Data Mining Techniques

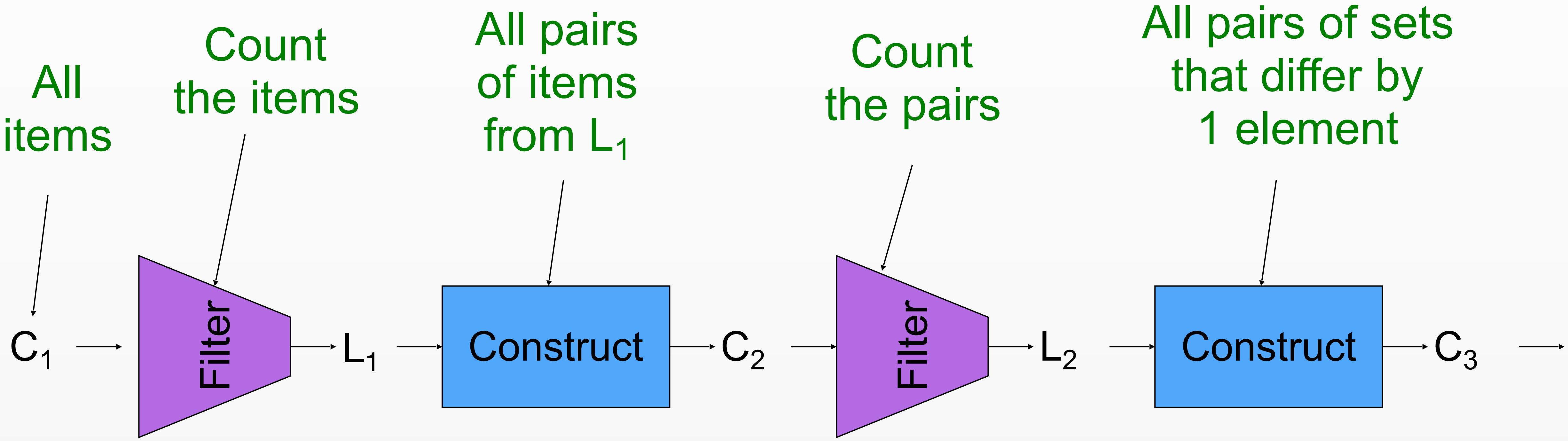
CS 6220 - Section 3 - Fall 2016

Lecture 16: Association Rules

Jan-Willem van de Meent
(credit: Yijun Zhao, Yi Wang,
Tan et al., Leskovec et al.)

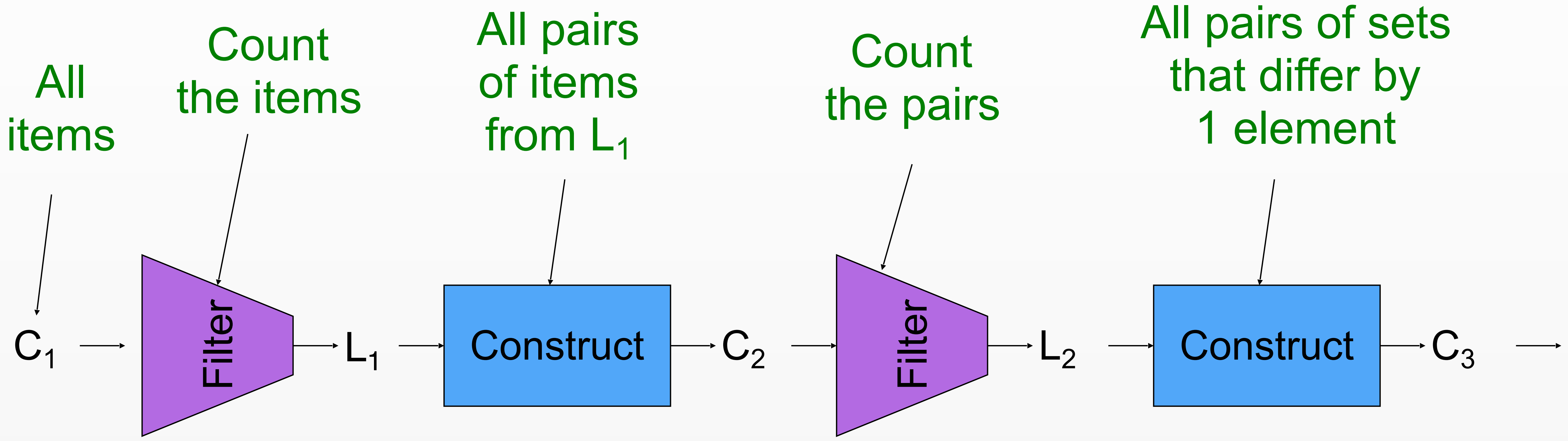
NORTHEASTERN UNIVERSITY
BOSTON, MASSACHUSETTS
FOUNDED AD 1898

Apriori: Summary



```
1.  Set $k = 0$
2.  Define $C_1$ as all size 1 item sets
3.  While $C_{k+1}$ is not empty
4.      Set $k = k + 1$
5.      Scan DB to determine subset $L_k \subseteq C_k$
        with support $\geq s$
6.      Construct candidates $C_{k+1}$ by combining
        sets in $L_k$ that differ by 1 element
```

Apriori: Bottlenecks



1. Set $k = 0$
2. Define C_1 as all size 1 item sets
3. ***While C_{k+1} is not empty***
4.   Set $k = k + 1$
5.   Scan DB to determine subset $L_k \subseteq C_k$ with support $\geq s$ (I/O limited)
6.   **Construct candidates C_{k+1} by combining sets in L_k that differ by 1 element** **(Memory limited)**

Apriori: Main-Memory Bottleneck

- For many frequent-itemset algorithms, main-memory is the critical resource
  - As we read baskets, we need to count something, e.g., occurrences of pairs of items
  - The number of different things we can count is limited by main memory
  - For typical market-baskets and reasonable support (e.g., 1%), k = 2 requires most memory
  - Swapping counts in/out is a disaster (why?)

Counting Pairs in Memory

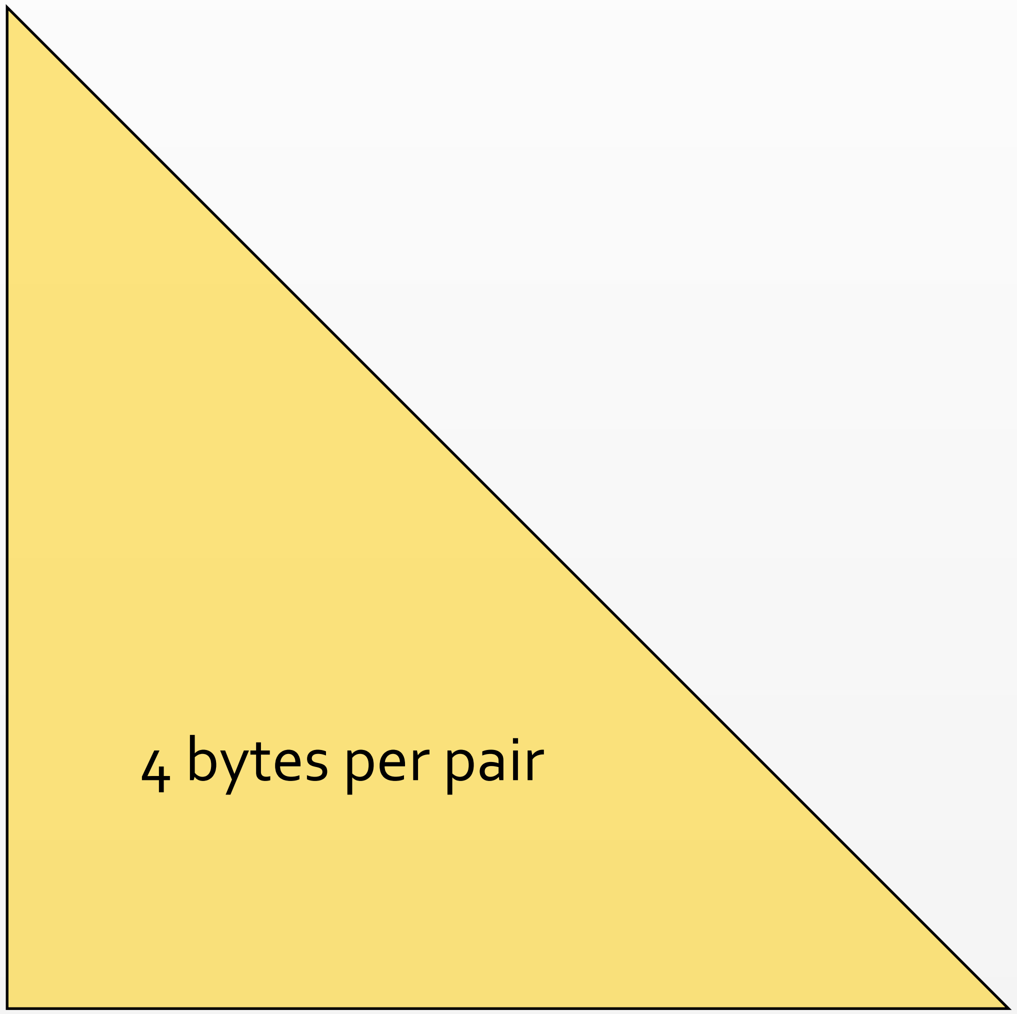
Two approaches:

