Mining Massive Datasets

Lecture 10

Artur Andrzejak
http://pvs.ifi.uni-heidelberg.de

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Final Exam A – **Final Data!**

- Finals A:
  - Date: 21. February 2022 (Monday)
    - Last week of the semester
  - Time: 14:30 - 16:30 CET (~ 90 minutes for the exam)
  - Location
    - INF 230 gHS + SR INF 205
- Finals B:
  - There will be also finals B at the start of the SoSe 2023
  - You can choose to participate at either finals A or B
- Conditions
  - No books, scripts, computer, smartphone etc. („Es sind keine Hilfsmittel zugelassen“)
  - Please bring your photo ID (Personalausweis / Pass etc.)

Note on Slides

A substantial part of these slides come (either verbatim or in a modified form) from the book
Mining of Massive Datasets
by Jure Leskovec, Anand Rajaraman, Jeff Ullman (Stanford University).
For more information, see the website accompanying the book: http://www.mmds.org.

Current Topic

High dim. data
- Locality sensitive hashing
- Clustering
- Dimensio-nality reduction

Graph data
- PageRank, SimRank
- Community Detection
- Spam Detection

Infinite data
- Filtering data streams
- Web advertising
- Queries on streams

Machine learning
- SVM
- Decision Trees
- Perceptron, kNN

Apps
- Recommender systems
- **Association Rules**
- Duplicate document detection

Programming in Spark & MapReduce

Recall: The Market-Basket Model

- A large set of **items**
  - e.g., things sold in a supermarket
- A large set of **baskets**
  - Each basket is a small subset of items
  - E.g., the things one customer buys on one day
- Goal: discover **association rules**
  - People who bought {x,y,z} tend to buy {v,w}
    - Amazon!

Input:

TID	*Items in a basket*
1	Bread, Coke, Milk
2	Beer, Bread
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Coke, Diaper, Milk

Output:

Rules Discovered:

{Milk} --> {Coke}
{Diaper, Milk} --> {Beer}

Recall: Frequent Itemsets

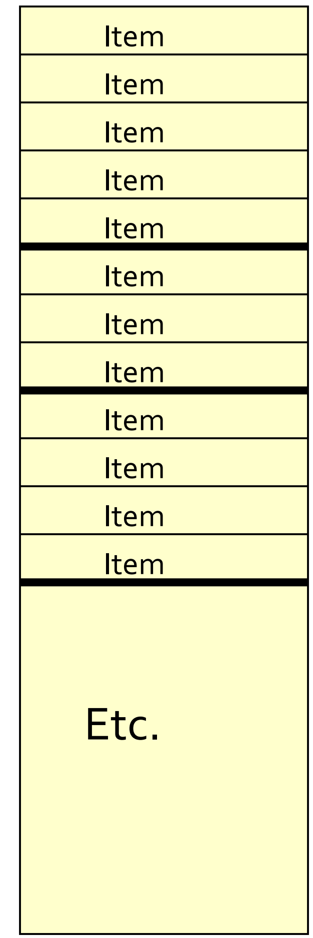
- **Simplest question:** Find sets of items that appear together "frequently" in baskets
- **Support** for itemset ***I***: Number of baskets containing all items in ***I***
  - (Often expressed as a fraction of the total number of baskets)
- Given a ***support threshold s***, ***frequent itemsets*** are sets of items that appear in at least ***s*** baskets

TID	*Items*
1	Bread, Coke, Milk
2	Beer, Bread
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Coke, Diaper, Milk

