Chapter 5: Mining Frequent Patterns, Association and Correlations

Dong-Kyu Chae

PI of the Data Intelligence Lab @HYU
Department of Computer Science & Data Science
Hanyang University

Data
Intelligence
LAB

Challenges of Frequent Pattern Mining

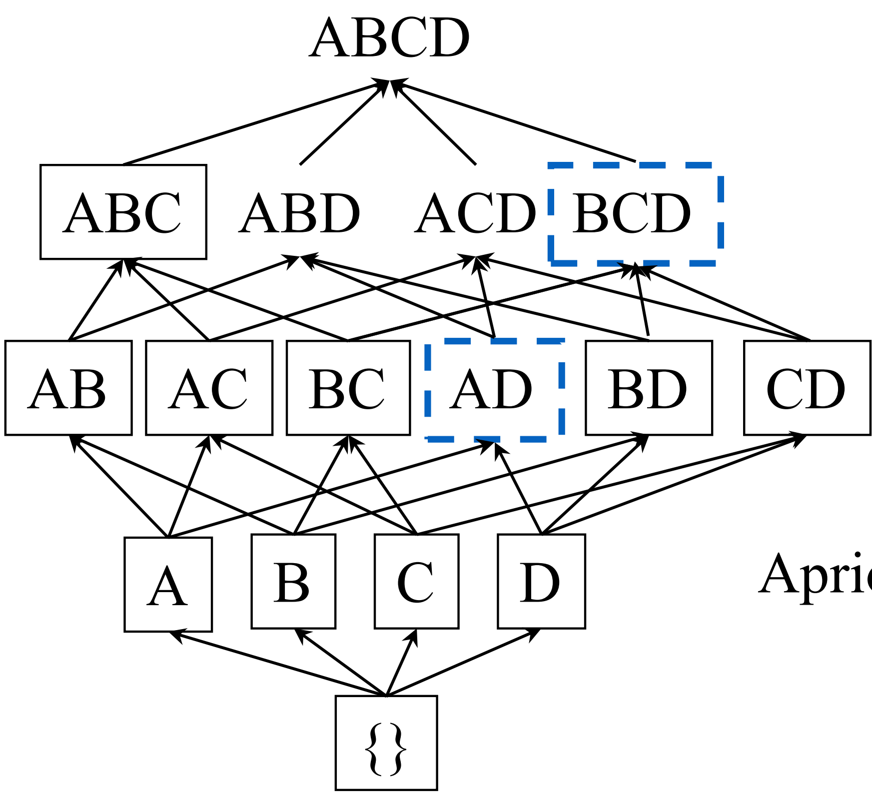
Challenges

- Multiple scans of DB (*k* times): too much costly
- Huge number of candidates
  - To find frequent itemset $i_1 i_2 \ldots i_{100}$
  - # of scans: 100
  - # of Candidates = $2^{100}-1$ = 1.27×10^{30} !
- Tedious workload of the candidate-generation-and-test process
  - support counting for candidates

Improving Apriori: general ideas

- Reduce the number of DB scans
- Reduce the number of candidates
- Improve the candidate counting approaches