- Approach 1: Count all pairs using a matrix
- Approach 2: Keep a table of triples $[i, j, c]$ = "the count of the pair of items $\{i, j\}$ is c."
  - If integers and item ids are 4 bytes, we need approximately 12 bytes for pairs with count > 0
  - Plus some additional overhead for the hashtable

Note:

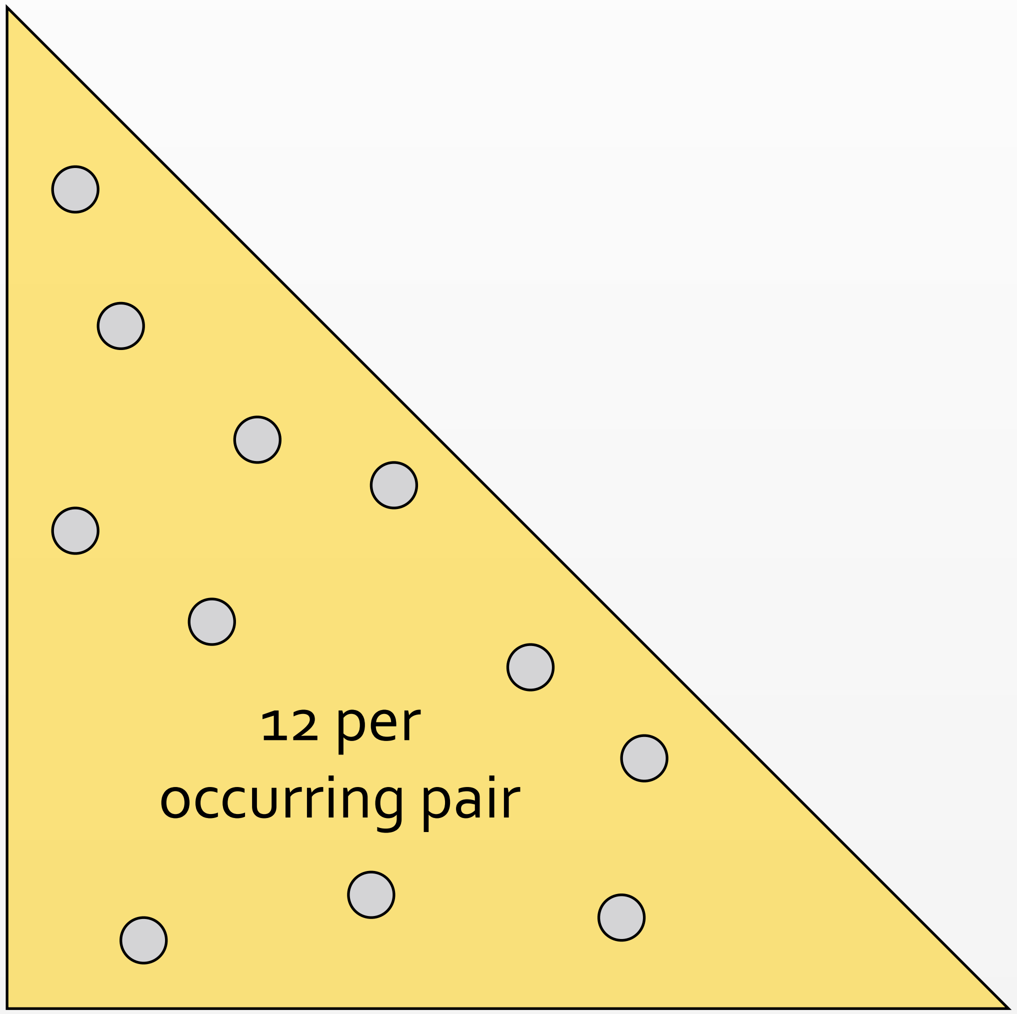
- Approach 1 only requires 4 bytes per pair
- Approach 2 uses 12 bytes per pair (but only for pairs with count > 0)

