Big Data Analytics-Selected Topics (60-475)

Frequent Itemset Mining and Association Rules

Lecture 4

School of Computer Science
Faculty of Science
University of Windsor

Dr. Mehdi Kargar
Winter 2017

Outline

- In last lecture, we talked about A-Priori algorithm for finding frequent itemsets
- In this lecture, we focus on how to find frequent **pairs** even more efficiently
  - First, we have a quick overview of hash tables

Hash Function

- A **hash function** is a function that:
  - When applied to a key, returns an integer, between 0 to N-1
    - Key could be one integer, two integers, or even five integers
    - Key could also be a String
  - When applied to equal keys, returns the same number
  - When applied to unequal keys, is unlikely to return the same number
- Hash functions are very important for searching, that is, looking things up fast

Hash Function

- Consider the problem of **searching** an array for a given value
  - If the array is **not sorted**, the search requires **O(n)** time
  - If the array is **sorted**, we can do a binary search which requires **O(log n)** time
  - Can we do better?
    - How about an **O(1)** time?
    - O(1) is constant time!

Hash Function

- Suppose we have a **magical function** that, given a **key** to search for, it tells us exactly where in the array to look for the key
  - If it is in that location, it is in the array
  - If not, then it is not in the array
- That is the only purpose of this function
- This function is called a **hash function**

Example

- Suppose we have this hash function:
  - hashCode("apple") = 5
    hashCode("watermelon") = 3
    hashCode("grapes") = 8
    hashCode("orange") = 7
    hashCode("blueberry") = 0
    hashCode("strawberry") = 9
    hashCode("mango") = 6
    hashCode("banana") = 2

0	blueberry
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	orange
8	grapes
9	strawberry

Example - Collision

- Suppose we have this hash function:
  - hashCode("apple") = 5
    hashCode("watermelon") = 3
    hashCode("grapes") = 8
    hashCode("orange") = 7
    hashCode("blueberry") = 0
    hashCode("strawberry") = 9
    hashCode("mango") = 6
    hashCode("banana") = 2

    hashCode("kiwi") = 6
- There are different ways to deal with the collision in a hash table

0	blueberry
1	
2	banana
3	watermelon
4	
5	apple
6	mango, **kiwi**
7	orange
8	grapes
9	strawberry

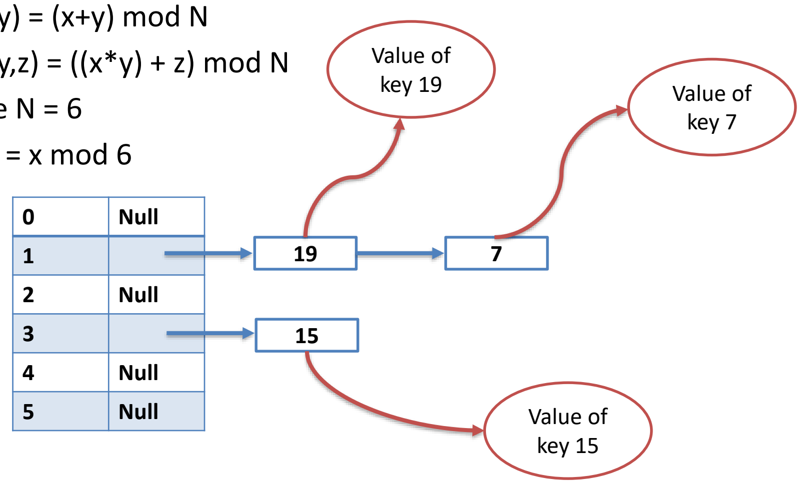
Set vs. Map

- Sometimes we just want to store a set of keys
  - Keys (objects) are either in the set or not
- Sometimes we want a map
  - To look up for one object based on the value of its key
- We use a **key** to find the place in the map
- The associated value is the information we want to look up
- Hashing works the same for sets and maps

. . .	*key*	*value*
141		
142	James	James info
143	sparrow	sparrow info
144	BMW	BMW info
145	seagull	seagull info
146		
147	bluejay	bluejay info
148	owl	owl info