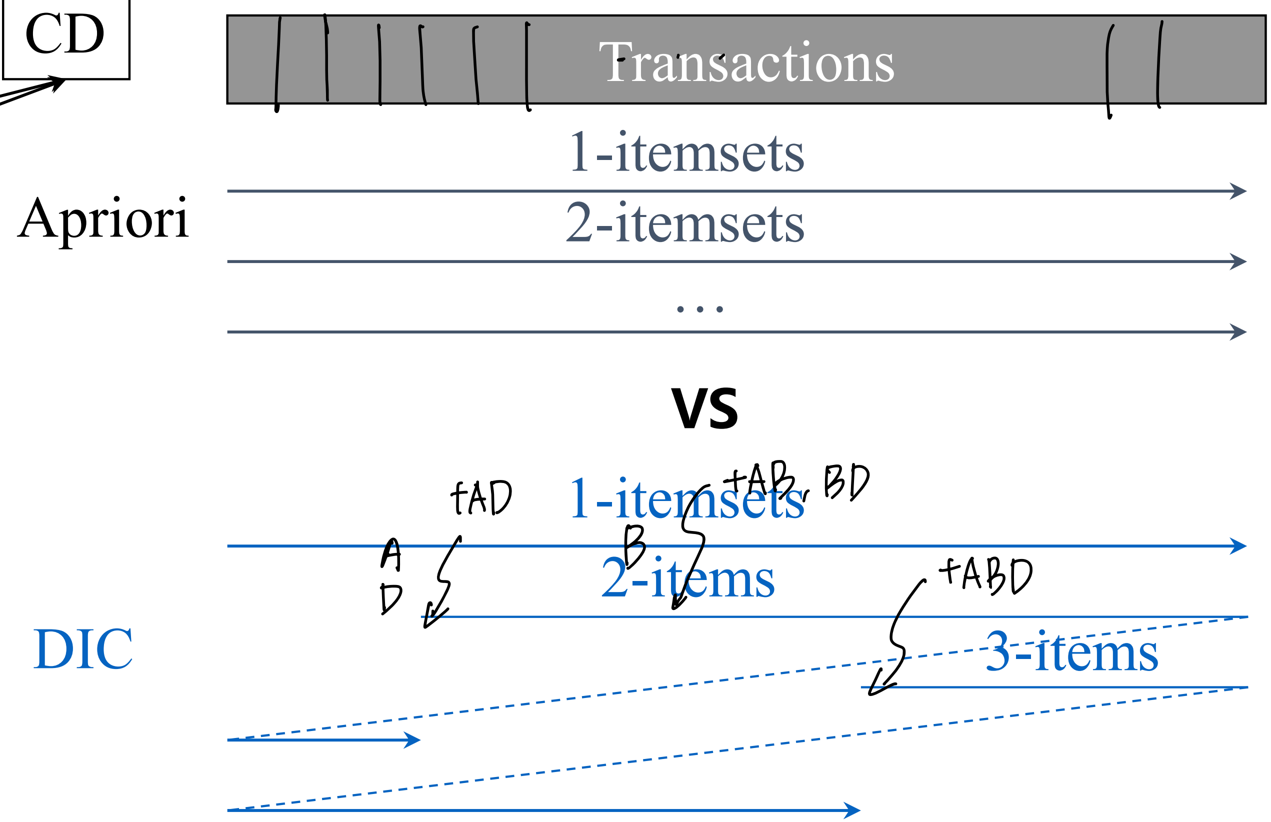
DIC: Reduce Number of Scans



Itemset lattice

DIC: Dynamic Itemset Counting

- Once both A and D are determined frequent, the counting of AD begins **at that time**
- Once all length-2 subsets of BCD are determined frequent, the counting of BCD begins at that time



Partition: Scan Database Only Twice

Approach

- Divide a DB into k pieces (local databases called *partition*)

k=4

m.s=40

P1	P2	P3	P4

l.m.s=40/4=10

  - Each partition should reside in main memory
- Find *local frequent patterns* in each partition (scan 1)
  - localMinSup is set as (minSup / k) partition 개수
  - Local frequent patterns have their localSup larger than localMinSup in any local database
- Consolidate global frequent patterns (scan 2)

{A, B, C} : 11 (P1), 3(P2), 4(P3), 10(P4) ⇒ Total frequency: 28 (< m.s)

→ Not frequent pattern

Partition: Scan Database Only Twice

- **It guarantees that frequent patterns are never missed**
  - Any itemset globally frequent in DB must be frequent in *at least one partition* of DB

{A B} : 41

9 9 9 9 = 36

14 9 9 9 = 41

P1	**P2**	**P3**	**P4**

- A global frequent pattern must be a local frequent pattern
- But, a local frequent pattern may not be a global frequent pattern. This must be confirmed by scanning the entire DB (2nd scan)

