Midterm 1 Review

CS 165 | Section

Overview

Part 1: Query processing (Question 1 - 6)

Part 2: Designing a secondary tree-based index (Question 1 - 3)

Part 3: Multi-query optimization (Question 1 - 2)

Setting

Database

- Single table *R*
- *K* attributes {*A1,A2 ... Ak*}
- *N* tuples

Storage

- Column-at-a-time
- Fixed width (*Ai.width*)
- No index

Hardware

- Single CPU core
- 4 layers of memory
- Size of Mi = Mi.size
- Block size of Mi = Mi.block
  - M4.block = 10 * M3.block
  - M3.block = 5 * M2.block
  - M2.block = M1.block
- M4.size > Total Data size > M3.size

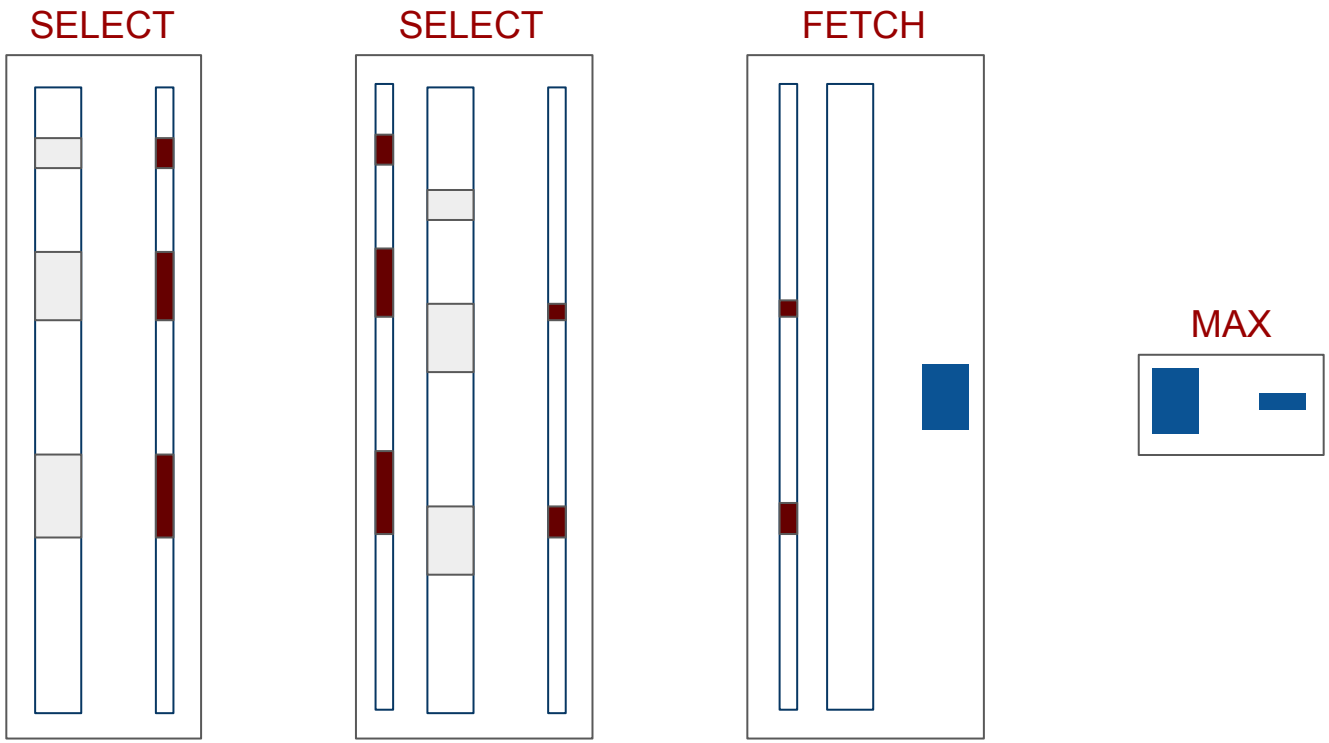
SELECT MAX(D) FROM R WHERE A < 10 and B > 20;

Late tuple reconstruction
Column-at-a-time processing
Bit vectors for intermediate results

SELECT MAX(D) FROM R WHERE A < 10 and B > 20;

Late tuple reconstruction
Column-at-a-time processing
Bit vectors for intermediate results

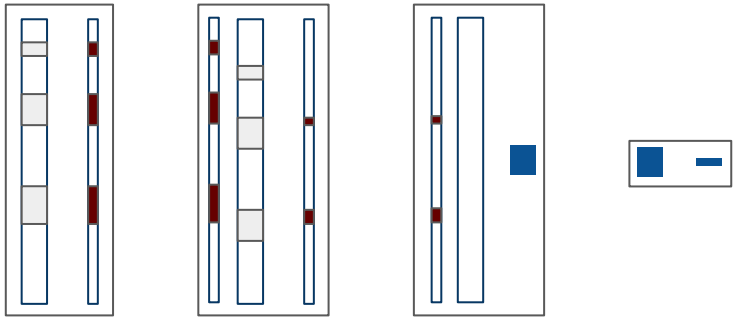
option 1



SELECT MAX(D) FROM R WHERE A < 10 and B > 20;