Hash Function for Integers

- A hash function **h** maps keys to integers in a fixed interval **[0, N - 1]**
  - h(x) = x mod N
  - h(x,y) = (x+y) mod N
  - h(x,y,z) = ((x*y) + z) mod N
- Assume N = 6
  - h(x) = x mod 6



Example

- Assume we want to keep the **counts** of **pairs [i,j]**, but we do not have space in memory for all pairs
- We can put each pair in a bucket
  - We do have space in memory for all buckets
    - **Number of Buckets << Number of Pairs**
  - We use a hash function **h(i,j)**, to find the bucket for each pair
- Number of buckets (size of hash table) is 6 (N = 6)
- **h(i,j) = (i+j) mod 6**
- Here is our pairs:
  - [1,4] h(1,4) = 5%6 = 5
  - [3,5] h(3,5) = 8%6 = 2
  - [1,4] h(1,4) = 5%6 = 5
  - [2,3] h(2,3) = 5%6 = 5
  - [1,4] h(1,4) = 5%6 = 5
  - [2,6] h(2,6) = 8%6 = 2
  - [1,2] h(1,2) = 3%6 = 3
  - [2,7] h(2,7) = 9%6 = 3
  - [1,3] h(1,3) = 4%6 = 4
  - [1,4] h(1,4) = 5%6 = 5

	Count
0	0
1	0
2	2
3	2
4	1
5	5

Back to finding frequent itemsets

How to improve A-Priori?

PCY (Park-Chen-Yu) Algorithm

- **Observation:**
  In pass 1 of A-Priori, most **memory** is **idle**
  - We store only individual item counts
  - **Can we use the idle memory to reduce memory required in pass 2?**
- **Pass 1 of PCY:** In addition to item counts, maintain a hash table with **as many buckets as fit in memory** **(why the maximum number of buckets that fits in memory?)**
  - Keep a **count** for each bucket into which **pairs** of items are hashed
    - **For each bucket just keep the count, not the actual pairs that hash to the bucket!**

PCY Algorithm – First Pass

nested for loop!

```
FOR (each basket) :
    FOR (each item in the basket) :
        add 1 to item's count;
    FOR (each pair of items) :
        hash the pair to a bucket;
        add 1 to the count for that bucket;
```

New in PCY

- **Few things to note:**
  - Pairs of items need to be generated from the input file; they are not present in the file
  - We are not just interested in the presence of a pair, but we need to see whether it is present at least ***s*** (support) times

Observations about Buckets

- **Observation: If a bucket contains a frequent pair, then the bucket is surely frequent**
- However, even without any frequent pair, a bucket can still be frequent ☹
  - So, we cannot use the hash to eliminate any member (pair) of a "frequent" bucket
- But, for a bucket with **total count less than *s*, none of its pairs can be frequent** ☺
  - Pairs that hash to this bucket can be eliminated as candidates (**even if the pair consists of 2 frequent items**)
- **Pass 2:**
  Only count pairs that hash to frequent buckets

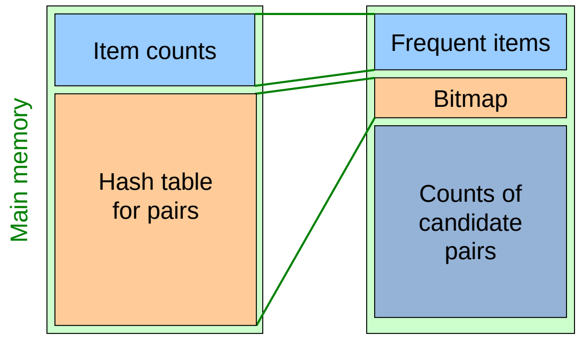
PCY Algorithm – Between Passes

- **Replace the buckets by a bit-vector:**
  - **1** means the bucket count exceeded the support s (call it a **frequent bucket**); **0** means it did not
- **4-byte integer counts are replaced by bits, so the bit-vector requires 1/32 of memory**
- Also, decide which items are frequent and list them for the second pass