Sampling

- **Randomly select a sample of an original database, mine frequent patterns within the sample (SDB) using Apriori (in the same way as before)**

  - Use a smaller value of the minimum support for SDB (say, **min_sup* (size of SDB) / (size of DB)**)

- **Problems with the simple sampling**
  - Some of frequent patterns found in SDB are not really frequent in the original database (similar with the local frequent patterns in Partition)
  - Some of true frequent patterns could be missed if they are not included in SDB (different with Partition)

Sampling

- **Solutions: two more scanning for verification**
- **1. Scan the whole database once, applying the original minimum support**
  - Verify the frequent itemsets (**S**) found in SDB, and its negative borders (NB: not in S, but all its subsets are in S + single items)
    - S = {a}, {b}, {c}, {f}, {a,b}, {a,c}, {a,f}, {c,f}, {a,c,f}, ...
    - NB = {b,c}, {b,f}, {d}, {e}, ...
- **2. Scan the whole database again**
  - Find missed frequent patterns (due to the success of NBs)
  - For example, **{a,b,c}** due to the success of {a,b}, {a,c}, and {b,c}

DHP: Direct Hashing and Pruning

- When generating L_k , this algorithm also generates all the **size k+1 itemsets for each transaction**, and **hashes** them to a hash table and keeps a count

Database

Tid	Items
100	A, C, D
200	B, C, E
300	A, B, C, E
400	B, E

C_1	count
{A}	2
{B}	3
{C}	3
~~{D}~~	~~1~~
{E}	3

→

L_1
{A}
{B}
{C}
{E}

minimum support, s = 2

ex) predefined order
: A1, B2, C3, D4, E5

We can use any hash function

Making a hash table

100	**{A C}, {A D}, {C D}**
200	**{B C}, {B E}, {C E}**
300	**{A B}, {A C}, {A E}, {B C}, {B E}, {C E}**
400	**{B E}**

A C

(1 x 10 + 3 = 13) mod 7 = 6

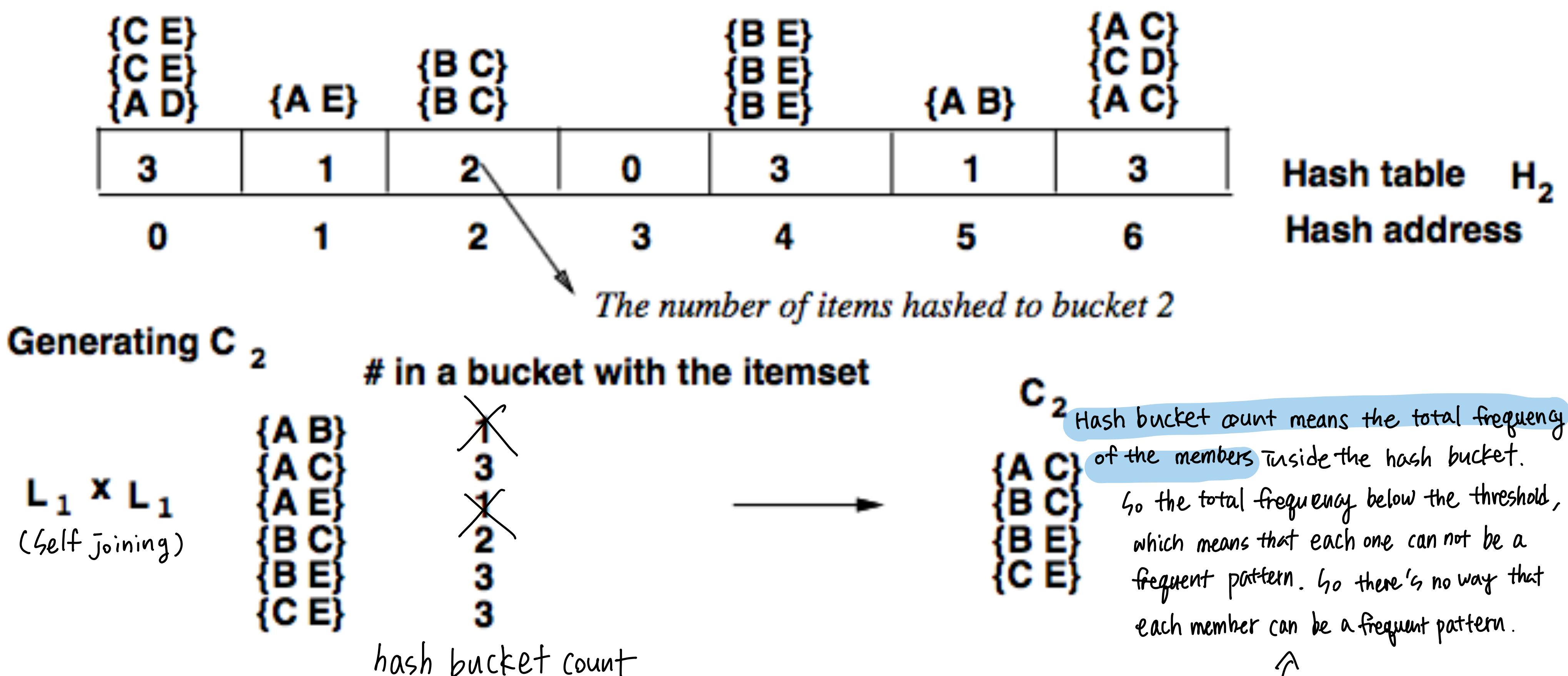
h({x y}) = ((order of x)*10 + (order of y)) mod 7;