Late tuple reconstruction
Column-at-a-time processing
Bit vectors for intermediate results

option 1

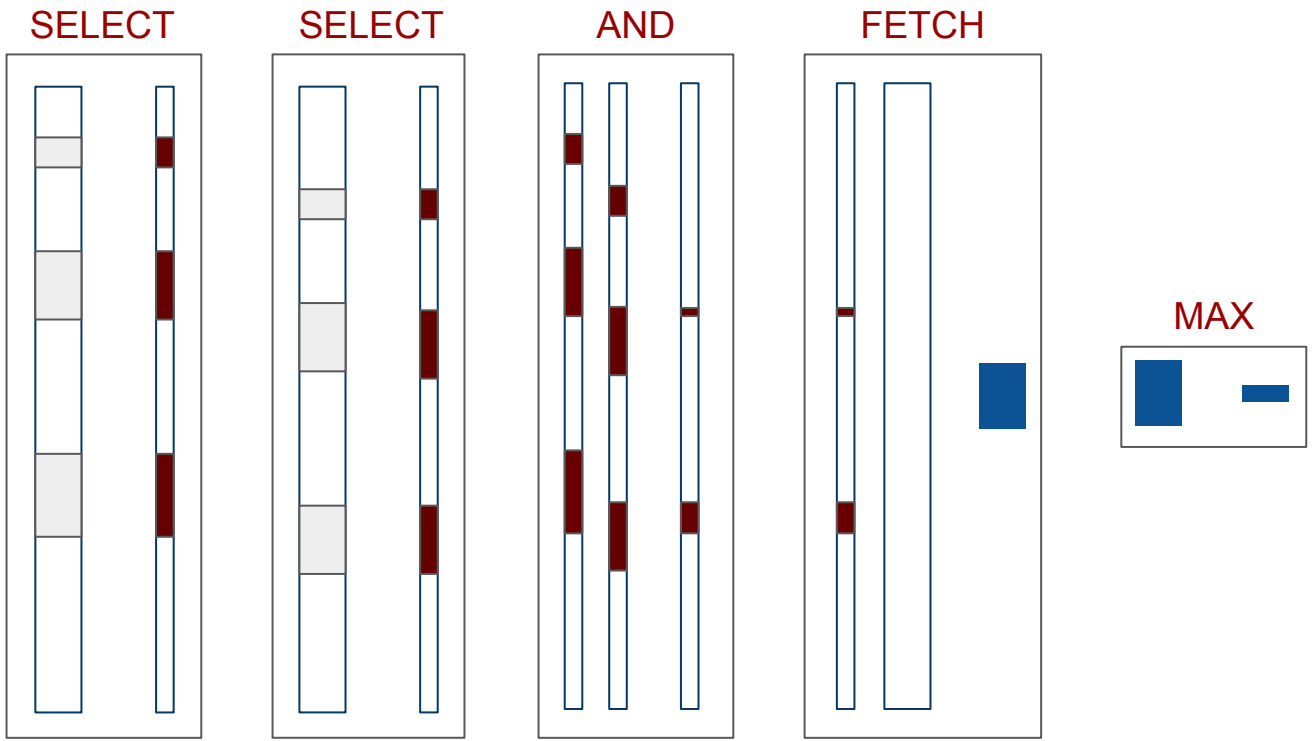


```
inter1=select(A,null,10)
inter2=select(inter1,B,20,null)
inter3=fetch(D,inter2)
result=max(inter3)
```

SELECT MAX(D) FROM R WHERE A < 10 and B > 20;

Late tuple reconstruction
Column-at-a-time processing
Bit vectors for intermediate results

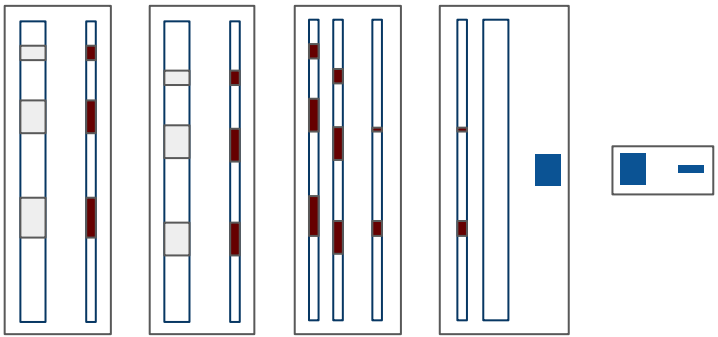
option 2



SELECT MAX(D) FROM R WHERE A < 10 and B > 20;

Late tuple reconstruction
Column-at-a-time processing
Bit vectors for intermediate results

option 2



```
inter1=select(A,null,10)
inter2=select(B,21,null)
inter3=and(inter1,inter2)
inter4=fetch(D,inter3)
result=max(inter4)
```

SELECT MAX(D) FROM R WHERE A < 10 and B > 20;

Late tuple reconstruction
Column-at-a-time processing
Bit vectors for intermediate results

```
select(column,left,right){
    for(i=0;i<column.length;i++){
        if (column[i] >= left && column[i] < right){
            result[i] = 1;
        }else{
            result[i] = 0;
        }
    }
    return result;
}
```

SELECT MAX(D) FROM R WHERE A < 10 and B > 20;

Late tuple reconstruction
Column-at-a-time processing
Bit vectors for intermediate results

```
select(bitvec,column,left,right){
    for(i=0;i<column.length;i++){
        if (column[i] >= left && column[i] < right && bitvec[i]==1){
            result[i] = 1;
        }else{
            result[i] = 0;
        }
    }
    return result;
}
```

SELECT MAX(D) FROM R WHERE A < 10 and B > 20;

Late tuple reconstruction
Column-at-a-time processing
Bit vectors for intermediate results

```
and(bitvec1,bitvec2){
    for(i=0;i<bitvec1.length;i++){
        result[i] = bitvec1[i] && bitvec2[i];
    }
    return result;
}
```

SELECT MAX(D) FROM R WHERE A < 10 and B > 20;

Late tuple reconstruction
Column-at-a-time processing
Bit vectors for intermediate results

```
fetch(column,bitvec){
    result=[];
    for(i=0;i<column.length;i++){
        if(bitvec[i]==1){
            result.append(column[i]);
        }
    }
    return result;
}
```

