2 Association Rules and Frequent Itemsets

The *market-basket problem* assumes we have some large number of *items*, e.g., "bread," "milk." Customers fill their market baskets with some subset of the items, and we get to know what items people buy together, even if we don't know who they are. Marketers use this information to position items, and control the way a typical customer traverses the store.

In addition to the marketing application, the same sort of question has the following uses:

1. Baskets = documents; items = words. Words appearing frequently together in documents may represent phrases or linked concepts. Can be used for intelligence gathering.

2. Baskets = sentences, items = documents. Two documents with many of the same sentences could represent plagiarism or mirror sites on the Web.

2.1 Goals for Market-Basket Mining

1. *Association rules* are statements of the form $\{X_1, X_2, \ldots, X_n\} \Rightarrow Y$, meaning that if we find all of $X_1, X_2, \ldots, X_n$ in the market basket, then we have a good chance of finding Y. The probability of finding Y for us to accept this rule is called the *confidence* of the rule. We normally would search only for rules that had confidence above a certain threshold. We may also ask that the confidence be significantly higher than it would be if items were placed at random into baskets. For example, we might find a rule like $\{milk, butter\} \Rightarrow bread$ simply because a lot of people buy bread. However, the beer/diapers story asserts that the rule $\{diapers\} \Rightarrow beer$ holds with confidence siginificantly greater than the fraction of baskets that contain beer.

2. *Causality.* Ideally, we would like to know that in an association rule the presence of $X_1, \ldots, X_n$ actually "causes" Y to be bought. However, "causality" is an elusive concept. nevertheless, for market-basket data, the following test suggests what causality means. If we lower the price of diapers and raise the price of beer, we can lure diaper buyers, who are more likely to pick up beer while in the store, thus covering our losses on the diapers. That strategy works because "diapers causes beer." However, working it the other way round, running a sale on beer and raising the price of diapers, will not result in beer buyers buying diapers in any great numbers, and we lose money.

3. *Frequent itemsets.* In many (but not all) situations, we only care about association rules or causalities involving sets of items that appear frequently in baskets. For example, we cannot run a good marketing strategy involving items that no one buys anyway. Thus, much data mining starts with the assumption that we only care about sets of items with high *support*; i.e., they appear together in many baskets. We then find association rules or causalities only involving a high-support set of items (i.e., $\{X_1, \ldots, X_n, Y\}$ must appear in at least a certain percent of the baskets, called the *support threshold.*

2.2 Framework for Frequent Itemset Mining

We use the term *frequent itemset* for "a set S that appears in at least fraction s of the baskets," where s is some chosen constant, typically 0.01 or 1%.

We assume data is too large to fit in main memory. Either it is stored in a RDB, say as a relation $Baskets(BID, item)$ or as a flat file of records of the form $(BID, item1, item2, \ldots, itemn)$. When evaluating the running time of algorithms we:

- Count the number of passes through the data. Since the principal cost is often the time it takes to read data from disk, the number of times we need to read each datum is often the best measure of running time of the algorithm.

There is a key principle, called *monotonicity* or the *a-priori trick* that helps us find frequent itemsets:

- If a set of items S is frequent (i.e., appears in at least fraction s of the baskets), then every subset of S is also frequent.

To find frequent itemsets, we can:

1. Proceed levelwise, finding first the frequent items (sets of size 1), then the frequent pairs, the frequent triples, etc. In our discussion, we concentrate on finding frequent pairs because:

   (a) Often, pairs are enough.

   (b) In many data sets, the hardest part is finding the pairs; proceeding to higher levels takes less time than finding frequent pairs.

   Levelwise algorithms use one pass per level.

2. Find all *maximal frequent itemsets* (i.e., sets S such that no proper superset of S is frequent) in one pass or a few passes.

2.3 The A-Priori Algorithm

This algorithm proceeds levelwise.

1. Given support threshold s, in the first pass we find the items that appear in at least fraction s of the baskets. This set is called L_1, the frequent items. Presumably there is enough main memory to count occurrences of each item, since a typical store sells no more than 100,000 different items.