Comparing the 2 Approaches



4 bytes per pair

Triangular Matrix



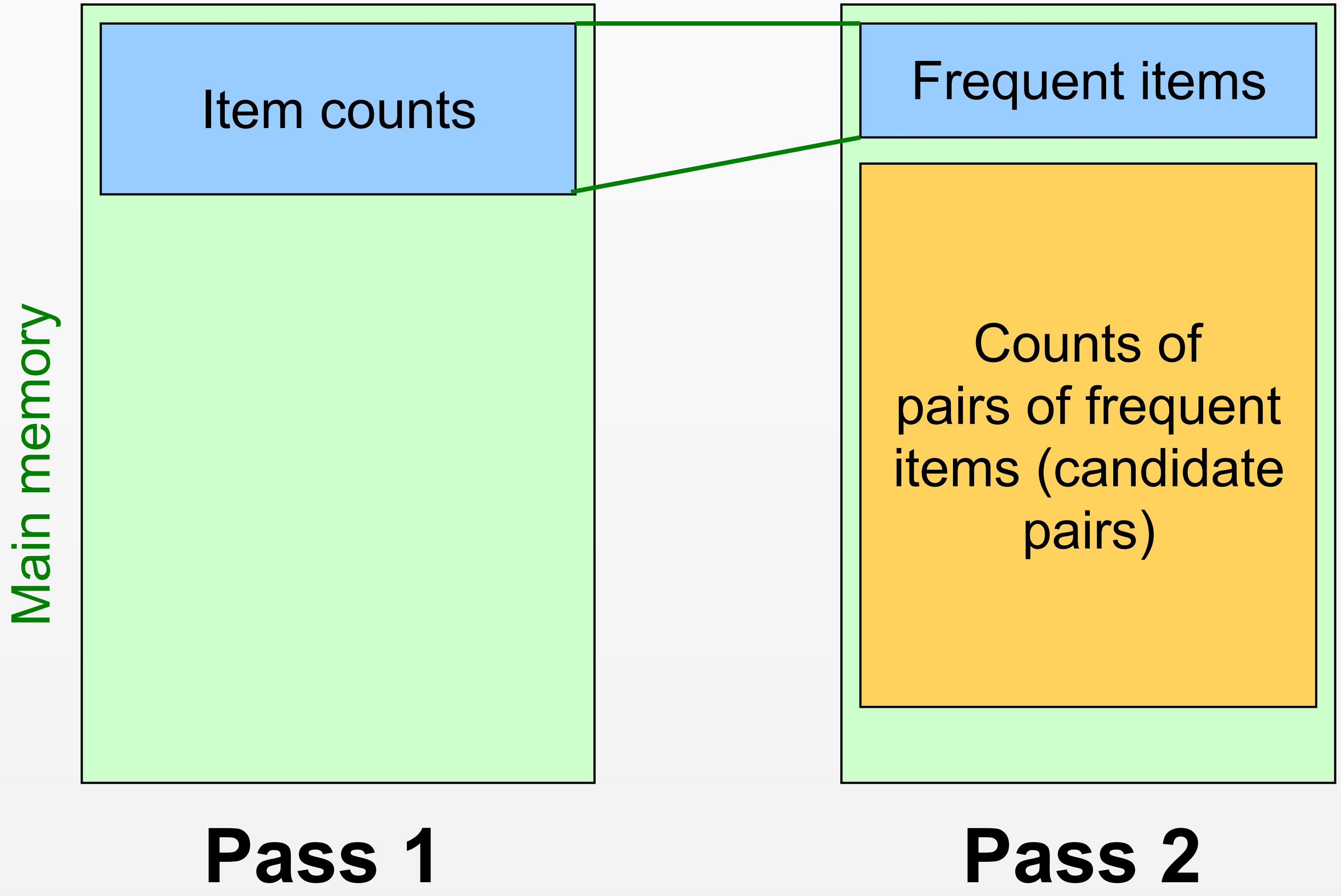
12 per
occurring pair

Triples

Comparing the two approaches

- Approach 1: Triangular Matrix
  - n = total number items
  - Count pair of items $\{i, j\}$ only if $i<j$
  - Keep pair counts in lexicographic order:
    - $\{1,2\}, \{1,3\},\dots, \{1,n\}, \{2,3\}, \{2,4\},\dots,\{2,n\}, \{3,4\},\dots$
  - Pair $\{i, j\}$ is at position $(i-1)(n-i/2) + j-1$
  - Total number of pairs $n(n-1)/2$; total bytes= $2n^2$
  - Triangular Matrix requires 4 bytes per pair
- Approach 2 uses 12 *bytes* per occurring pair *(but only for pairs with count > 0)*
  - Beats Approach 1 if less than 1/3 of possible pairs actually occur

Main-Memory: Picture of Apriori



PCY (Park-Chen-Yu) Algorithm

- Observation: In pass 1 of Apriori, most memory is idle
  - We store only individual item counts
  - Can we reduce the number of candidates C_2 (therefore the 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
  - 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

```
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 whether it is present at least *s* (support) times

J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets, http://www.mmds.org

Eliminating Candidates using 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 ***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)

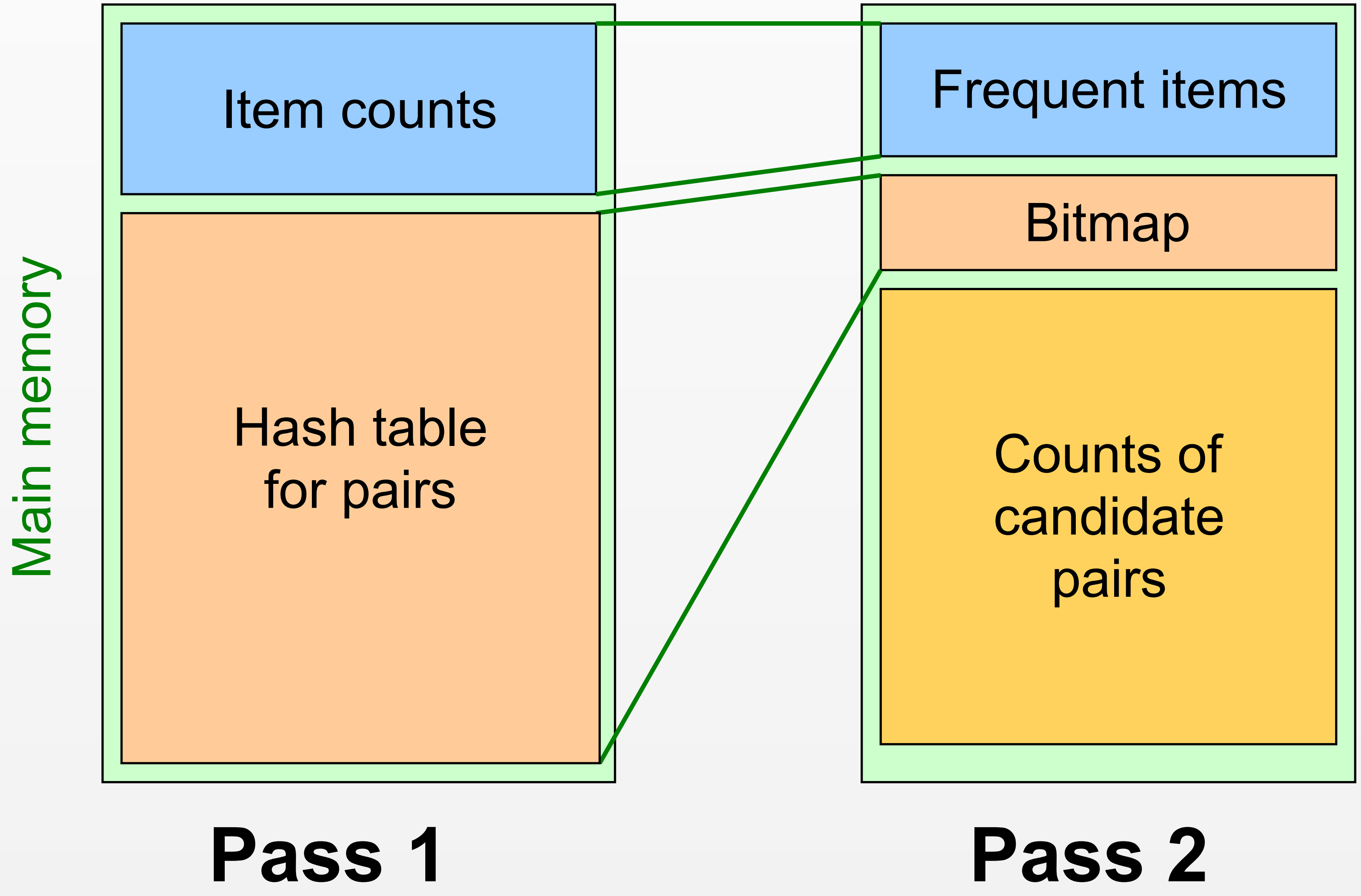
- Both conditions are necessary for the pair to have a chance of being frequent

PCY Algorithm – Summary

New in PCY (steps 3–4)