SELECT MAX(D) FROM R WHERE A < 10 and B > 20;

Late tuple reconstruction
Column-at-a-time processing
Bit vectors for intermediate results

```
max(array){
    max=intmin;
    for(i=0; i<array.length; i++){
        if(array[i]>max){
            max=array[i];
        }
    }
    return max;
}
```

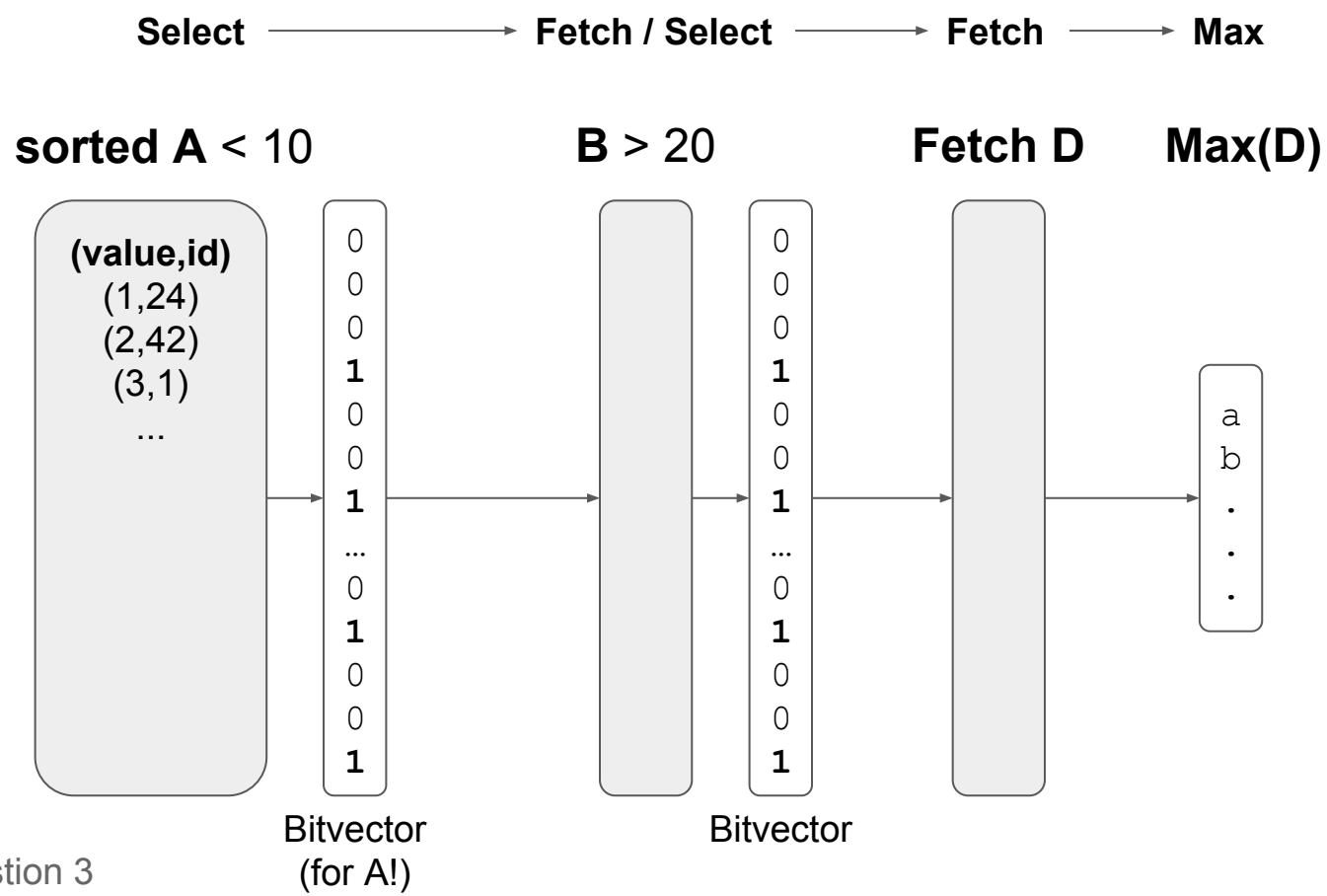
SELECT MAX(D) FROM R WHERE A < 10 and B > 20;