2. Pairs of items in L_1 become the *candidate pairs* C_2 for the second pass. We hope that the size of C_2 is not so large that there is not room for an integer count per candidate pair. The pairs in C_2 whose count reaches s are the frequent pairs, L_2.

3. The candidate triples, C_3 are those sets $\{A, B, C\}$ such that all of $\{A, B\}$, $\{A, C\}$, and $\{B, C\}$ are in L_2. On the third pass, count the occurrences of triples in C_3; those with a count of at least s are the frequent triples, L_3.

4. Proceed as far as you like (or the sets become empty). L_i is the frequent sets of size i; C_{i+1} is the set of sets of size $i+1$ such that each subset of size i is in L_i.

2.4 Why A-Priori Helps

Consider the following SQL on a $Baskets(BID, item)$ relation with 10^8 tuples involving 10^7 baskets of 10 items each; assume 100,000 different items (typical of Wal-Mart, e.g.).

```
SELECT b1.item, b2.item, COUNT(*)
FROM Baskets b1, Baskets b2
WHERE b1.BID = b2.BID AND b1.item < b2.item
GROUP BY b1.item, b2.item
HAVING COUNT(*) >= s;
```

Note: s is the support threshold, and the second term of the `WHERE` clause is to prevent pairs of items that are really one item, and to prevent pairs from appearing twice.

In the join $Baskets \bowtie Baskets$, each basket contributes $\binom{10}{2} = 45$ pairs, so the join has 4.5×10^8 tuples.

A-priori "pushes the `HAVING` down the expression tree," causing us first to replace $Baskets$ by the result of

```
SELECT *
FROM Baskets
GROUP by item
HAVING COUNT(*) >= s;
```

If $s = 0.01$, then at most 1000 items' groups can pass the `HAVING` condition. Reason: there are 10^8 item occurrences, and an item needs $0.01 \times 10^7 = 10^5$ of those to appear in 1% of the baskets.

- Although 99% of the items are thrown away by a-priori, we should not assume the resulting *Baskets* relation has only 10^6 tuples. In fact, *all* the tuples may be for the high-support items. However, in real situations, the shrinkage in *Baskets* is substantial, and the size of the join shrinks in proportion to the *square* of the shrinkage in *Baskets*.

2.5 Improvements to A-Priori

Two types:

1. Cut down the size of the candidate sets C_i for $i \geq 2$. This option is important, even for finding frequent pairs, since the number of candidates must be sufficiently small that a count for each can fit in main memory.

2. Merge the attempts to find $L_1, L_2, L_3, \ldots$ into one or two passes, rather than a pass per level.

2.6 PCY Algorithm

Park, Chen, and Yu proposed using a hash table to determine on the first pass (while L_1 is being determined) that many pairs are not possibly frequent. Takes advantage of the fact that main memory is usualy *much* bigger than the number of items. During the two passes to find L_2, the main memory is laid out as in Fig. 1.

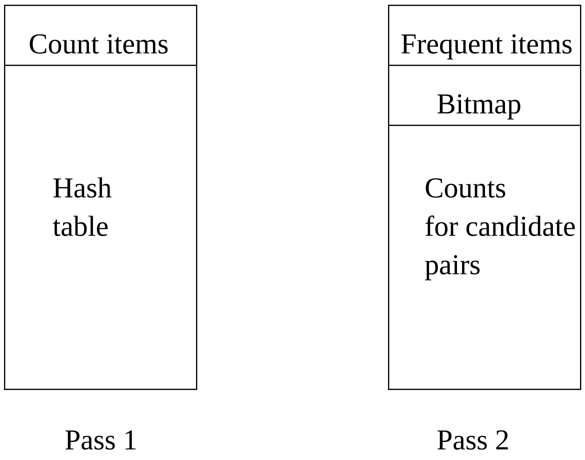

Figure 1: Two passes of the PCY algorithm

Assume that data is stored as a flat file, with records consisting of a basket ID and a list of its items.

1. Pass 1:

   (a) Count occurrences of all items.

   (b) For each bucket, consisting of items $\{i_1, \ldots, i_k\}$, hash all pairs to a bucket of the hash table, and increment the count of the bucket by 1.