```
1. Set $k = 0$
2. Define $C_1$ as all size 1 item sets
3. Scan DB to construct $L_1 \subseteq C_1$
   and a hash table of pair counts
4. Convert pair counts to bit vector
   and construct candidates $C_2$
5. While $C_{k+1}$ is not empty
6.   Set $k = k + 1$
7.   Scan DB to determine subset $L_k \subseteq C_k$
     with support $\geq s$
8.   Construct candidates $C_{k+1}$ by combining
     sets in $L_k$ that differ by 1 element
```

Main-Memory: Picture of PCY



Main memory

Item counts

Hash table for pairs

Frequent items

Bitmap

Counts of candidate pairs

Pass 1

Pass 2

J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets, http://www.mmds.org

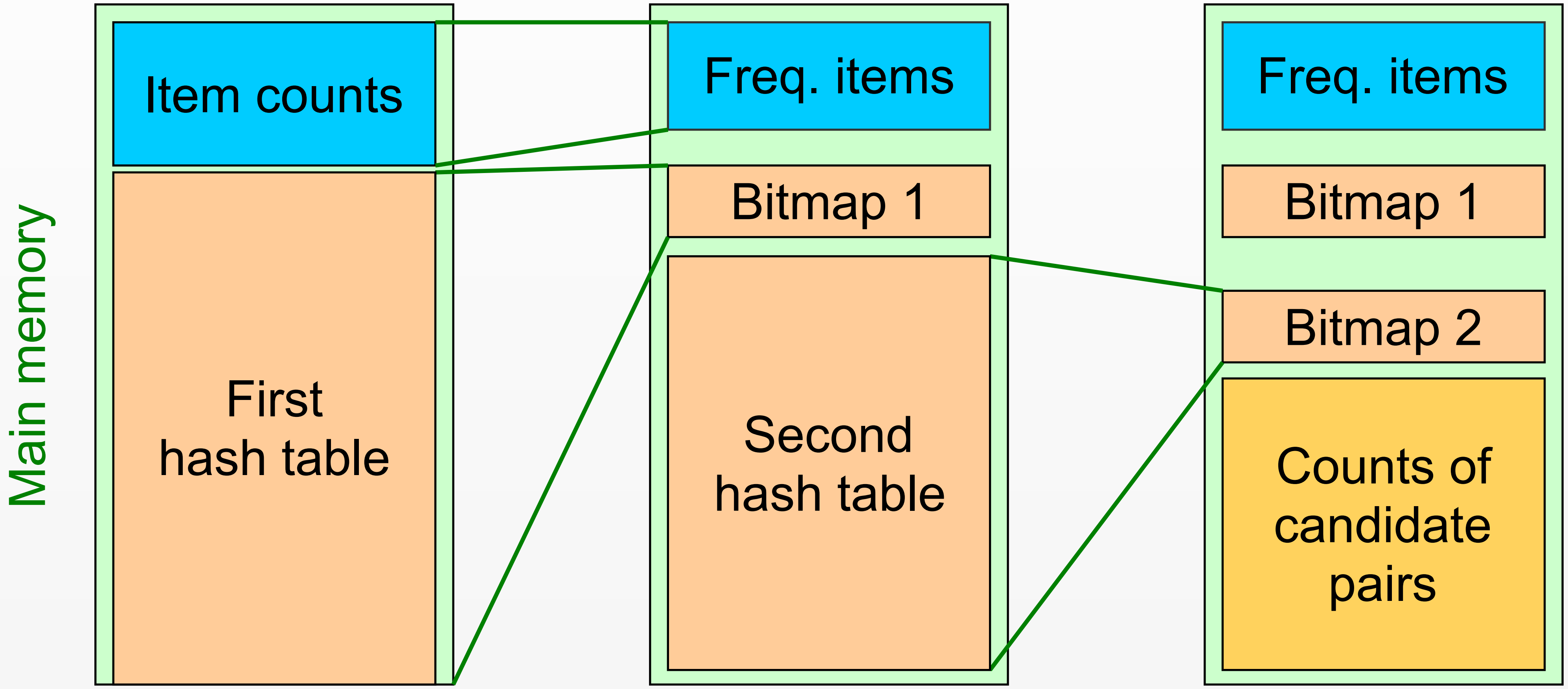
Main-Memory Details

- 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 (we cannot use triangular matrix approach, why?)
  - Thus, hash table must eliminate approx. 2/3 of the candidate pairs for PCY to beat A-Priori

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 PCY



Main memory

Item counts

First
hash table

Freq. items

Bitmap 1

Second
hash table

Freq. items

Bitmap 1

Bitmap 2

Counts of
candidate
pairs

Pass 1

Count items
Hash pairs {i,j}

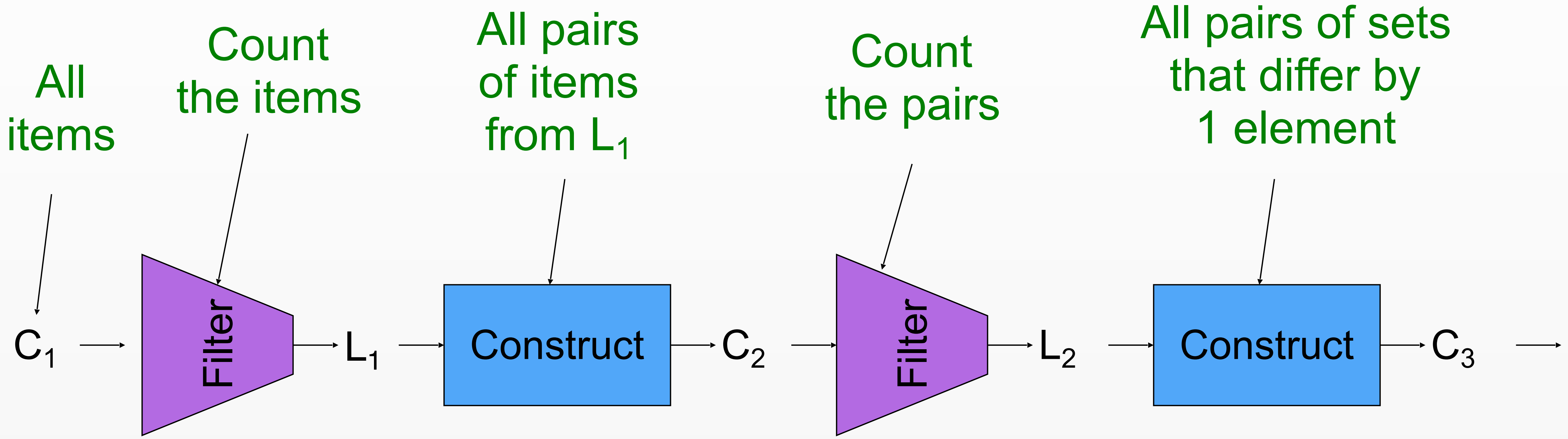
Pass 2

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

Apriori: Bottlenecks



All items

Count the items

All pairs of items from L_1

Count the pairs

All pairs of sets that differ by 1 element

$C_1 \rightarrow$ Filter $\rightarrow L_1 \rightarrow$ Construct $\rightarrow C_2 \rightarrow$ Filter $\rightarrow L_2 \rightarrow$ Construct $\rightarrow C_3 \rightarrow$