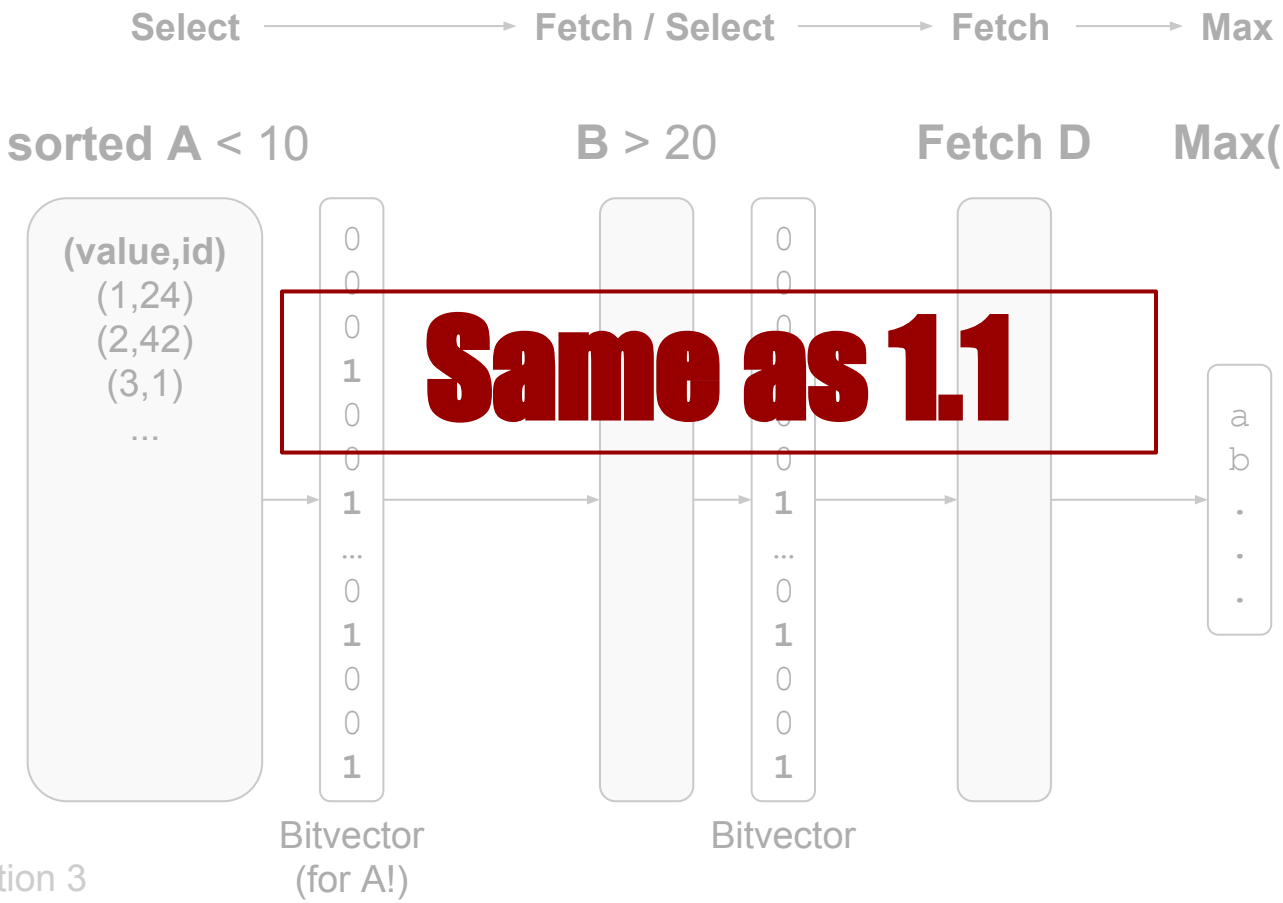
Late tuple reconstruction
Column-at-a-time processing
Bit vectors for intermediate results

Common mistakes:

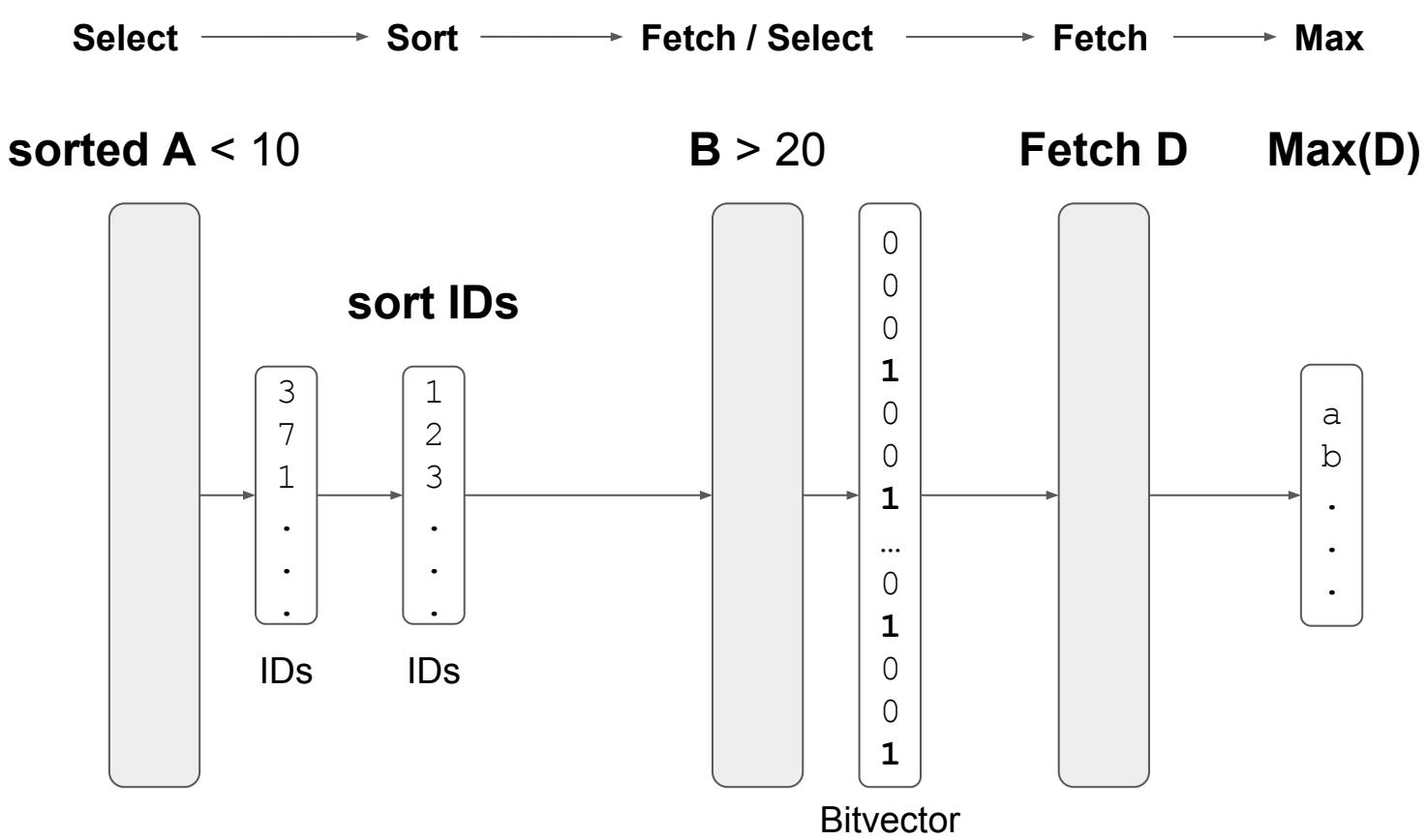
- Position vectors vs. bit vectors
- `Fetch`ing values before `select` leading to loss of positions
- Selectivity of A<10 vs. selectivity of B>20

