The Partition Algorithm for mining ARs

- **Apriori scans the database (set of transactions) several times, in order to compute the supports of candidate frequent k-itemsets.**
- **The Partition Algorithm for mining ARs scans the database only twice.**
  - ▷ **First scan: generate a set of all potentially large itemsets**
  - ▷ **Second scan: set up counters for each potentially large itemset and compute their actual supports**
- **During the first scan, a superset of the actual large itemsets is generated. (i.e. false positives may be generated, but no false negatives are generated)**

The Partition Algorithm executes in two phases:

- **Phase I: the algorithm logically divides the database into a number of non-overlapping partitions. The partitions are considered one at a time and all large itemsets for that partition are generated.**
  At the end of phase I, these large itemsets are merged to generate a set of all potentially large itemsets.
- **Phase II: the actual supports for these itemsets are generated and the large itemsets are identified.**

The partition sizes are chosen such that each partition can be accommodated in the main memory so that the partitions are read only once in each phase.

Partition Algorithm Assumptions

- **The transactions are in the format** $\langle TID, i_j, i_k, \ldots, i_n \rangle$
- **The items in a transaction are assumed to be kept sorted in the lexicographic order.**
- **The** $TIDs$ **are monotonically increasing.**
- **The database resides on secondary storage and the approximate size of the database in blocks or pages is known in advance.**

Partition Algorithm Definitions

A Partition p of the database $\mathcal{D}$ is any subset of the transactions contained in $\mathcal{D}$

Any two different partitions are non-overlapping

$$p_i \cap p_j = \emptyset$$

and the union of all partitions must equal $\mathcal{D}$.

The local support for an itemset is the fraction of transactions containing that itemset in a partition.

A local candidate itemset is an itemset, that is being tested for minimum support within a given partition.

A local large itemset is an itemset whose local support in a partition exceeds the minimum threshold for the support.

• A local large itemset may or may not be large in the context of the entire database.

• We define global support, global large itemset, and global candidate itemset as above except they are in the context of the entire database 2).

• The goal is to find all global large itemsets.

Notation:

Individual itemsets are represented by lowercase letters.

Sets of itemsets are represented by uppercase letters.

When there is no ambiguity we omit the partition number when referring to a local itemset.

The notation $c[1] \cdot c[2] \cdots c[k]$ is used to represent a k-itemset c consisting of items $c[1]$, $c[2]$, $\ldots$, $c[k]$.

Outline of the Partition Algorithm

- ▷ Initially the database $\mathcal{D}$ is logically partitioned into n partitions.
- ▷ Phase I of the algorithm takes n iterations.
- ▷ During iteration i only partition p_i is considered.
- ▷ The function `gen_large_itemsets` takes a partition p_i as input and generates local large itemsets of all lengths, $L_1^i, L_2^i, \ldots, L_l^i$ as the output.
- ▷ In the merge phase the local large itemsets of same lengths from all n partitions are combined to generate the global candidate itemsets.
- ▷ In phase II, the algorithm sets up counters for each global candidate itemset, and counts their support for the entire database and generates the global large itemsets.

Complexity: The algorithm reads the entire database once during phase I and once during phase II.

Correctness: Any potential large itemset appears as a large itemset in at least one p_i.

C_k^p	Set of local candidate k–itemsets in partition p
L_k^p	Set of local large k–itemsets in partition p
L^p	Set of all local large itemsets in partition p
C_k^G	Set of global candidate k–itemsets
C^G	Set of all global candidate itemsets
L_k^G	Set of global large k–itemsets

Table 1: Notation

P = partition_database($\mathcal{D}$)
n = Number of partitions
for $i = 1$ to n **begin** // *Phase I*
    read_in_partition($p_i \in P$)
    L^i = gen_large_itemsets(p_i)
end
for ($i = 2$; $L_i^j \neq \emptyset$, $j = 1, 2, \ldots, n$; i++) **do**
    $C_i^G = \cup_{j=1,2,\ldots,n} L_i^j$ // *Merge Phase*
for $i = 1$ to n **begin** // *Phase II*
    read_in_partition($p_i \in P$)
    for all candidates $c \in C^G$ gen_count(c, p_i)
end
$L^G = \{c \in C^G | c.\text{count} \geq \mathit{minSup}\}$

