Partition algorithm for Association Rules

Sharma Chakravarthy
Information **T**echnology **Lab**oratory (IT Lab)
Computer Science and Engineering Department
The University of Texas at Arlington, Arlington, TX

Email: sharma@cse.uta.edu
URL: http://itlab.uta.edu/sharma

Outline

- Introduction
- Problem Description
- Partition Algorithm
- Performance Comparison
- Parallelization
- Conclusions

Paper

An Efficient Algorithm for Mining Association Rules in Large databases

Ashok Savasere, Ed Omiecinski, and Sham Navathe

Vldb95

You can see how many algorithms have been proposed between 1993 (first paper on association rules) and 1995

Itemset Lattice + Aprori Principle

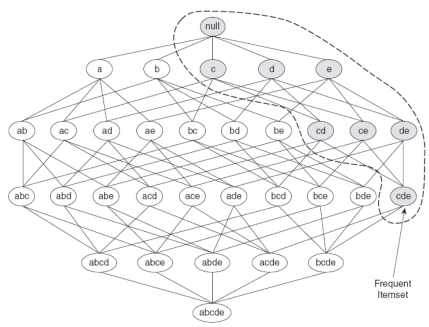


Figure 6.3. An illustration of the *Apriori* principle. If $\{c, d, e\}$ is frequent, then all subsets of this itemset are frequent.

Itemset Lattice + Apriori Principle

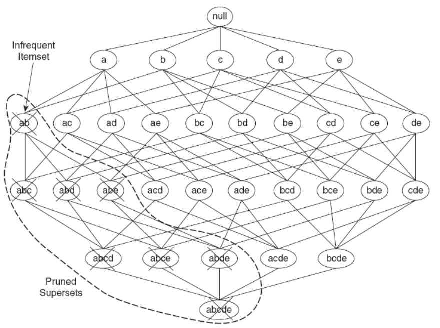


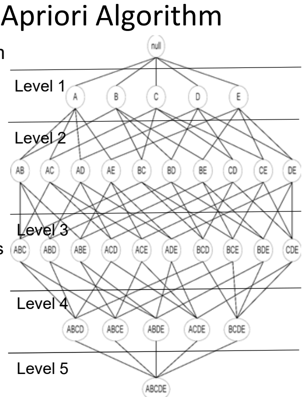
Figure 6.4. An illustration of support-based pruning. If $\{a, b\}$ is infrequent, then all supersets of $\{a, b\}$ are infrequent.

Limitations of Apriori

- Apriori is one of the first algorithms that successfully tackled the exponential size of the frequent itemset space
- Nevertheless the Apriori suffers from two main weaknesses
  - High I/O overhead from the generate-and-test strategy: several passes are required over the database to find the frequent itemsets
  - The performance can degrade significantly on dense databases, as large portion of the itemset lattice becomes frequent

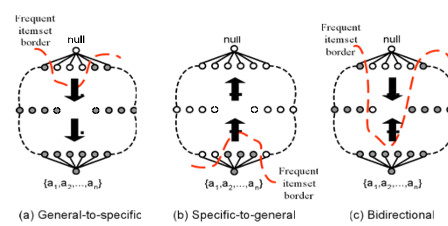
Characteristics of Apriori Algorithm

- Breadth-first search algorithm
  - all frequent itemsets of given size are generated in each pass
  - Starts with high support, works towards low support
- General-to-specific search:
  - start with itemsets with frequent support, work towards lower- support region
- Generate-and-test strategy:
  - generate candidates, test by support counting



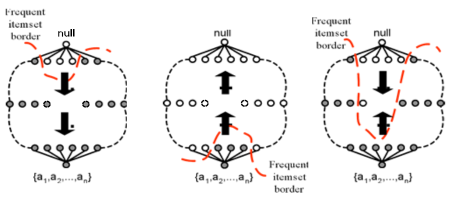
Alternative methods for generating frequent itemsets: Traversal of itemset lattice

- Apriori uses general-to-specific search: start from most highly supported itemsets, work towards lower support region
- Works well if the frequent itemset border is close to the top of the lattice



