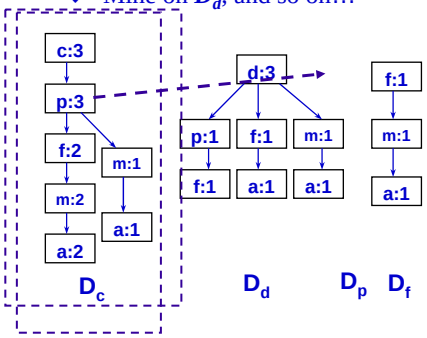
CFP-tree

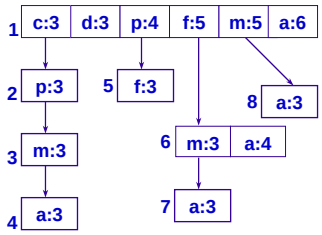
ID	Transactions
1	c, p, f, m, a
2	d, f, a
3	f, m, a
4	c, p, f, m, a
5	d, p, f
6	c, p, m, a
7	d, m, a



A Running Example (contd.)

- Mine on D_c
  - ***p*** is the only child of ***c***, a new node (2) is created as the child of ***c***
  - ***m*** and ***a*** have the same support as ***c***, a node (3, 4) is created for each of them
- Push-right D_c
- Mine on D_d, and so on…





CFP-tree

Experiment Setting

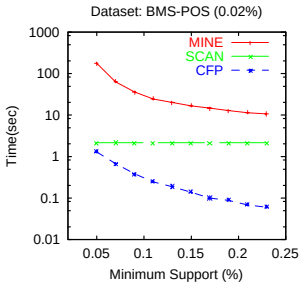
- Test environment
  - 2.26GHz Pentium IV, 512MB memory
  - Window XP
- Datasets

Datasets	size	#Trans	#Item	AvgTL	MaxTL
BMS-POS	19.20MB	515,597	1657	6.53	164
Pumsb	14.75MB	49,046	2113	74.00	74
T20I10D1000k	89.57MB	987,139	8876	20.23	54

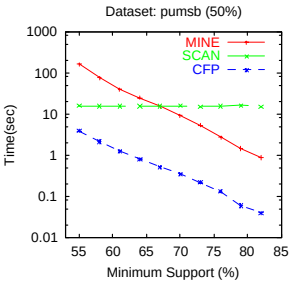
- BMS-POS is obtained from kdd cup 2000 website, and it contains click-stream data
- Pumsb comes from UCI machine learning repository, and it contains census data
- T20I10D1000k is generated by IBM synthetic dataset generator

Querying Processing Time

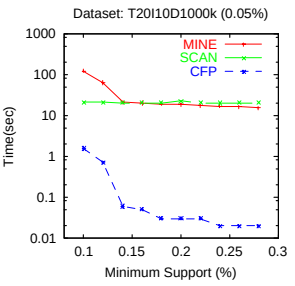
- Queries with minimum support constraints



BMS-POS (80.7MB)



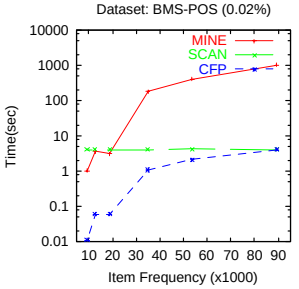
pumsb (142.1MB)



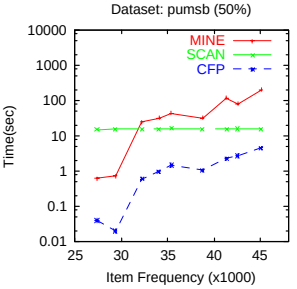
T20I10D1000k(199.5MB)

Querying Processing Time (contd.)

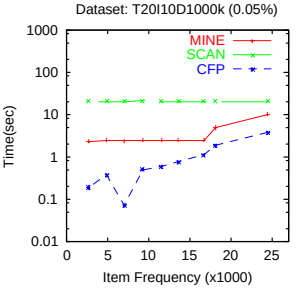
- Queries with item constraints



BMS-POS (80.7MB)



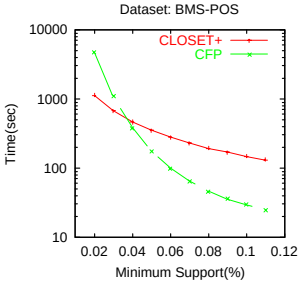
pumsb (142.1MB)



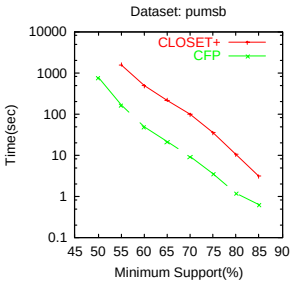
T20I10D1000k(199.5MB)

Construction Time

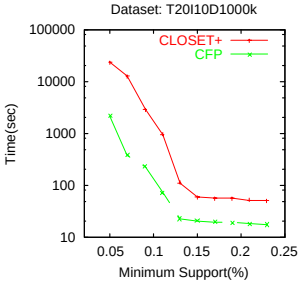
- Compare with CLOSET+ algorithm [kdd03]



BMS-POS



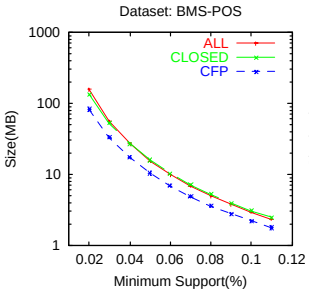
pumsb



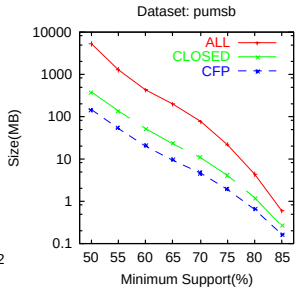
T20I10D1000k

CFP-tree Size

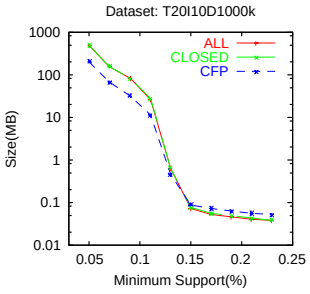
- ALL : all frequent patterns in flat format
- CLOSE: frequent closed patterns in flat format
- CFP : CFP-tree



BMS-POS



pumsb



T20I10D1000k

Conclusion

- We summarized the approaches in frequent itemset mining
- Mining frequent itemsets from very large transactional databases: Search space partitioning
- To support efficient mining of frequent itemsets with different support and containing different items: CFP trees
- Mining frquent itemset and itemsets from data streams – another challenge