Query Execution Plan

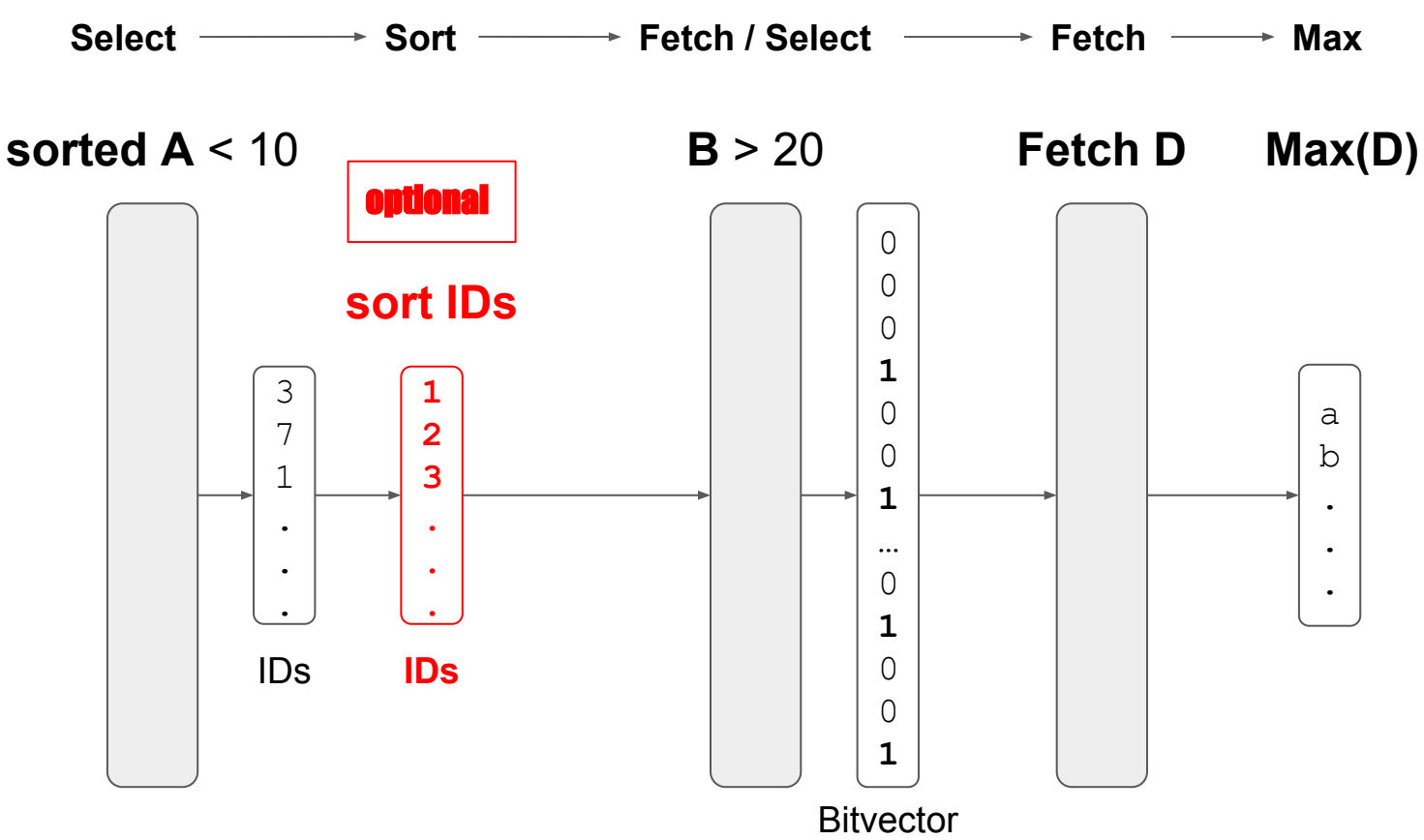



Select
Fetch / Select
Fetch
Max
sorted A < 10
B > 20
Fetch D
Max(D)
(value,id)
(1,24)
(2,42)
(3,1)
...
Same as 1.1
Bitvector
(for A!)
Bitvector

Query Execution Plan



Query Execution Plan



DSL Specification

```
inter1 = select(A,null,10)
inter2 = select(inter1,B,20,null)
inter3 = fetch(D,inter2)
result = max(inter3)
```

```
inter1 = select(A,null,10)
inter2 = select(B,20,null)
inter3 = fetch(D,inter2)
result = max(inter3)
```

Same as 1.1

Operator Pseudo-Code

```
// Scan the secondary index
index_select(size, index_val, index_pos, max):
  // Binary search for the cutoff
  imin = 0, imax = size
  while(imin < imax):
    imid = midpoint(imin,imax)
    if(index_val[imid] < max):
      imin = imid + 1
    else:
      imax = imid

  // Create the output
  bv = init_bitvector()
  for(i = 0; i < imax; i++):
    bv.set(index_pos[i])
  return bv
```

Common Problems

- Secondary index vs primary index! (B, C and D are not sorted)
- Index already exists as sorted column! (No need to build a B-Tree)
- QEP, DSL or pseudo-code missing
- Ignored rest of plan (Say it is the same!)