Figure 1: Partition Algorithm

Generation of Local Large Itemsets

- The procedure `gen_large_itemsets` takes a partition and generates all large itemsets (of all lengths) for that partition.
- Lines $3 - 8$ show the candidate generation process.
- The prune step is performed as follows:

  prune(c: k-itemset)

  forall $(k-1)$-subsets s of c do

      if $s \notin L_{k-1}$ then

          return "c can be pruned"
- The prune step eliminates extensions of $(k-1)$- itemsets which are not found to be large, from being considered for counting support.
- Example: if $L_3^p = \{(123), (124), (134), (135), (234)\}$, the candidate generation initially generates the itemsets (1234) and (1345). Itemset (1345) is pruned since (145) is not in L_3^p.
- Same technique as Apriori, except that here, as each candidate itemset is generated, its count is determined immediately.

```
procedure gen_large_itemsets(p: database partition)
L_1^p = {large 1-itemsets along with their tidlists}
for ( k = 2; L_k^p ≠ ∅; k++) do begin
    forall itemsets l_1 ∈ L_{k-1}^p do begin
        forall itemsets l_2 ∈ L_{k-1}^p do begin
            if l_1[1] = l_2[1] ∧ l_1[2] = l_2[2] ∧ ... ∧
                l_1[k-2] = l_2[k-2] ∧ l_1[k-1] < l_2[k-1] then
                c = l_1[1] · l_1[2] ··· l_1[k-1] · l_2[k-1]
                if c cannot be pruned then
                    c.tidlist = l_1.tidlist ∩ l_2.tidlist
                    if |c.tidlist| / |p| ≥ minSup then
                        L_k^p = L_k^p ∪ {c}
        end
    end
end
return ∪_k L_k^p
```

Figure 2: Procedure gen_large_itemsets

Counts for the candidate itemsets

- Associated with every itemset, we define a structure called `tidlist`.
- A `tidlist` for itemset c contains the TIDs of all transactions that contain the itemset c within a given partition.
- The TIDs in a `tidlist` are kept in sorted order.
- The cardinality of the `tidlist` of an itemset divided by the total number of transactions in a partition, gives the (local) support for that itemset, in that partition.
- Initially, the `tidlists` for 1-itemsets are generated directly by reading the partition.
- The `tidlist` for a candidate k-itemset, is generated by joining the `tidlists` of the two $(k-1)$-itemsets that were used to generate the candidate k-itemset.
- Example: the `tidlist` for the candidate 4-itemset (1234) is generated by joining the `tidlists` of 3-itemsets (123) and (124).

Correctness: The candidate generation process correctly produces all potential large candidate itemsets.

Correctness: The intersection of `tidlists` gives the correct support for a k-itemset.

Generation of Final Large Itemsets

- The global candidate set is generated as the union of all local large itemsets from all partitions.
- In phase II of the algorithm, global large itemsets are determined from the global candidate set.
- This phase also takes n (number of partitions) iterations.
- Initially, a counter is set up for each candidate itemset and initialized to 0.
- Next, for each partition, `tidlists` for all 1-itemsets are generated.
- The support for a candidate itemset in that partition is generated by intersecting the `tidlists` of all 1-subsets of that itemset.
- The cumulative count gives the global support for the itemset.
- Procedure gen_final_counts

Correctness: Since the partitions are non-overlapping, a cumulative count over all partitions gives the support for an itemset in the entire database.

```
forall 1-itemsets do
    generate the tidlist
for( k = 2; C_k^G ≠ ∅; k++) do begin
    forall k–itemset c ∈ C_k^G do begin
        templist = c[1].tidlist ∩c[2].tidlist ∩ ... ∩ c[k].tidlist
        c.count = c.count + | templist |
    end
end
```

Figure 3: Procedure gen_final_counts