PCY Algorithm – Pass 2

- Count all pairs $\{i, j\}$ that meet the conditions for being a **candidate pair**:
  1. Both i and j are frequent items
  2. The pair $\{i, j\}$ hashes to a bucket whose bit in the bit vector is **1** (i.e., a **frequent bucket**)
- **Both conditions are necessary for the pair to have a chance of being frequent**

Main-Memory: Picture of PCY



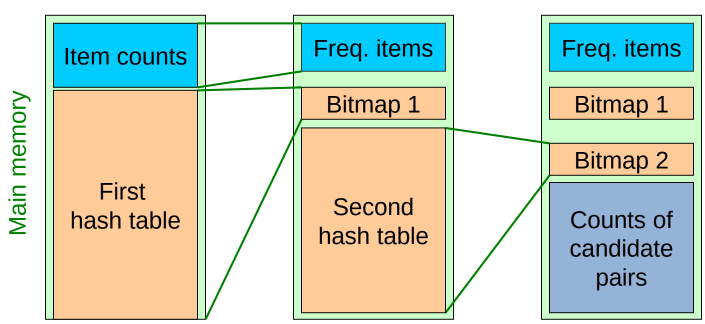
Main-Memory Details

- **Buckets require a few bytes each:**
  - **Note:** we do not have to count past *s*
  - #buckets is *O(main-memory size)*
- On second pass, a table of (item, item, count) **triples** is essential
- In **PCY**, we **cannot** use **triangular matrix** approach
  - **why?** (this is an important why!)
  - Thus, hash table **must eliminate approximately 2/3** of the candidate pairs for PCY to **beat** A-Priori
    - **why?**

Refinement: Multistage Algorithm

- **Limit the number of candidates to be counted**
  - **Remember:** Memory is the bottleneck
  - Still need to generate all the itemsets but we only want to count/keep track of the ones that are frequent
- **Key idea:** After Pass 1 of PCY, rehash only those pairs that **qualify** for Pass 2 of PCY
  - ***i*** and ***j*** are frequent, and
  - ***{i, j}*** hashes to a frequent bucket from **Pass 1**
- On middle pass, fewer pairs contribute to buckets, so fewer ***false positives***
- **Requires 3 passes over the data**

Main-Memory: Multistage



Pass 1

Count items
Hash pairs {i,j}

Pass 2

Hash pairs {i,j}
into Hash2 iff:
i,j are frequent,
{i,j} hashes to
freq. bucket in B1

Pass 3

Count pairs {i,j} iff:
i,j are frequent,
{i,j} hashes to
freq. bucket in B1
{i,j} hashes to
freq. bucket in B2

Multistage – Pass 3

- **Count only those pairs {i, j} that satisfy these candidate pair conditions:**
  1. Both i and j are frequent items
  2. Using the **first hash function**, the pair hashes to a bucket whose bit in the first bit-vector is **1**
  3. Using the **second hash function**, the pair hashes to a bucket whose bit in the second bit-vector is **1**

Important Points

1. **The two hash functions have to be independent**
2. **We need to check both hashes on the third pass**
   - If not, we would end up counting pairs of items that hashed first to an infrequent bucket but happened to hash second to a frequent bucket

Example - Multistage

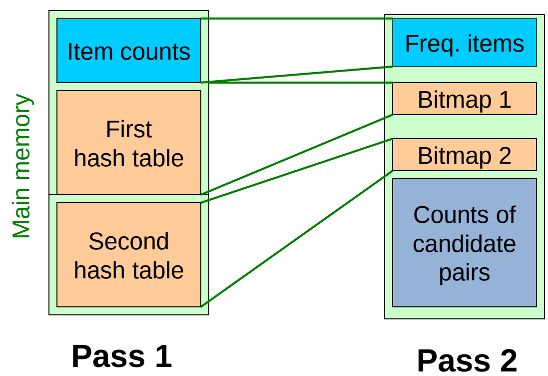
- Assume we have two hash functions **h1** and **h2**
- **h1** maps **only** pairs (1,2) and (3,6) to **bucket #5** in the first hash table
- **h2** maps **only** pairs (1,2) and (8,9) to **bucket #7** in the second hash table
- Here are frequency of each pair:
  - freq(1,2) = 50, freq(3,6) = 50, freq(8,9) = 400
- Assume the support threshold **s** is set to **200**
- In the **first hash table**, the frequency of **bucket #5** is **100**
  - **bucket #5 is not frequent in the first hash table**
- In the **second hash table**, the frequency of **bucket #7** is **400**
  - **bucket #7 is frequent in the second hash table**
- **If we only check the second hash table, we would count pair (1,2) although we shouldn't have!**