Plan A: Scan

1 cache miss for every block that fits into the cache *

$$\#M2\_Misses = (\#tuples * a.width)/M2.block + c$$

* This assumes that there is no prefetching

Plan B: Index Scan

1 cache miss for every block that is (randomly) accessed

Binary search + position scan

$$\#M2\_Misses = \log(|A|/M2.block) + (selectivity*|A|)/M2.block + c$$

Common Problems

- A cache miss will fetch a single block and **NOT** replace the whole cache!
- Cache miss means: *How often do we have to bring in a new block?*
- Ignored rest of the plan (this is part of the cost too!)
- There were two plans to compare
- Either textual description or equation missing

Which plan is best when and why, the plan with the index or the scan?

An ideal answer would build on 1.4. Hence, the main factor expected to be discussed is:

- The formulas of 1.4 and how they connect with the decision
- The ***selectivity*** of each query (i.e., how many values qualify)

Comments:

- Some answers discussed that for very small datasets we will always scan.
  - This absolutely correct, but it is not a complete solution.
- Index in presence of updates → hard to maintain
  - not an actual question of the midterm, but the insights were in the right track

Devise a solution that automatically picks the best plan in any scenario.

In order to always make "the best" decision between scans and indexes we expected the following discussion:

- Ideally, use again the previous formulas, having ***selectivity*** as a factor
- Add the provision of maintaining ***statistics*** for the values of the column
  - assuming histograms to help estimate selectivity for each query is the best option
  - assuming sampling of data during query time is a secondary option (the problem here is that there is a sampling overhead in each query)

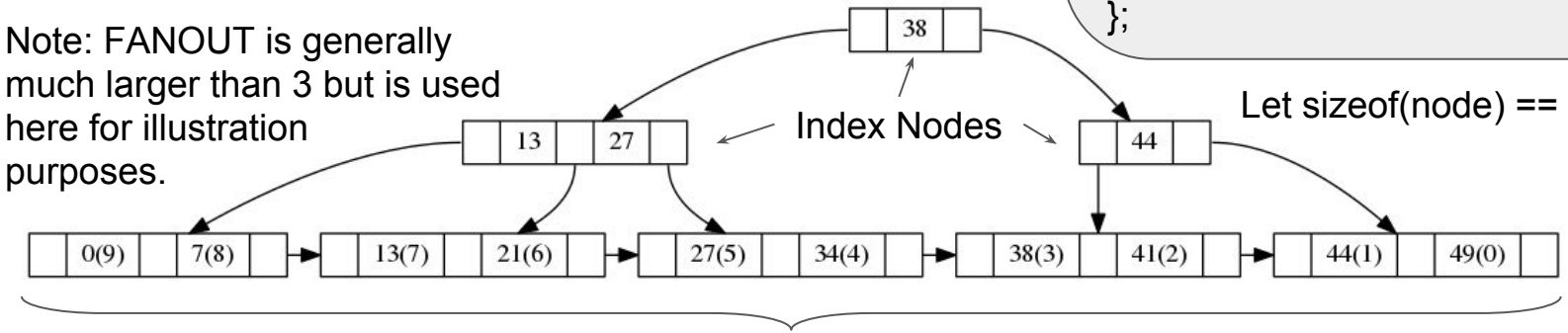
Describe in detail the structure of your index (nodes, connections, etc.)

The tree index holds key-value pairs. Keys are integers. Each value is either a position in the base data, an array of positions in the base data, or a pointer to a node. Each node is made of an unsigned length (the count of currently used keys), FANOUT-1 (sorted) keys, and FANOUT values. There are two types of nodes: index and leaf. All data is stored in the leaf nodes. Index nodes are used to quickly get to the data of interest (described in the next section). Leaf nodes are connected to each other forming a linked list.

```
typedef union data {
        unsigned i;
        struct node *n;
        int *a;
} data;

struct node {
        unsigned len;
        int keys[FANOUT - 1];
        data values[FANOUT];
};
```

Note: FANOUT is generally much larger than 3 but is used here for illustration purposes.



Let sizeof(node) == M2.block

Describe how you insert data, delete data, and probe for ranges.

INSERT(node root, entry{key:foo, value:bar}) - Since all data is held in the leaves we must first navigate through the index nodes to the correct position for our new entry. For this, we first look at the top most index node, called the root. We search the sorted keys to find the insert position of foo. That is, we examine the keys in the root to find the offset where foo would fit (note that if foo is equal to one of the keys, it logically goes to the right of the existing key) and we use this offset to lookup the value. While root is not a leaf node, we recursively call insert passing the current node's value at the offset we just found.

Now that we have found the leaf we again find the position to insert foo in this node. If foo is not equal to any of the keys, we insert the entry in the offset position and move the entries that are greater than foo one spot to the right. If foo is a duplicate we add bar to the existing list for key foo. If the node length is equal to FANOUT, we split the node. First, a new empty node is initialized (new_node). Based on key, the FANOUT/2 largest entries are copied into new_node. New_node's "next" pointer is set equal to old_node's. Then, old_node's length is adjusted and it's "next" pointer is set to point at new_node. Finally the minimum key in new_node is passed along with the address of new_node.

Describe how you insert data, delete data, and probe for ranges.

