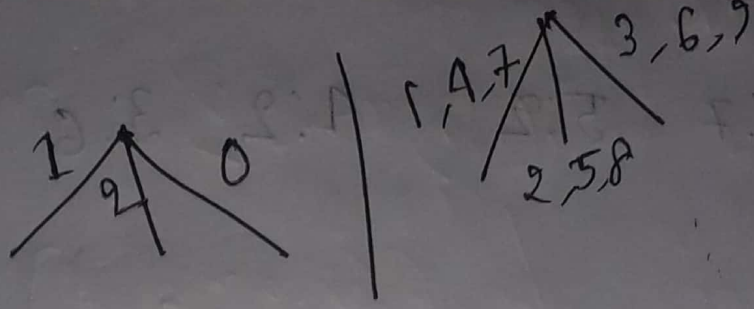
{1,2}, {1,3}, {1,4}, {1,5}, {2,3}, {2,4}, {2,5}
, {3,4}, {3,5}, {4,5}

(1)



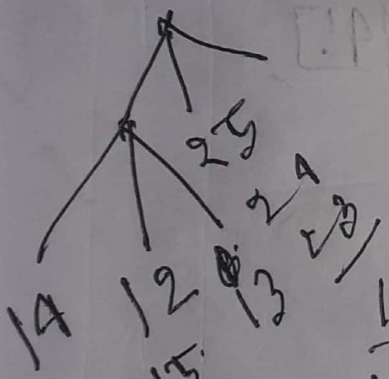
12
15
14

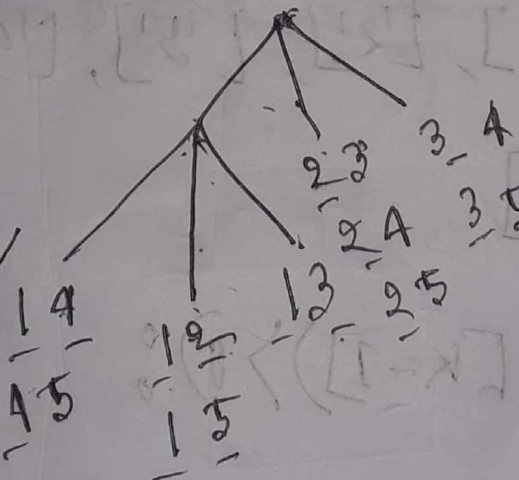
12
13
14
15x

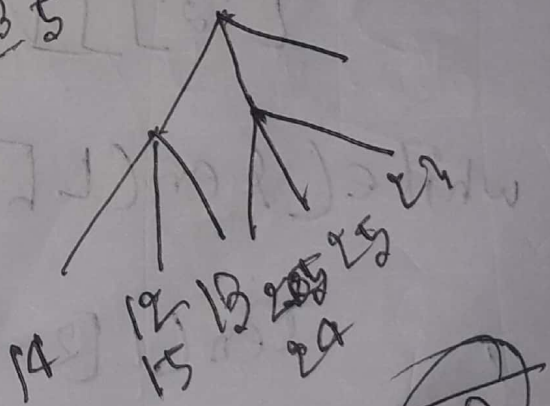
3) 1 (0
0
1

3) 2 (0
0

3) 2 (0
0
2







(2)

(1)

(9)

Apriori

(2)

Frequent One item

1:6 2:7 5:2 4:2 3:6

K=2;

L=[]

L=[0]

L.Lappend(L1)

L = [0, [['1'], ['2'], ['5'], ['4'], ['3']]]

while(len(L[K-1])>0):

{

len(L[2-1])>0 → ~~len~~ L[1]

=1,2,5,4

,3

len(L[1])

=5

ck, kind_ck = api_gen(L[1], k-2) (3)

def api_gen(dataset=[1,3,1,3], k=2)

{

ck = [ ]

lenlk = len(dataset) = 5

for i → 0 [2,3,0] start 2,3,0

for i in range(5) → 0+1=1

for j in range(i+1, 5)

L2 = list(dataset[0])[:3-2]

=

i j

[0][1]

[0][2]

[0][3]

[0][4]

[1][2]

[1][3]

[1][4]

[2][3]

[2][4]

[3][4]

L1 = 1

L2 = 10

L1 = [ ]

L2 = [ ]

0 1

0 2

0 3

if [1]=[] == [3]=[]:

Ck.append(list(set

(dataset[0] | set

↓ (dataset[1]

1 ↓

2

Ck=[1, 2]

find_ck=[]

for candidate in Ck → len(Ck) = 10

all_subsets = list(subset_generation

(set(candidate), k-1)

↓

candidate = [1, 2] k=2-1

Ck এর মধ্যে সব candidate itemset, যাকের length এর

→ [1, 2]

(5)

for candidate in Ck:

{

all_subsets = list(subset_gen

(1, 4), 1) → মানে হল

1 length এর

subset দিয়ে করবো

print(all_subsets) → [2] [1]

bound = True

সেটের একটি non-frequent হলে ও append হবে না

for i in range(all_sub

value = sort(all_sub

আবার, apriori ^function এ

print ("C%d" % (k)) → মানে C2
(k → এর মান 2)

print (find_ck)

h_tree = gen_hash_tree (ck → candidate, max_leaf_count, max_child_count)
(max_leaf_count ↓ 3; max_child_count ↓ 3)

def generate_hash_tree (cand_items, 3, 3):

{

htree = Hashtree (max_child, max_leaf)
(Hashtree → just construction initializ; max_child → 3; max_leaf ↓ 3)

Class Hashtree:

h_tree = self.children{} → pointer to its children

self.leaf_status = True

↓

to know whether present node is leaf or not

self.bucket{} → containing itemsets in bucket

self.max_leaf_c = 3

self.max_child_c = 3

self.freq_items = []