Alternative methods for generating frequent itemsets: Traversal of itemset lattice

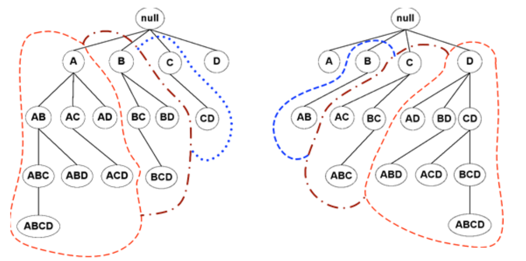
- Specific-to-general search: look first for the most specific frequent itemsets, work towards higher support region
- Works well if the border is close to the bottom of the lattice
  - Dense databases



(a) General-to-specific (b) Specific-to-general (c) Bidirectional

Prefix and Suffix Trees

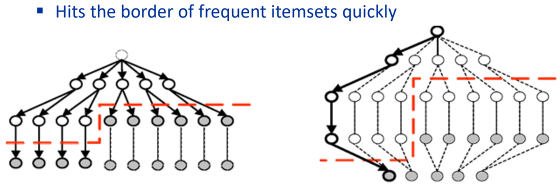
- Left: prefix tree and equivalence classes defined for prefixes of length k=1
- Right: suffix tree and equivalence classes defined for prefixes of length k=1



(a) Prefix tree (b) Suffix tree

Alternative methods for frequent itemsets generation: Breadth-first vs Depth-first

- Apriori traverses the itemset lattice in breadth-first manner
- Alternatively, the lattice can be searched in depth-first manner: extend single itemset until it cannot be extended
  - Often used to find maximal frequent itemsets
  - Hits the border of frequent itemsets quickly



(a) Breadth first (b) Depth first

Preview

- Presents an efficient algorithm for mining association rules that is different from known algorithms (Apriori and AprioriTid)
- Reduces both CPU and I/O overheads significantly.
- Suitable for very large size databases
- Also suitable for parallelization.

Preamble

- Need for Inferring valuable high-level information based on large volume of data.
- Association rules identify set of items that are most often purchased with other set of items.
- Problems with earlier algorithms are large disk I/O, poor response time and poor resource utilization.
- Need to develop fast and efficient algorithms that can handle large volumes of data.

Problem Description

- Generate all rules that have support and confidence greater than min_sup and min_conf
  - Generate all ***large/Frequent*** itemsets that have support above min_sup.
    - Nontrivial problem
      - when number of itemsets are huge.
      - When the database is large
  - Generate all rules that have min_conf for each large/frequent itemset.
    - Much simpler than generating large itemsets.
- How to reduce the number of passes on the database?

Previous Work

- AIS, SETM, Apriori, AprioriTid, and AprioriHybrid
- These algorithms vary mainly by,
  - How the candidate itemsets are generated.
  - How the support for the candidate itemsets are counted (typically by making a pass on the db)
  - Use of efficient memory data structures (e.g., hash tree) to reduce computation
  - Use of buffer management for reducing I/O

Partition Algorithm

- Scans entire database only twice (in two phases)
- Phase 1
  - Divide the database into a number of non-overlapping partitions.
  - Load each partition and process it (can be held in memory)
  - Generate all frequent itemsets for each partition (use partition min_sup)
  - **Merge all these large itemsets into a set of all potentially large/frequent itemsets.**
- Phase 2
  - Count the actual support for these candidate itemsets.
  - Second pass on the database
- Identify all the itemsets that have the minimum global support.

Correctness

- We are generating locally large itemsets in each partition
- Then we are merging local large itemsets
- Then we are making sure that these local large itemsets are indeed **globally large by counting their support again**

- Assume the database has 100 Txs and min_sup is 20
- If it is partitioned into 5 partitions of 20
  - The min_sup for each partition is 4 (partition min_sup)
  - Partition min_sup can be computed for partitions of any size!

Correctness (Contd.)

- Local large itemset is with respect to the local size of the database (or partition size)
- Key concept: Any potential (global) large itemset appears as large itemset **<u>in at least</u>** one of the partitions.
- The above is true independent of how D is partitioned and the number of partitions!
  - Partition sizes do not have to be the same size!