Refinement: Multihash

- **Key idea:** Use several independent hash tables on the first pass
- The **danger** of using two hash tables on one pass is that each hash table has half as many buckets as the one large hash table of PCY
  - We have to be sure most buckets will still not reach count s
- If so, we can get a benefit like multistage, but in only 2 passes

Big Data Analytics

Main-Memory: Multihash



25

Big Data Analytics

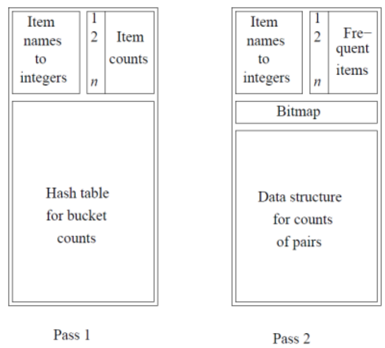
PCY: Extensions

- Either **multistage** or **multihash** can use **more than two hash functions**
- In **multistage**, there is a point of diminishing returns, since the **bit-vectors eventually consume all of main memory**
- For **multihash**, the bit-vectors occupy exactly what one PCY bitmap does, but **too many hash functions makes all counts $\geq s$**

26

Quiz I

- Describe how the bitmap is used in **PCY** algorithm
- Why is the hash map in main memory from Pass 1 transformed into a bitmap in **PCY** algorithm?



Answer to Quiz I

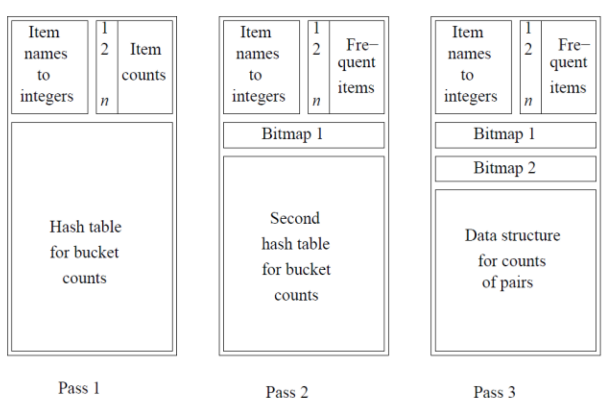
- Describe how the bitmap is used in **PCY** algorithm
- Why is the hash map in main memory from Pass 1 transformed into a bitmap in **PCY** algorithm?
- Answer:
  - Between the passes of PCY, the hash table is summarized as a bitmap, with one bit for each bucket. The bit is 1 if the bucket is frequent and 0 if not.
  - Thus, integers of 32 bits are replaced by single bits, and the bitmap shown in the second pass in the figure takes up only 1/32 of the space that would otherwise be available to store counts.

Quiz II

- Describe the key idea behind the **multistage** algorithm

Answer to Quiz II

- Describe the key idea behind the **multistage** algorithm
- **Answer**: the multistage algorithm uses additional hash tables to reduce the number of candidate pairs



Quiz III

- What can potentially go wrong if instead of 3 passes, 100 passes are used in **multistage** algorithm?

Answer to Quiz III

- What can potentially go wrong if instead of 3 passes, 100 passes are used in **multistage** algorithm?
- **Answer**: We may run out of memory as 100 bit-vectors have to be stored. Thus, we may not have enough space to keep data structures for counts of pairs.