   (c) At the end of the pass, determine L_1, the items with counts at least s.

   (d) Also at the end, determine those buckets with counts at least s.

   - Key point: a pair (i, j) cannot be frequent unless it hashes to a frequent bucket, so pairs that hash to other buckets need not be candidates in C_2.

   Replace the hash table by a *bitmap*, with one bit per bucket: 1 if the bucket was frequent, 0 if not.

2. Pass 2:

   (a) Main memory holds a list of all the frequent items, i.e. L_1.

(b) Main memory also holds the bitmap summarizing the results of the hashing from pass 1.

   - Key point: The buckets must use 16 or 32 bits for a count, but these are compressed to 1 bit. Thus, even if the hash table occupied almost the entire main memory on pass 1, its bitmap ocupies no more than 1/16 of main memory on pass 2.

(c) Finally, main memory also holds a table with all the candidate pairs and their counts. A pair (i, j) can be a candidate in C_2 only if *all* of the following are true:

   i. i is in L_1.
   ii. j is in L_1.
   iii. (i, j) hashes to a frequent bucket.

   It is the last condition that distinguishes PCY from straight a-priori and reduces the requirements for memory in pass 2.

(d) During pass 2, we consider each basket, and each pair of its items, making the test outlined above. If a pair meets all three conditions, add to its count in memory, or create an entry for it if one does not yet exist.

- When does PCY beat a-priori? When there are too many pairs of items from L_1 to fit a table of candidate pairs and their counts in main memory, yet the number of frequent buckets in the PCY algorithm is sufficiently small that it reduces the size of C_2 below what can fit in memory (even with 1/16 of it given over to the bitmap).

- When will most of the buckets be infrequent in PCY? When there are a few frequent pairs, but most pairs are so infrequent that even when the counts of all the pairs that hash to a given bucket are added, they still are unlikely to sum to s or more.

2.7 The "Iceberg" Extensions to PCY

1. *Multiple hash tables*: share memory between two or more hash tables on pass 1, as in Fig. 2. On pass 2, a bitmap is stored for each hash table; note that the space needed for all these bitmaps is exactly the same as what is needed for the one bitmap in PCY, since the total number of buckets represented is the same. In order to be a candidate in C_2, a pair must:

   (a) Consist of items from L_1, and
   (b) Hash to a frequent bucket in *every* hash table.

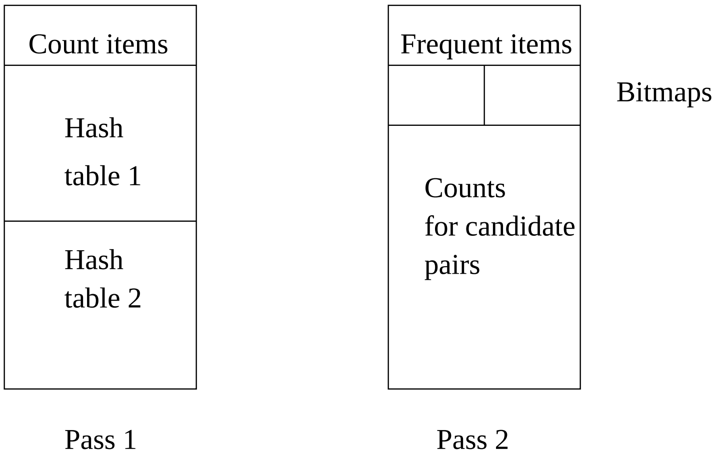

Figure 2: Multiple hash tables memory utilization

2. Iterated hash tables *Multistage*: Instead of checking candidates in pass 2, we run another hash table (different hash function!) in pass 2, but we only hash those pairs that meet the test of PCY; i.e., they are both from L_1 and hashed to a frequent bucket on pass 1. On the third pass, we keep bitmaps from both hash tables, and treat a pair as a candidate in C_2 only if:

   (a) Both items are in L_1.

   (b) The pair hashed to a frequent bucket on pass 1.

   (c) The pair also was hashed to a frequent bucket on pass 2.