```
1. Set $k = 0$
2. Define $C_1$ as all size 1 item sets
3. While $C_{k+1}$ is not empty
4.   Set $k = k + 1$
5.   Scan DB to determine subset $L_k \subseteq C_k$        (I/O limited)
     with support $\geq s$
6.   Construct candidates $C_{k+1}$ by combining           (Memory
     sets in $L_k$ that differ by 1 element                 limited)
```

FP-Growth Algorithm – Overview

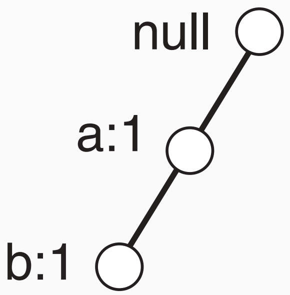
- Apriori requires one pass for each *k* (2+ on first pass for PCY variants)
- Can we find ***all*** frequent item sets in fewer passes over the data?

FP-Growth Algorithm:

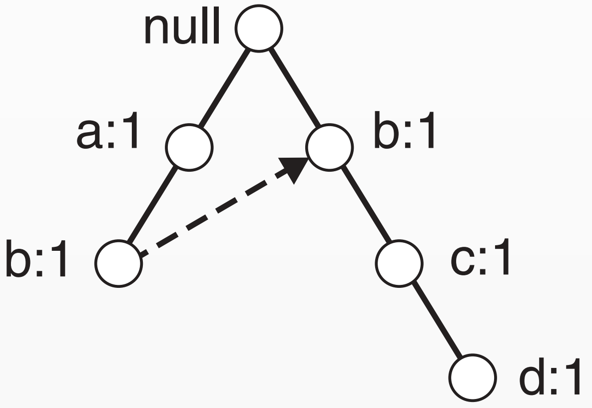
- *Pass 1*: Count items with support ≥ s
- Sort frequent items in descending order according to count
- *Pass 2*: Store all frequent itemsets in a frequent pattern tree (FP-tree)
- Mine patterns from FP-Tree

FP-Tree Construction

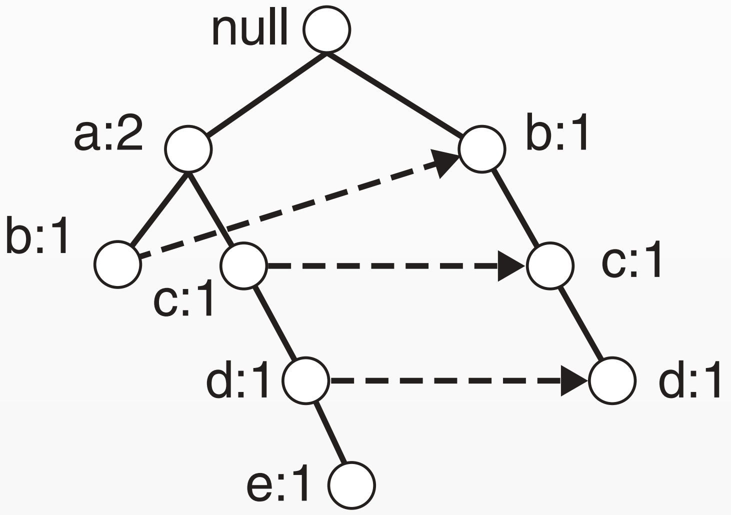
TID = 1



TID = 2

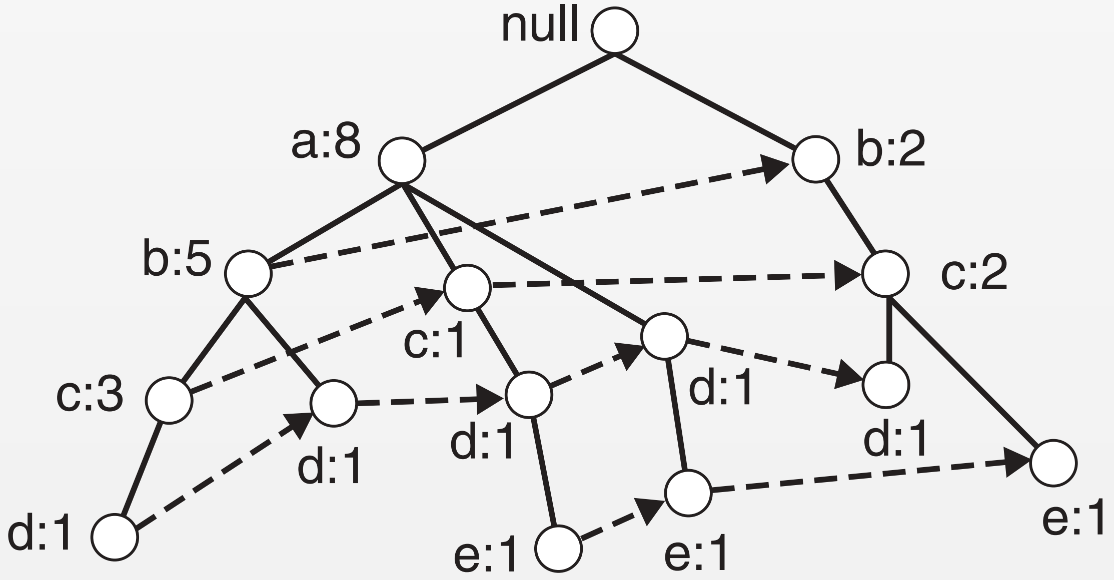


TID = 3



TID	Items Bought	Frequent Items
1	{a,b,f}	{a,b}