Support of {Beer, Bread} = 2

Itemsets: Computation Model

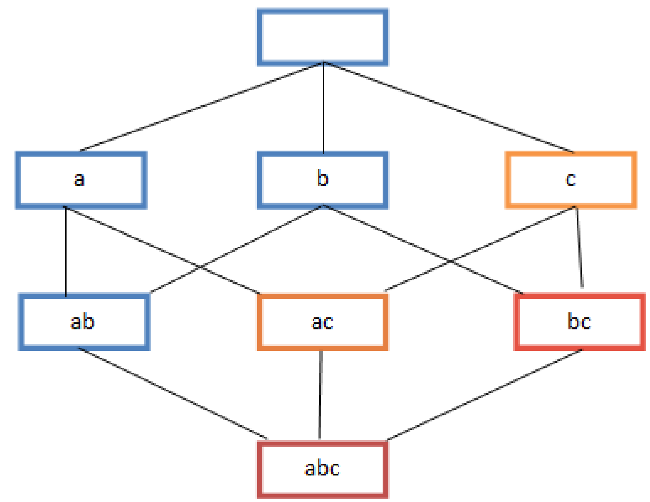
- **Back to finding frequent itemsets**
- Typically, data is kept in flat files rather than in a database system:
  - Stored on disk
  - Stored basket-by-basket
  - Baskets are **small** but we have many baskets and many items
    - Expand baskets into pairs, triples, etc. as you read baskets
    - Use ***k*** nested loops to generate all sets of size ***k***



Items are positive integers, and boundaries between baskets are –1.

Note: We want to find frequent itemsets. To find them, we have to count them. To count them, we have to generate them.

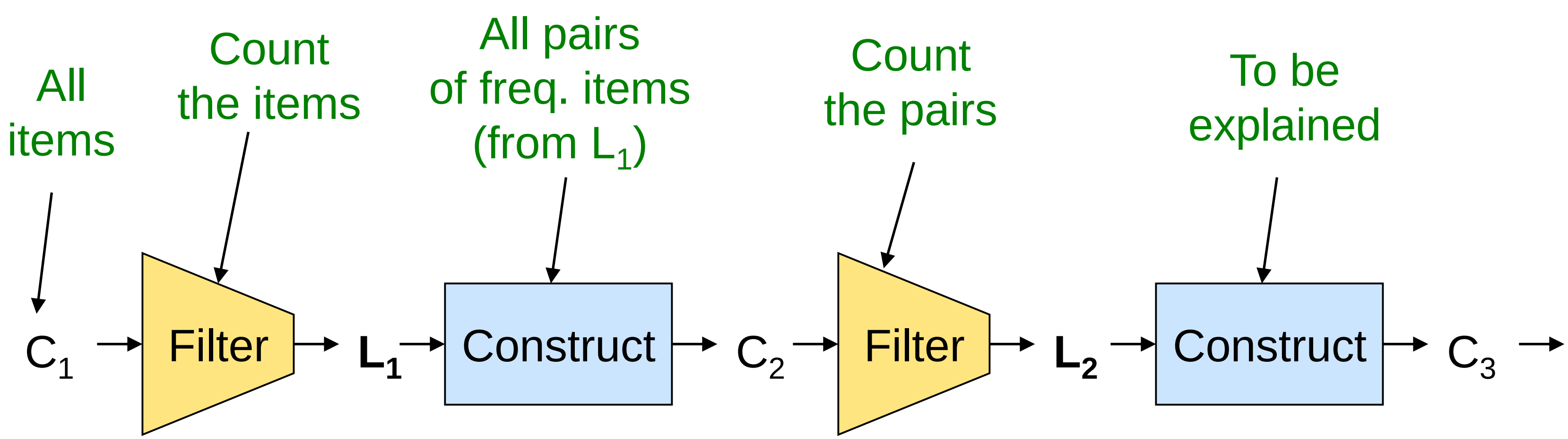
Monotonicity and A-Priori Algorithm



- A **two-pass** approach called ***A-Priori*** limits the need for main memory
- **Key idea: monotonicity**
  - If a set of items I appears at least s times, so does every **subset** J of I
- **Contrapositive for pairs:**
  If item i does not appear in s baskets, then no pair including i can appear in s baskets
- For pairs: find pairs by counting all candidate pairs of frequent singletons

Frequent Triples, Etc.