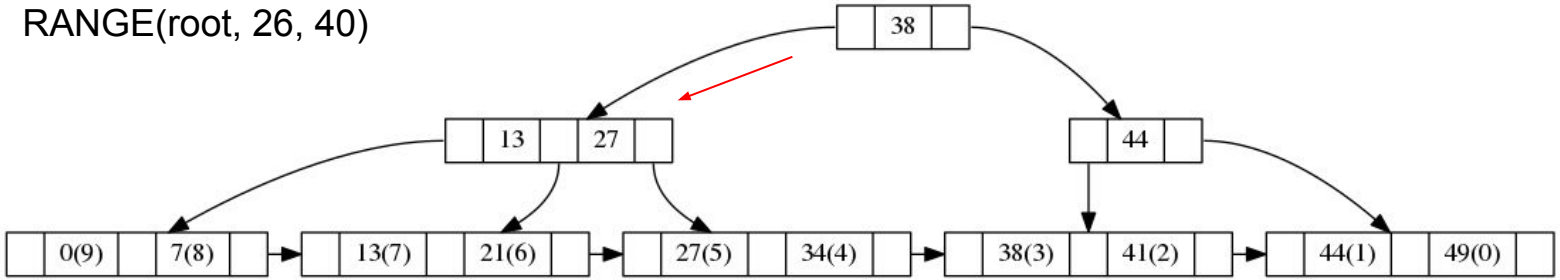
INSERT(continued) -
When the recursive insert returns to the caller if an entry has been passed up, it is inserted in the current node and if this causes a length equal to FANOUT the index node is split in the same way previously described. This continues up to the root. If the root becomes full, it too is split the only difference is that a new root is created on split.

DELETE(node root, key foo) - Deleting is the inverse operation to insert. The tree is descended recursively in the same way as insert. The target is removed from the leaf and the other entries are adjusted to keep the node dense. If the node falls below a certain fullness, for example half, the entries from the neighboring nodes can be used to fill in. If the neighbors are also at the threshold, two nodes can be collapsed in the opposite manner that they are split. However, in practice lazy deletes are more common. Working under the assumption that trees tend to grow rather than shrink, underfull nodes are rarely combined or rebalanced.

Describe how you insert data, delete data, and probe for ranges.

RANGE(node root, int left, int right) - A range is probed by first descending to the leaves using left as the target. Once the leaves are reached, the values are collected until a key bigger than right is located. Since the keys are in order, this is a sequential read within a node, and a linked list traversal for subsequent nodes.

RANGE(root, 26, 40)

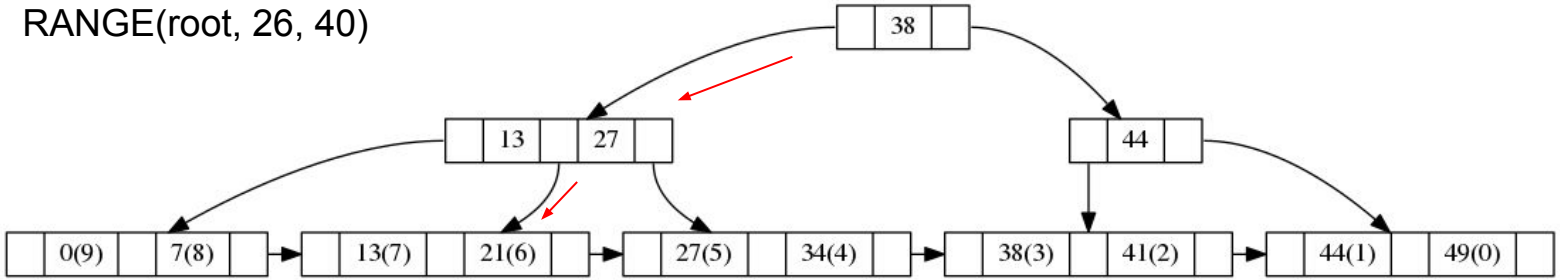


Results:

Describe how you insert data, delete data, and probe for ranges.

RANGE(node root, int left, int right) - A range is probed by first descending to the leaves using left as the target. Once the leaves are reached, the values are collected until a key bigger than right is located. Since the keys are in order, this is a sequential read within a node, and a linked list traversal for subsequent nodes.

RANGE(root, 26, 40)

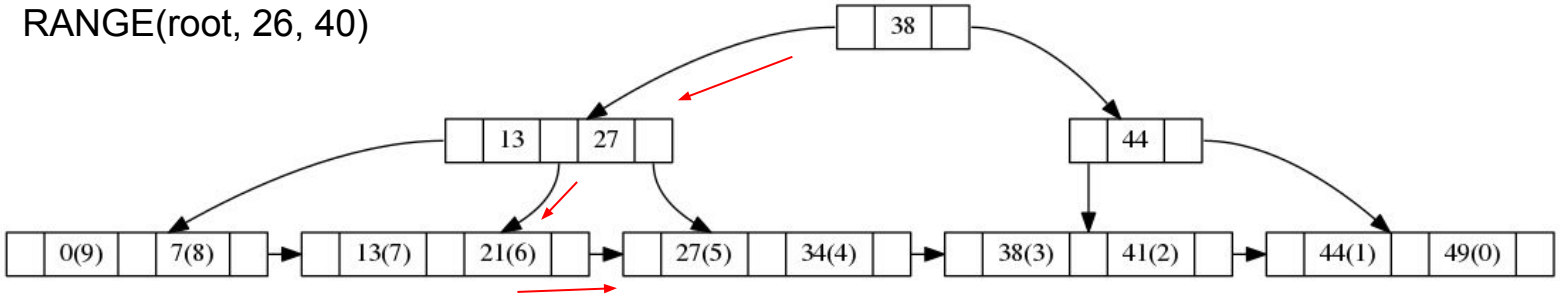


Results:

Describe how you insert data, delete data, and probe for ranges.

RANGE(node root, int left, int right) - A range is probed by first descending to the leaves using left as the target. Once the leaves are reached, the values are collected until a key bigger than right is located. Since the keys are in order, this is a sequential read within a node, and a linked list traversal for subsequent nodes.