2	{b,g,c,d}	{b,c,d}
3	{h, a,c,d,e}	{a,c,d,e}
4	{a,d, p,e}	{a,d,e}
5	{a,b,c}	{a,b,c}
6	{a,b,q,c,d}	{a,b,c,d}
7	{a}	{a}
8	{a,m,b,c}	{a,b,c}
9	{a,b,n,d}	{a,b,d}
10	{b,c,e}	{b,c,e}

TID = 10

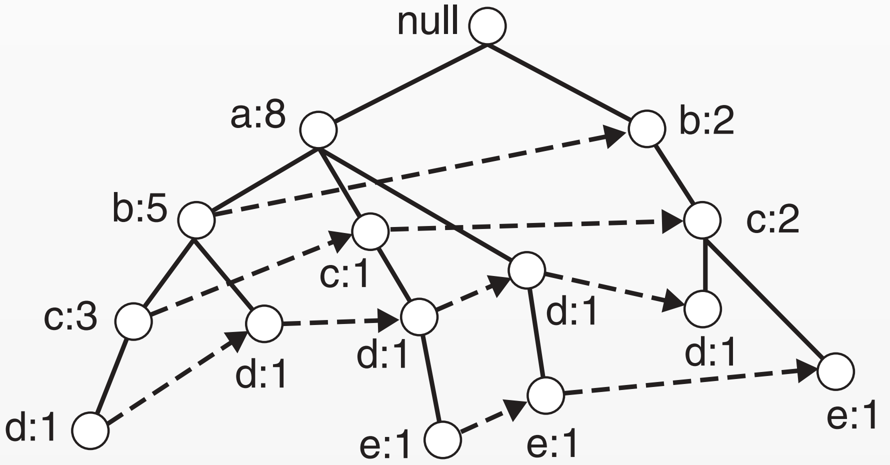


a: 8, b: 7, c: 6, d: 5, e: 3,
~~f: 1~~, ~~g: 1~~, ~~h: 1~~, ~~m: 1~~, ~~n: 1~~

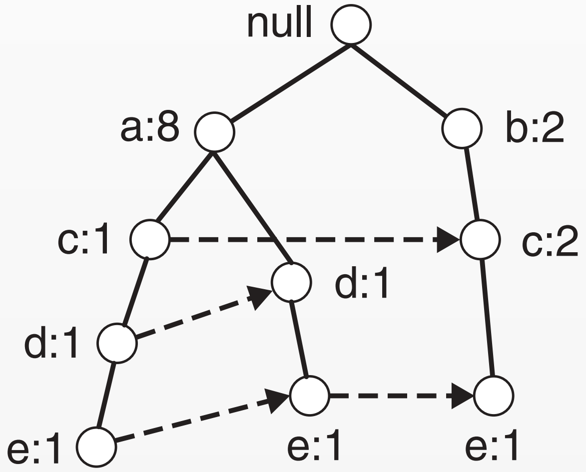
Mining Patterns from the FP-Tree

Step 1: Extract subtrees ending in each item

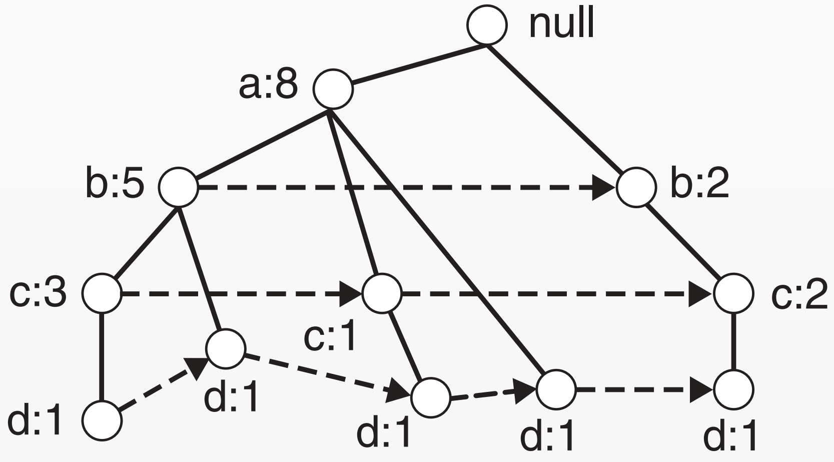
Full Tree



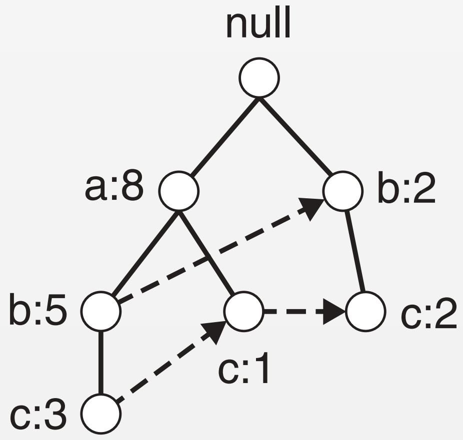
Subtree *e*



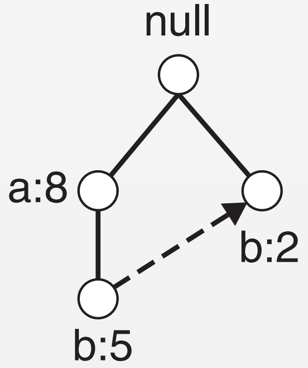
Subtree *d*



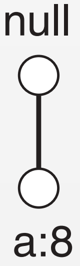
Subtree *c*



Subtree *b*



Subtree *a*

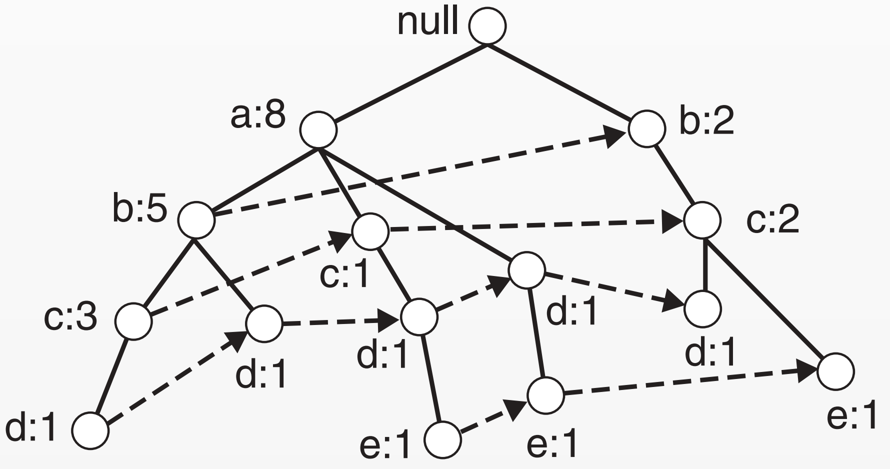


a: 8, b: 7, c: 6, d: 5, e: 3, ~~f: 1~~, ~~g: 1~~, ~~h: 1~~, ~~m: 1~~, ~~n: 1~~

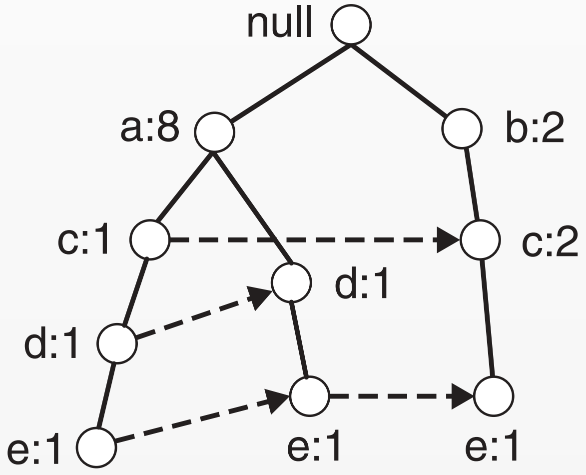
Mining Patterns from the FP-Tree

Step 2: Construct Conditional FP-Tree for each item

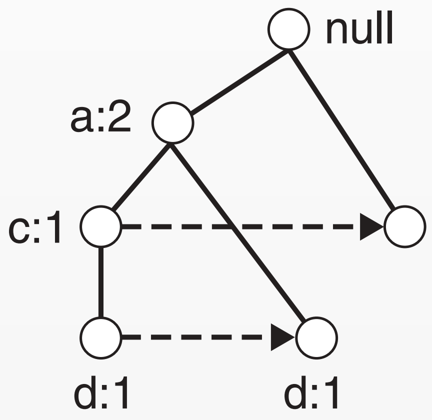
Full Tree



Subtree *e*



Conditional *e*