- **For each *k*, we construct two sets of *k-tuples*** (sets of size *k*):
  - ***C_k*** = ***candidate k-tuples*** = those that might be frequent sets (support $\geq$ **s**) based on information from the pass for ***k*–1**
  - ***L_k*** = the set of truly frequent ***k***-tuples



Recall: PCY (Park-Chen-Yu) Algorithm

PCY (Park-Chen-Yu) Algorithm

- **Observation:**
  In pass 1 of A-Priori, most memory is idle
  - We store only individual item counts
- => Use this idle RAM to reduce RAM used in pass 2!
- **Pass 1 of PCY:** In addition to item counts, maintain a **hash table h** with as many buckets as fit 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!)*
  - *Why? If a pair p is frequent, "its" bucket $\boldsymbol{h}(p)$ will receive count above a threshold s for frequent pairs!*
- **Pass 2 of PCY:**
  Count only those pairs that hash to frequent buckets

PCY Algorithm – Pass 1

```
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

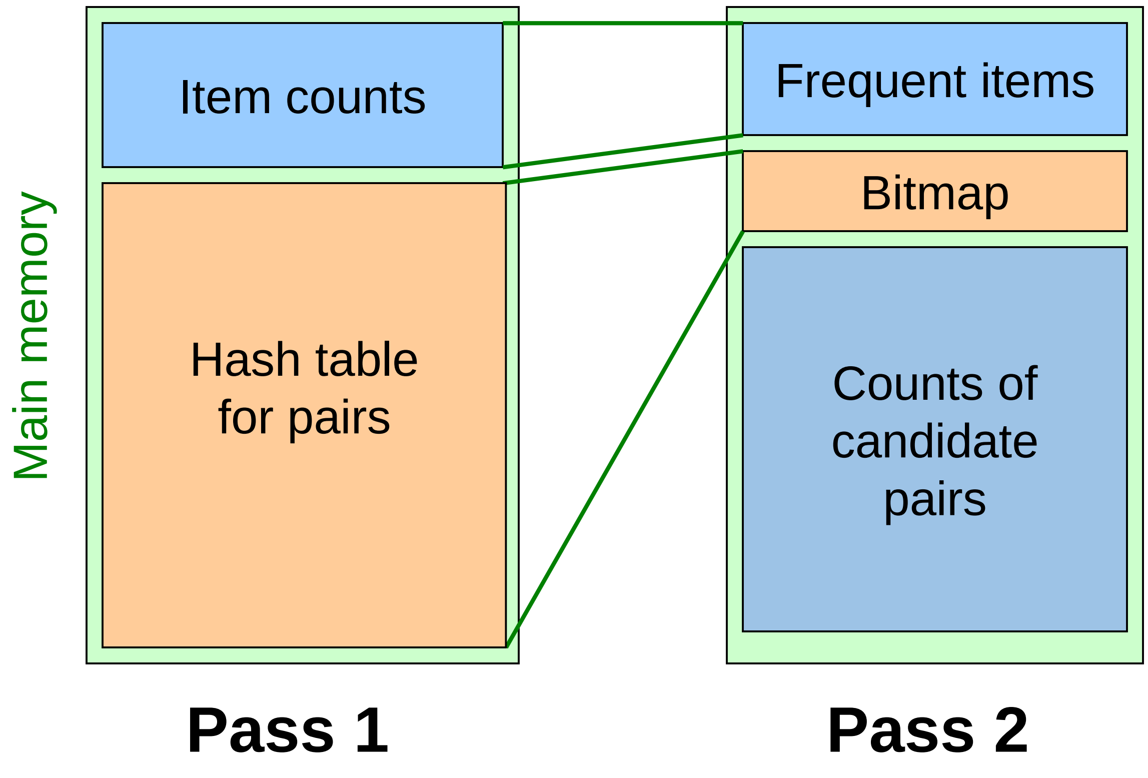
- **Few things to note:**
  - Pairs of items need to be generated from the input file; they are not present in the file
  - Hash function should have many buckets, i.e. must be likely to hash *different pairs to different buckets*

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