---

for itemset in candidate_items → 2 length of candidate

htree_insert([1, 2])

Scanned with CamScanner

Hash tree (8)

class

def insert(self, itemset)

itemset = (1, 2)

self.recursivdy_insert(root, (1, 2), 0, 0)

def recursivdy_insert(self, node = root, (1, 2), item, index = 0, count = 0)

if index == len(itemset)

0

2

x

if node_leaf_status = True

if itemset in node.bucket: → {}

X

~~node.bucket~~

else:

node_bucket[(1,2)] = 0 → 0 value

(7)

if len(node_bucket) == self_max_ → 1

leaf_count = 3

X

আবার,

for itemset in candidate_itemsets → (1,5) → C2

h_tree.insert(1,5)

def insert(1,5):

itemset = (1,5)

self_re_insert(root, (1,5), 0, 0)

recursively_insert((1,5), 0, 0)

(16)

if (0 == len(1,5)

×

if node.leaf_status = True

node.bucket(1,5) = 0

(1,9),

(1,3)

if (0 index == len(1,3)

×

~~if (node~~

node.bucket(1,3) = 0

if (len(node.bucket == max_leaf_count

for old_itemset, old_count

in node.bucket.items() → { (1,2)
(1,5)
(1,4)

(1)

hash_key = self._hash_function

(old_itemset[index])

↓ ↓

(1,2) 0

~~(1,2~~

def hash_function(self, val) → (1,2)

return int(val) → (1,4

1. self.max_child

- count

1 % 3 = 1

hash_key = 1

if hash_key not in node.
children:

node_children[I] = Hash_node()

(12)

self.children = {}

self.leaf_status = True

self.bucket = {}

self._recur_insert(node.children
[I], old_itemset=
(1,2),
index = 0+1, old_count
= 0

def recursively_insert(node, [1], (1,2), index = 1, count = 0)

if index == len(itemset)

↓

1 × 2

(3)

if node.leaf_status: → True

→ (1,2)

if itemset in node.bucket ×

node.bucket(1,2) = 0

~~if len~~
~~else~~

else

node.bucket[(1,2)] = 0

if len(node.bucket) == 3) ×

↓

1

(19)

self.recursively_insert(node.children[hash_key], old_itemset, (L-1), index+1, old_count)

(L-1)

index → 0 + 1

old_count ↓ 0

itemset (1,2) = 0 → hash_key = 1

(1,5) = 0 → u = 1

(1,4) = 0 → u = 1

Same Hash key = 1

del node.bucket

1,2
1,5
1,4

node.leaf_status = False

recursion

(1,2) = 0 → count → hash_key = 2

(1,5) = 0 → hash_key = 5

(1,4) → hash_key = 1

gen_hash_tree

for itemset=(1,3) in candidate
- itemsets

h_ti insert (1,3)

def insert
itemset = (1,3)
self.rec_ins (root, (1,3), 0, 0)

if index == len(1,3)
↓ ↓
0 2 X

if node_leaf_status → False

else
↗(1,3)
hash_key = hash_function(itemset[0]

def hash_function (1,3)

return 1 % 3 = 1

hash_key = (1)

self.recursively_insert

(node_children[1], (1,3),

index=0+1, count=1)

if index >= len(itemset)

↓ ↓

1 2

×

if node.leaf_status → False

else

hash_key = hash_function (itemset[1]) → 3

hash_function = 3 % 3 = 0 → hash_key

if hash_key not in node.children

↓ 0 == ✓

node.children[0] = HashNode()

self.children = {}

self.leaf_status = True

self.bucket = {}

self.recursively_insert(node.children[0], (1,3), index=1+1, count=0) = 2

(18)

if index == len(itemset)

2 == 2

else

node.hashed[(1,3)] = 0

return

def generate_hash_tree

h.tree.insert(2,5)

def insert(self, itemset) → (2,5)

self.recursively_insert((2,5), 0, 0)

def recursively_insert

if index == len(self) ×

↓

0

if node.leaf_starting → True

else

node.bucket[(2,5)] = 0 → 2

(2,4) = 0 → 1

(2,3) = 0 → 0

if len(node.bucket) == self.max_leaf → 3

- count → 3

for (2,5), 0

hash_key = self.hash_function(old_item[2,5]) → 0

3) 2 (0
0
2

(20)

if node.leaf_status → True

if itemset in node.bucket X

node.bucket[(2,5)] = 0

0
(2,5)

0
(3,4)
0
(2,3)

if len(node.bucket == 3)

↓

2

for old_itemset, old_count in

node.bucket.items()

↓

(2,5)

hash_key = [0] ← index

ret 2 % 3 = 2

self.rec_ins(2, (2,5), index=0+1, old_count=0)

node.bucket[(2,5)] = count + 1 = 0 + 1

3) 5 (1
3
2

(1)

```
def generate_k_subsets
    (Transaction, k=2)
    subsets = [ ]
    for itemset in Dataset:
        subsets.extend(
    return subsets
for subset in k_subsets:
    k_tree.add_support(subset)
```

2 length ka subset

def add_support (Memset)

index = 0

while True:

if Transverse_HNODE.LEAF_STATUS:

Transverse_HNODE.bucket

[0, 2] = def

(23)

~~h_tree.get_f~~

h_tree.add_support([1, 2])

add_support([1, 2])

index = 0

while True:

hash_key = self.hash_function

(1, 2) → index = 0

1 % 3 = 1

hash_key = 1

index = 1

index = 1 + 1 = 2

(29)

hash_key = 2

index = 2 + 1 = 3