Conditional Pattern Base for e
acd: 1, ad: 1, bc: 1

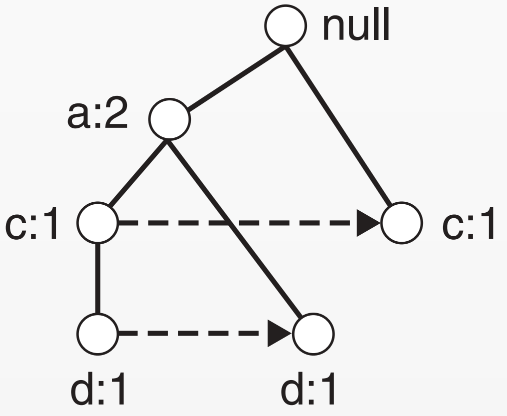
Conditional Node Counts
a: 2, ~~b: 1~~, c: 2, d: 2

- Calculate counts for paths ending in *e*
- Remove leaf nodes
- Prune nodes with count ≤ *s*

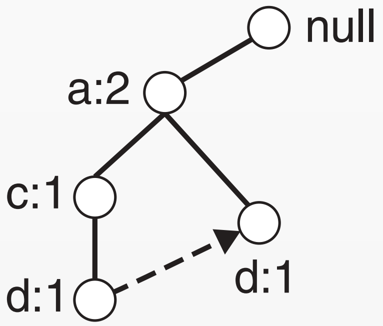
Mining Patterns from the FP-Tree

Step 3: Recursively mine conditional FP-Tree for each item

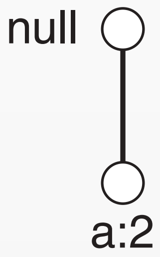
Conditional *e*



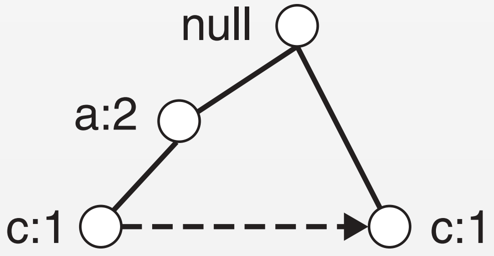
Subtree *de*



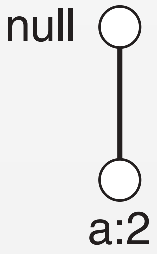
Conditional *de*



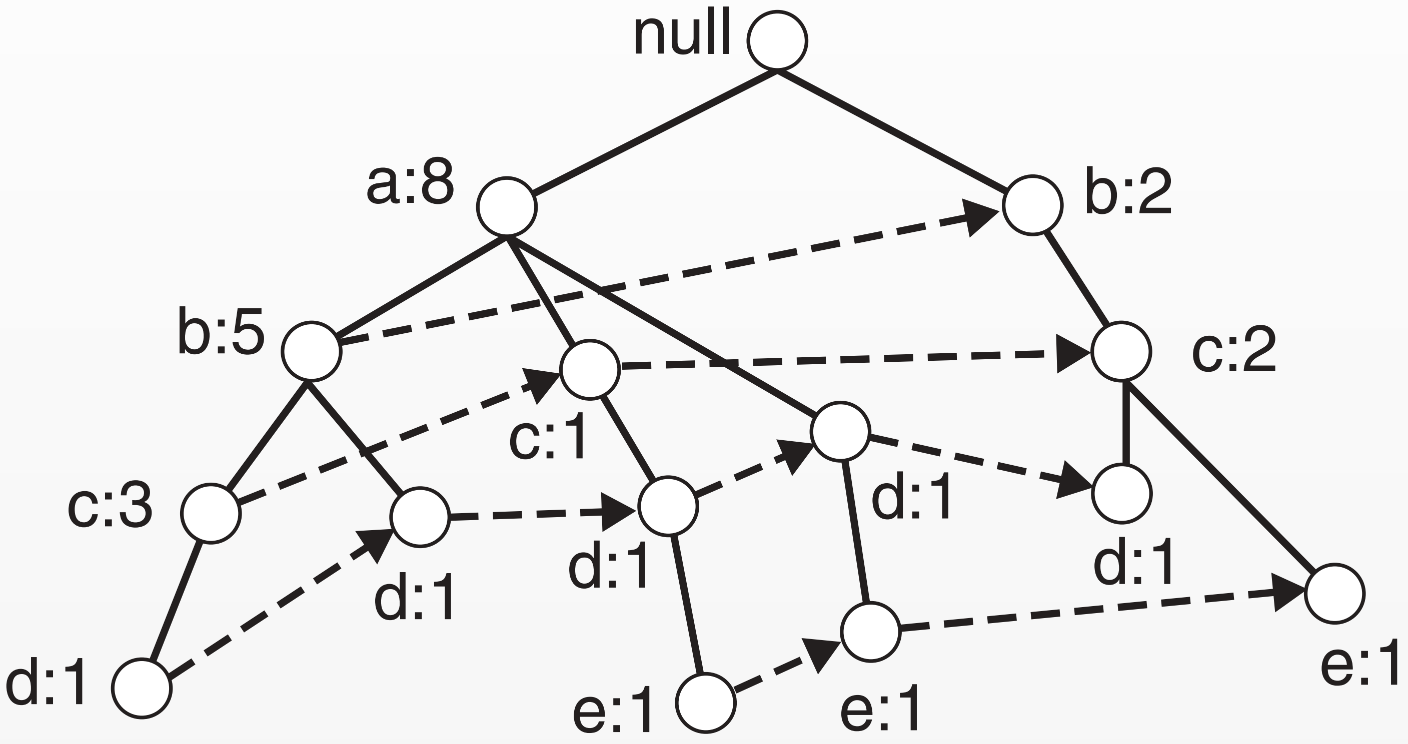
Subtree *ce*

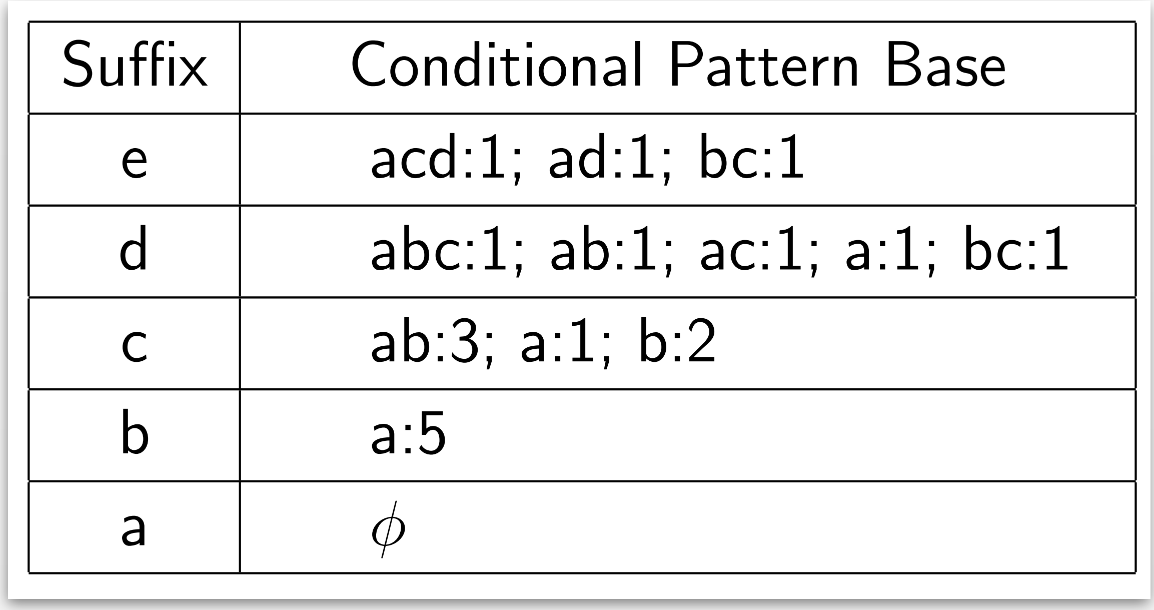


Subtree *ae*

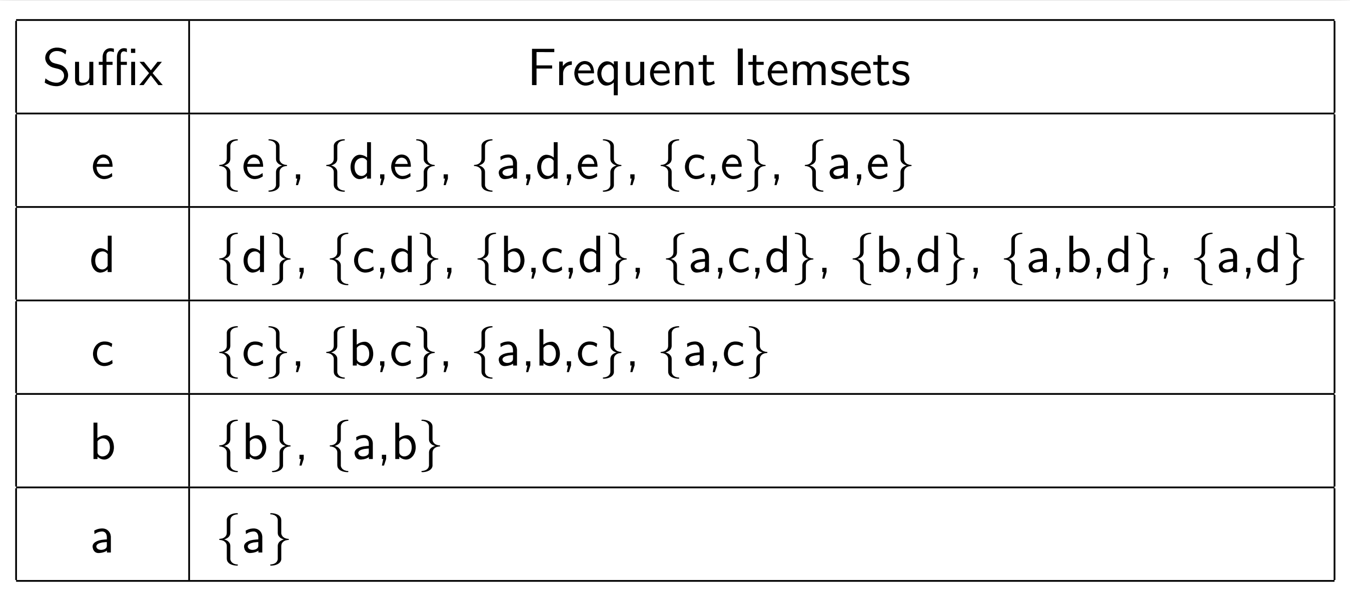


Mining Patterns from the FP-Tree



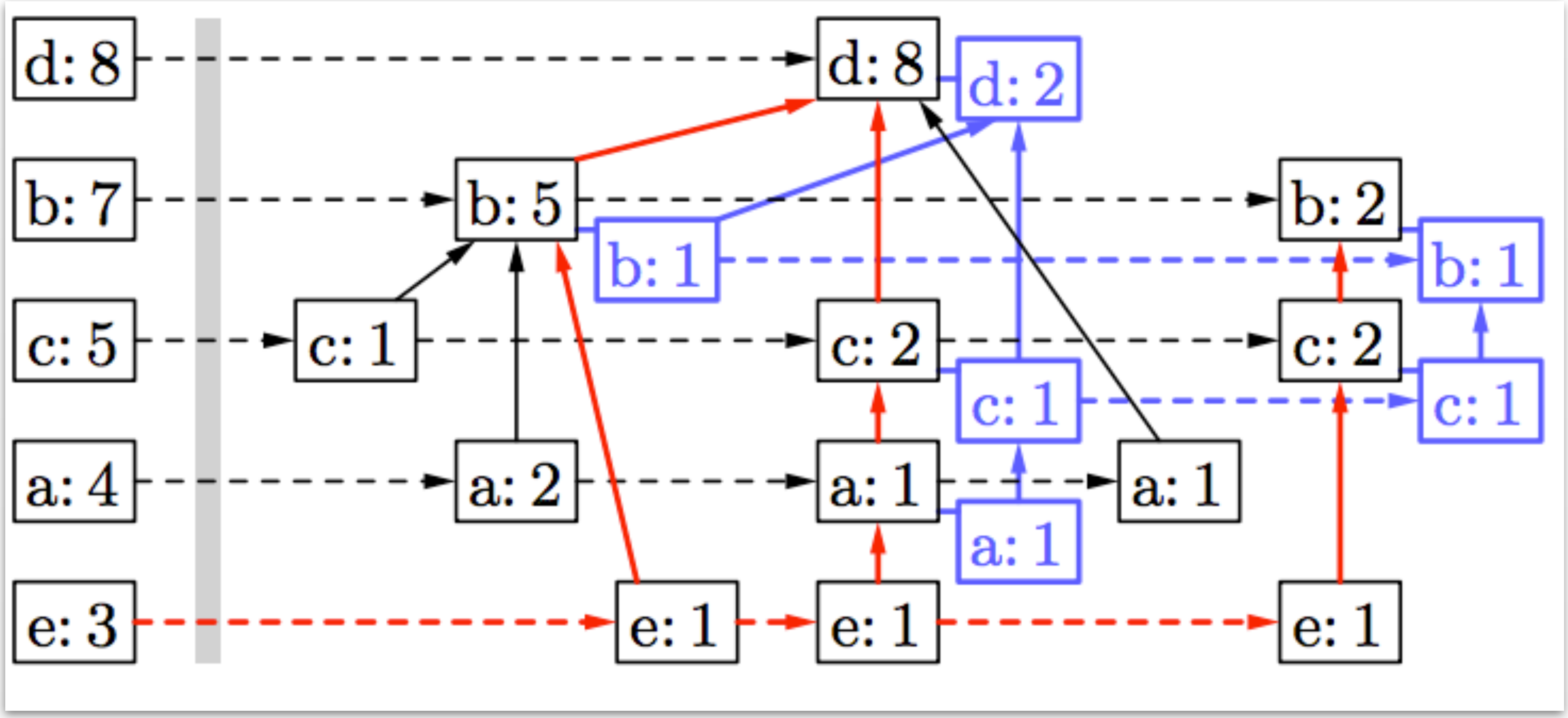


Suffix	Conditional Pattern Base
e	acd:1; ad:1; bc:1
d	abc:1; ab:1; ac:1; a:1; bc:1
c	ab:3; a:1; b:2
b	a:5
a	ϕ



Suffix	Frequent Itemsets
e	{e}, {d,e}, {a,d,e}, {c,e}, {a,e}
d	{d}, {c,d}, {b,c,d}, {a,c,d}, {b,d}, {a,b,d}, {a,d}
c	{c}, {b,c}, {a,b,c}, {a,c}