RANGE(root, 26, 40)

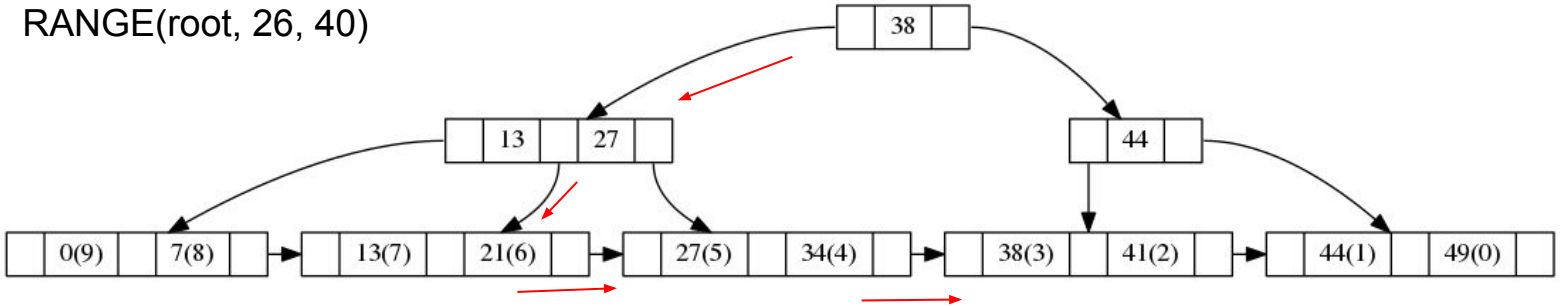


Results:

Describe how you insert data, delete data, and probe for ranges.

RANGE(node root, int left, int right) - A range is probed by first descending to the leaves using left as the target. Once the leaves are reached, the values are collected until a key bigger than right is located. Since the keys are in order, this is a sequential read within a node, and a linked list traversal for subsequent nodes.

RANGE(root, 26, 40)

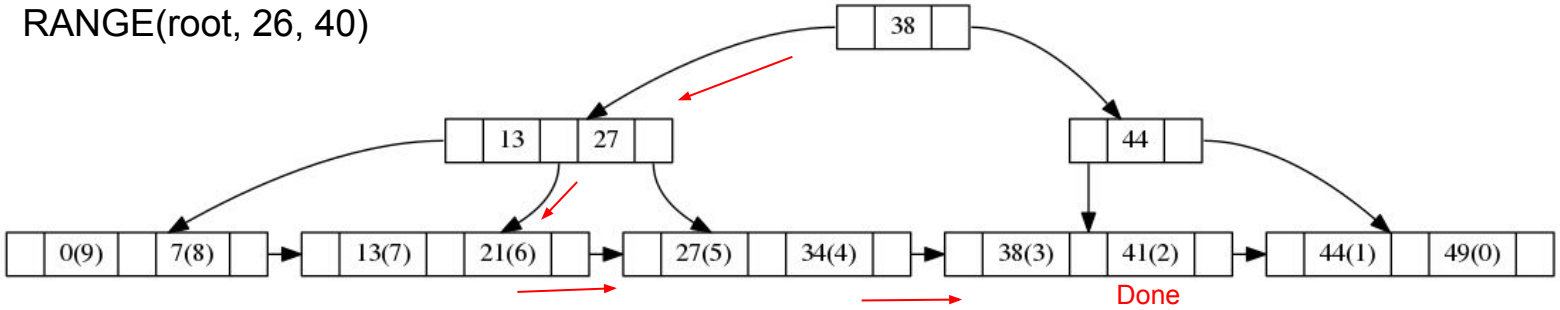


Results: 5, 4

Describe how you insert data, delete data, and probe for ranges.

RANGE(node root, int left, int right) - A range is probed by first descending to the leaves using left as the target. Once the leaves are reached, the values are collected until a key bigger than right is located. Since the keys are in order, this is a sequential read within a node, and a linked list traversal for subsequent nodes.

RANGE(root, 26, 40)



Results: 5, 4, 3

What is the cost in terms of M2 misses for each of [I/D & range probe]?

INSERT/DELETE - Since we sized each node to be one M2 block, each recursive call down the tree will be exactly one miss. So for a tree of N elements, insert and delete will in the best case be:

$$\lceil log_b N \rceil$$

where b = FANOUT. If all of the nodes are full, an insert will cause one additional miss at each level as additional space is requested for the new nodes or neighbor nodes are used to balance. This makes the worst case for both insert and delete a factor of 2 greater than the base.

RANGE - Range will use the same descent as before but also has a miss for each leaf that must be traversed. The number of leaves needed to be visited is related to the selectivity of the range and the fanout of the tree.

$$\lceil log_b N \rceil + S \cdot \frac{N}{b}$$

What kind of optimizations would you do for a scan and index?

Scan - Compression; shared scans; SIMD; vectorized processing. Compressed data means fewer memory misses and data movement. Shared scans keep data in the lower part of the memory hierarchy while multiple operations are performed, reducing misses. Single instruction multiple data (SIMD) issues one instruction to retire more than one tuple. Vectorized processing would help somewhat with data locality but not be a significant optimization on a single core machine.

Index - The opportunities to optimize tree are fewer. You could tune for fanout and intra-node search. Tuning for fanout would be done by sweeping different values of FANOUT to find the one with the lowest latency. Rather than use binary search within each node you could use a linear search which can be faster on small arrays.

Would you change your answer to part 1.5?

It depends. If you chose to use an index in part 1.5 the added optimizations to scan and relative lack to index means that scan may become faster for some workloads. In particular, scans can benefit from the addition of concurrent queries where there is little advantage for index queries to be grouped. So, the better scan gets and the more concurrency in the system, the more likely you are to use only a scan.