	{C E} {C E} {A D}	{A E}	{B C} {B C}		{B E} {B E} {B E}	{A B}	{A C} {C D} {A C}	
Hash bucket count →	3	1	2	0	3	1	3	**Hash table H_2**
	0	1	2	3	4	5	6	**Hash address**

The number of items hashed to bucket 2

DHP: Direct Hashing and Pruning

- While generating C_{k+1} via L_k, it first accesses to the hash table, looks at each candidate's **hash bucket count**.



- If the hash bucket count is below the min_sup, it cannot be a candidate!
  - **Effective in reducing # of candidates**

Data Intelligence LAB

Original Apriori VS. DHP

$Sup_{min} = 2$

Tid	Items
10	A, C, D
20	B, C, E
30	A, B, C, E
40	B, E

C_1

Itemset	sup
{A}	2
{B}	3
{C}	3
{D}	1
{E}	3

L_1

Itemset	sup
{A}	2
{B}	3
{C}	3
{E}	3

DHP

{C E} {C E} {A D}	{A E}	{B C} {B C}		{B E} {B E} {B E}	{A B}	{A C} {C D} {A C}
3	1	2	0	3	1	3
0	1	2	3	4	5	6

Remove before counting freq

DHP는 frequency 세기 전에 hash bucket count 에 기반하여 useless candidates를 제거한다

C_2

Itemset
{A, B}
{A, C}
{A, E}
{B, C}
{B, E}
{C, E}

DHP

Itemset
{A, C}
{B, C}
{B, E}
{C, E}

DHP

more reduced size of candidates

아직 frequent pattern 아닌 candidates

Apriori

C_2

Itemset	sup
{A, B}	1
{A, C}	2
{A, E}	1
{B, C}	2
{B, E}	3
{C, E}	2

L_2

Itemset	sup
{A, C}	2
{B, C}	2
{B, E}	3
{C, E}	2

Until now…

- **We have learned DIC, Partition, Sampling, DHP**
  - Reduce DB scanning time: DIC, Partition, Sampling
  - Reduce # of candidates: DHP, Sampling
- **But, they still have limitations, and they are still very slow**
- **Can we completely avoid candidate generation?**
  - Yes, FP(frequent pattern)-Growth can do this!
- **Next class: FP-Growth**
  - Mining Frequent Patterns Without Candidate Generation

Thank You

Data
Intelligence
Lab