b	{b}, {a,b}
a	{a}

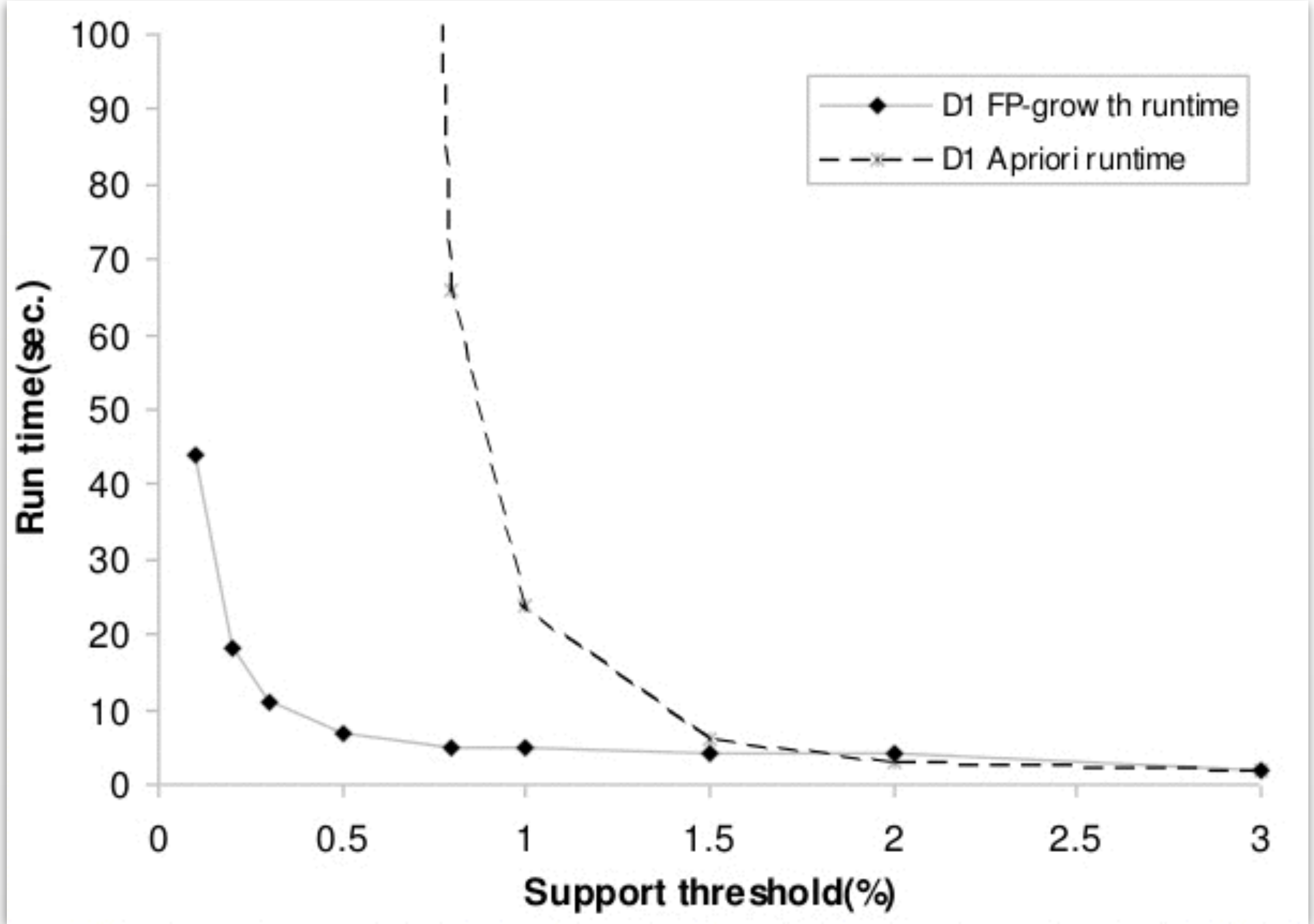
Projecting Sub-trees



- “Cutting” and “pruning” trees requires that we create copies/mirrors of the subtrees
- Mining patterns requires additional memory

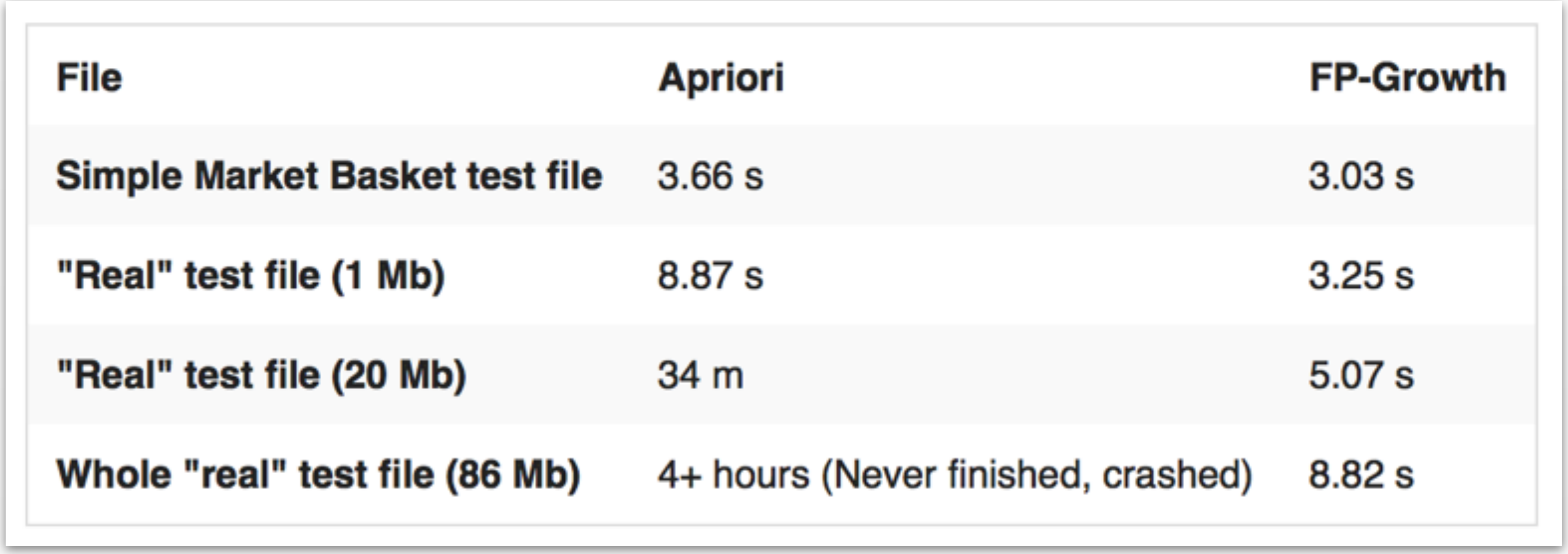
FP-Growth vs Apriori

Simulated data 10k baskets, 25 items on average



(from: Han, Kamber & Pei, Chapter 6)

FP-Growth vs Apriori



File	Apriori	FP-Growth
Simple Market Basket test file	3.66 s	3.03 s
"Real" test file (1 Mb)	8.87 s	3.25 s
"Real" test file (20 Mb)	34 m	5.07 s
Whole "real" test file (86 Mb)	4+ hours (Never finished, crashed)	8.82 s

http://singularities.com/blog/2015/08/apriori-vs-fpgrowth-for-frequent-item-set-mining

FP-Growth vs Apriori

Advantages of FP-Growth

- Only 2 passes over dataset
- Stores “compact” version of dataset
- No candidate generation
- Faster than A-priori

Disadvantages of FP-Growth

- The FP-Tree may not be “compact” enough to fit in memory
- Even more memory required to construct subtrees in mining phase