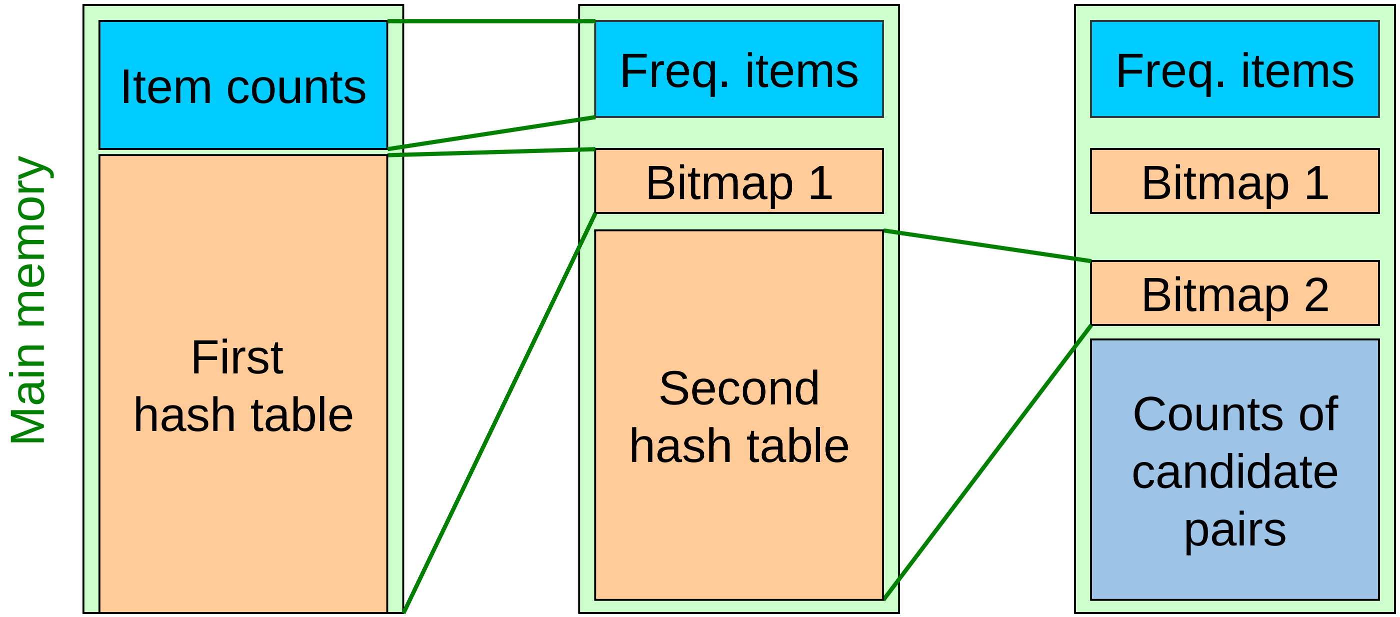
PCY (Park-Chen-Yu) Algorithm Refinement:

Multistage Algorithm

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**

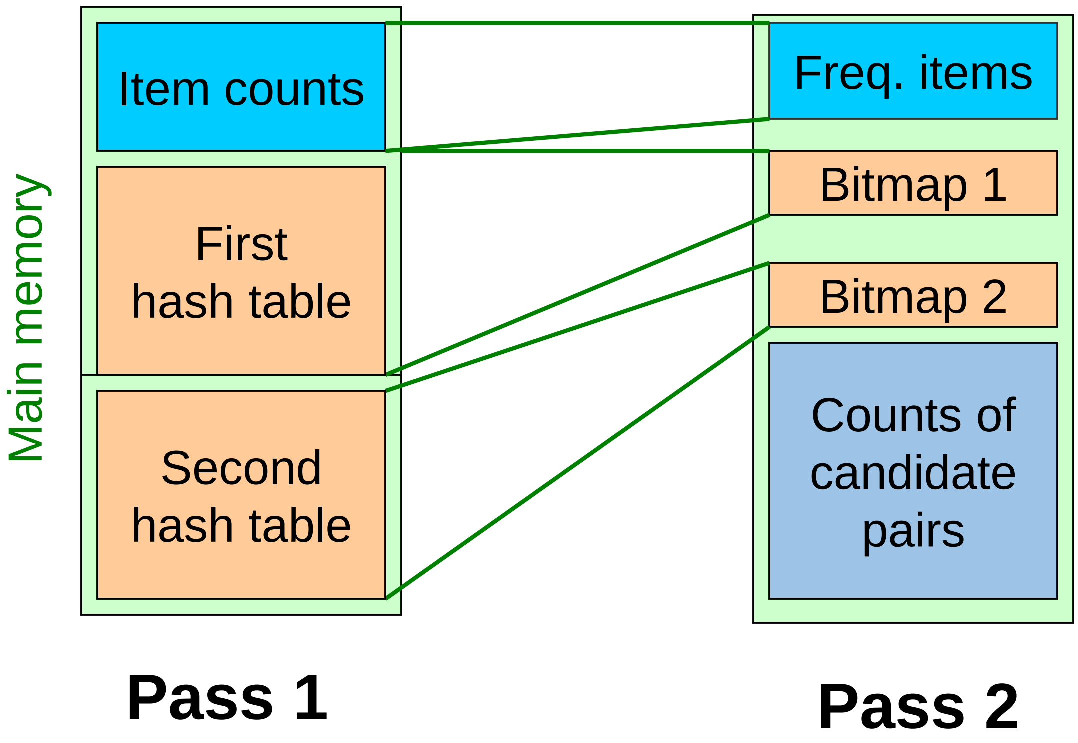
Important Points

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 frequent items that hashed first to an infrequent bucket but happened to hash second to a frequent bucket

Refinement: Multihash

- **Key idea:** Use several independent hash tables on the first pass
- **Risk:** Halving the number of buckets doubles the average count
  - 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

Main-Memory: Multihash



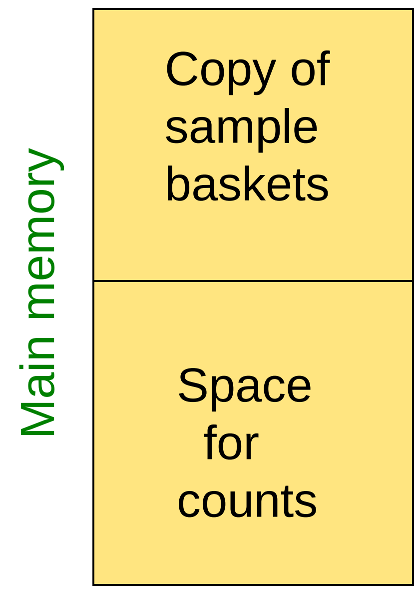
PCY (Park-Chen-Yu): Further Extensions - Frequent Itemsets in < 2 Passes

Frequent Itemsets in ≤ 2 Passes

- A-Priori, PCY, etc., take *k* passes to find frequent itemsets of size *k*
- **Can we use fewer passes?**
- Use 2 or fewer passes for all sizes, but may miss some frequent itemsets
  - Random sampling
  - SON (Savasere, Omiecinski, and Navathe)
  - More in a textbook

Random Sampling (1)

- Take a random sample of the market baskets
- Run a-priori or one of its improvements in main memory
  - So we don’t pay for disk I/O each time we increase the size of itemsets
  - Reduce support threshold proportionally to match the sample size



Random Sampling (2)

- Optionally, verify that the candidate pairs are truly frequent in the entire data set by a second pass (avoid false positives)

- But you don't catch sets frequent in the whole but not in the sample
  - Smaller threshold, e.g., $s/125$, helps catch more truly frequent itemsets
    - But requires more space

SON Algorithm – (1)

- Repeatedly read small subsets of the baskets into main memory and run an in-memory algorithm to find all frequent itemsets
  - Note: we are not sampling, but processing the entire file in memory-sized chunks
- An itemset becomes a candidate if it is found to be frequent in *any* one or more subsets of the baskets.

SON Algorithm – (2)

- On a **second pass**, count all the candidate itemsets and determine which are frequent in the entire set
- **Key "monotonicity" idea:** an itemset cannot be frequent in the entire set of baskets unless it is frequent in at least one subset

SON – Distributed Version

- SON lends itself to distributed data mining
- Baskets distributed among many nodes
  - Compute frequent itemsets at each node
  - Distribute candidates to all nodes
  - Accumulate the counts of all candidates

PCY: Extensions - Summary

- 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***

Thank you.

Questions?