Correctness?

- Take 100 transactions and support as 20 (20%)
- Suppose A, C and AC satisfy the min support. Say they appear in 20 (or more) transactions
- Let us partition the above database into 5 partitions (20 transactions per partition). For an itemset to be locally large, it should appear in at least 4 transactions (why?)
- The claim is that at least one of the 5 partitions will have min_sup for A, c and AC (more than one can have that). *But it cannot be the case that none of them have min support. (why?)*

Example

- 100 transactions and min_sup is 20 (or 20%)
- Let us assume A (25) and AC (20) both satisfy min_sup
- That is, they each appear in at least 20 transactions out of 100
- Take 5 partitions p1 to p5 of size 20 each
- Take A distribution in p's (min_sup is 4)
  - Let p1 contain A in 10 Txs, p2 in 5, p3 in 8, p4 in 2, p5 in 0
  - Now it will be large/frequent in p1, p2, and p3 and not frequent on p4 and p5
- Take AC distribution in p's
  - 3, 3, 3, 3, x x cannot be less than 4
  - 4, 4, 4, 4, 4 is the minimum equal distribution

- If an item occurs N times globally, it has to occur at least ceiling(N/k) times in at least one of the k partitions!
  - N/k is the minimum value of equal distribution

An itemset frequent in a partition but does not have global support

- 100 transactions and min_sup is 30 (or 30%)
- Assume C (25) and CF (20) both do not satisfy min_sup
- That is they each appear in < 30 transaction out of 100
- Take 5 arbitrary partitions p1 to p5 of sizes (partition min_sup) 15 (5), 25 (8), 20 (6), 20 (6) , 20 (6)
- consider C distribution in p's
  - P1 contains 1 C, p2 21 C, p3 0 C, p4 0 C, p5 6 C (total 28)
  - Now it will be frequent in p2 and p5, will be in global set but C is not frequent in the database
- Similarly, consider CF distribution in p's of equal size
  - 20, 0, 0, 0, 0 worst case distribution
  - CF is frequent in p1, but still not globally frequent!

Discovering Rules

- So far all the large itemsets and their supports are determined.
- The association rules can be discovered easily as follows,
  - If *l* is a large itemset, then for every subset *a* of *l*, the ratio *support(l)/support(a)* is computed.
  - If the ratio is at least equal to the minimum confidence, then the rule is,
    - a ➡ (l – a)

Partition Algorithm Contd.

1. *P = partition_database(D)*
2. *n = number of partitions*

//phase I

3. *For i = 1 to n do begin*
4. *read_in_partition ($p_i \in P$)*
5. *L^i = gen_large_itemsets(P_i)* //local candidate itemsets using *Apriori*
6. *End*

// Merge phase

7. *For i = 2, to n*
8. $C_i^G = \cup_{j=1,2,\ldots,n} L_i^j$, *where* $L_i^j \neq \emptyset$ *//global candidate itemsets*
9. *end*

- We have identified frequent itemsets in each partition
  - Not across all partitions

Partition Algorithm Contd.

//phase II

10. For i = 1 to n do begin
11. Read_in_partition (p_i in P) //why do we have to read all partitions again?
12. For all candidates c ∈ C^G gen_final_count(c, p_i)
13. End
14. L^G = {c in C^G | c.count >= minsup} // why do we need to do this?
15. Answer = L^G

Important:

- We retain only frequent/large itemsets from each partition
- For others, we do not keep their counts from each partition (why?)
  - We have count only for those that were frequent in that partition
  - Hence, a second pass for support counting of only frequent itemsets (much smaller)

Partition Algorithm Contd..