   Figure 3 suggests the use of memory. This scheme could be extended to more passes, but there is a limit, because eventually the memory becomes full of bitmaps, and we can't count any candidates.

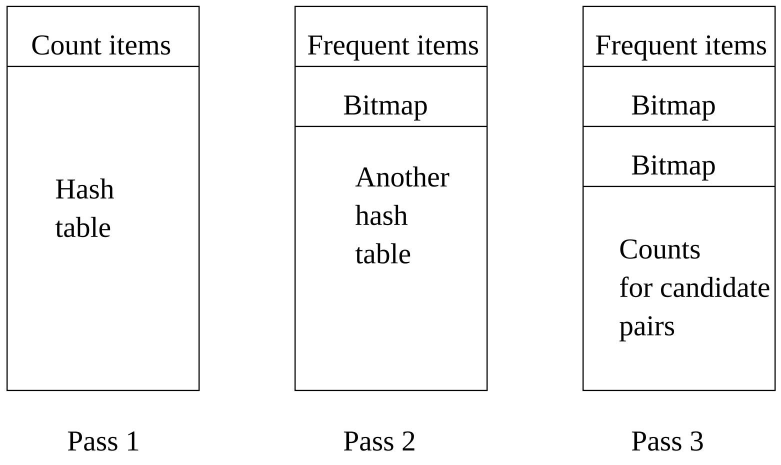

Figure 3: Multistage hash tables memory utilization

- When does multiple hash tables help? When most buckets on the first pass of PCY have counts *way* below the threshold s. Then, we can double the counts in buckets and still have most buckets below threshold.
- When does multistage help? When the number of frequent buckets on the first pass is high (e.g., 50%), but not all buckets. Then, a second hashing with some of the pairs ignored may reduce the number of frequent buckets significantly.

2.8 All Frequent Itemsets in Two Passes

The methods above are best when you only want frequent pairs, a common case. If we want all maximal frequent itemsets, including large sets, too many passes may be needed. There are several approaches to getting all frequent itemsets in two passes or less. They each rely on randomness of data in some way.

1. *Simple approach*: Taka a main-memory-sized sample of the data. Run a levelwise algorithm in main memory (so you don't have to pay for disk I/O), and hope that the sample will give you the truly frequent sets.
   - Note that you must scale the threshold s back; e.g., if your sample is 1% of the data, use $s/100$ as your support threshold.
   - You can make a complete pass through the data to verify that the frequent itemsets of the sample are truly frequent, but you will miss a set that is frequent in the whole data but not in the sample.
   - To minimize false negatives, you can lower the threshold a bit in the sample, thus finding more candidates for the full pass through the data. Risk: you will have too many candidates to fit in main memory.

2. *SON95* (Savasere, Omiecinski, and Navathe from 1995 VLDB; referenced by Toivonen). Read subsets of the data into main memory, and apply the "simple approach" to discover candidate sets. Every basket is part of one such main-memory subset. On the second pass, a set is a candidate if it was identified as a candidate in any one or more of the subsets.

   - Key point: A set cannot be frequent in the entire data unless it is frequent in at least one subset.

3. *Toivonen's Algorithm*:

   (a) Take a sample that fits in main memory. Run the simple approach on this data, but with a threshold lowered so that we are unlikely to miss any truly frequent itemsets (e.g., if sample is 1% of the data, use $s/125$ as the support threshold).

   (b) Add to the candidates of the sample the *negative border*: those sets of items S such that S is *not* identified as frequent in the sample, but *every* immediate subset of S is. For example, if $ABCD$ is not frequent in the sample, but all of ABC, ABD, ACD, and BCD are frequent in the sample, then $ABCD$ is in the negative border.

   (c) Make a pass over the data, counting all the candidate itemsets and the negative border. If no member of the negative border is frequent in the full data, then the frequent itemsets are exactly those candidates that are above threshold.

   (d) Unfortunately, if there is a member of the negative border that turns out to be frequent, then we don't know whether some of its supersets are also frequent, so the whole process needs to be repeated (or we accept what we have and don't worry about a few false negatives).