- Involves two procedures
- Procedure gen_large_itemsets
- Procedure gen_final_count

Generation of Local Large Itemsets

- Procedure Gen_large_itemsets takes a partition and generates all large itemsets for that partition.
- Prune step eliminates extensions of (k-1) itemsets which are not large from being considered for counting support.
- Unlike previous algorithms, in Partition algorithm each itemsets count is determined immediately when it is generated.
  - Tidlist and intersection operation is used
  - Since each partition is small, this can be maintained in memory
- Sort-merge join algorithm is used

Merging

- Initially empty
- All local large itemsets of the first partition are added to the global candidate set
- For subsequent partitions the local large itemsets are added only if the itemset is not already included
- Hash tables are used for merging efficiently
- Note that itemsets are in sorted order

Phase 2 counting

procedure gen_final_counts (C^G: global candidate set, p: database partition)

forall 1-itemsets do
    generate the tidlist
for (k = 2; C_k^G not empty; k++) do
    forall k-itemset c in C_k^G do
templist = c[1].tidlist ∩ c[2].tidlist ∩ … ∩ c[k].tidlist
        c.count = c.count + | templist|
    end
end

Generation of Final Large Itemsets

- Global candidate set is generated as the **union of all local large itemsets** from all partitions.
- Merge phase generates global large itemsets from the global candidate sets.
- Cumulative count gives the global support for the global itemsets (phase 2)

Performance Comparison

- Better technique for generating the counts than Apriori.
- Use of efficient data structures for computing count.
- Scans the database only twice, so great reduction in disk I/O.
- Improves performance up to a factor of seven
- Reduces number of disc reads by a factor of eight.

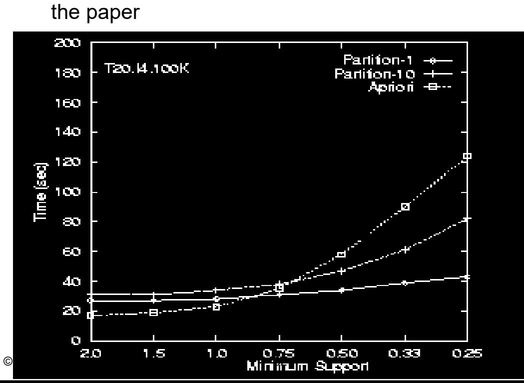
Performance Comparison Contd..

- Good for lower minimum support since reduces number of candidate itemsets.
- As shown in next figure, p Partition algorithm performs better than the Apriori mainly due to reduction in CPU overhead.
- For higher minimum support, Apriori performs slightly better since partition algorithm has a overhead of setting up data structures.

- Effect of data skew

Performance Comparison Contd.

- Please take a look at the performance comparisons in the paper



Parallelization

- Parallel database systems are delivering the performance requirements for very large databases.
- Suitable to augment these parallel databases to provide data mining capability.
- Recall that partitions are processed entirely independently in both the phases of partition algorithms.
- Indicates that the processing can be essentially done in parallel.
  - Parallel algorithms are different from partitioned algorithms
    - Partitioned algorithms can be executed sequentially on each partition!

Parallel Algorithm

1. Generate large itemsets for each processing node's local data independently.
2. Exchange large itemsets at each node with all other nodes.
3. Count support for each itemset in the candidate set with respect to the local data at each node.
4. Send local counts at each node to all other nodes.
5. The global support is the sum of all local supports.

Conclusions

- Fast and efficient for very large databases.
- Both CPU and I/O improvements over Apriori.
- Scans the database at most twice, so huge reduction in I/O overhead.
- Inherent parallelism for use on parallel machines and databases.
- Suited for very large databases in a high data and resource contention environment such as OLTP.

Generalizations

- Handling taxonomies (is-a hierarchies on items)
- Find rules with items at all levels of the taxonomy
- Extensions to apriori
- Optimizations
- Handling numeric and categorical attributes
- Optimal discretization of numeric attributes

Extensions

- Incremental maintenance of frequent sets
  - when to scan the whole database
  - self-maintainable unless the set of frequent itemsets expands
- Rules with item constraints
  - embed the constraints into the discovery
- Maximal association rules

Thank You !!!

For more information visit:
http://itlab.uta.edu

Sequential patterns

- Find frequent sequences (ordered set of items)
- Example: <computer, modem> <printer>
- Input: customer sequences with transaction time
- Timing constraints: window-size, min-gap, max-gap
- Algorithms:
  - Post processing on frequent itemsets
  - GSP algorithm