# CS222/CS122C: Principles of Data Management

UCI, Fall 2019
Notes #08

# Indexing Performance, Composite Indexes, Index-Only Plans

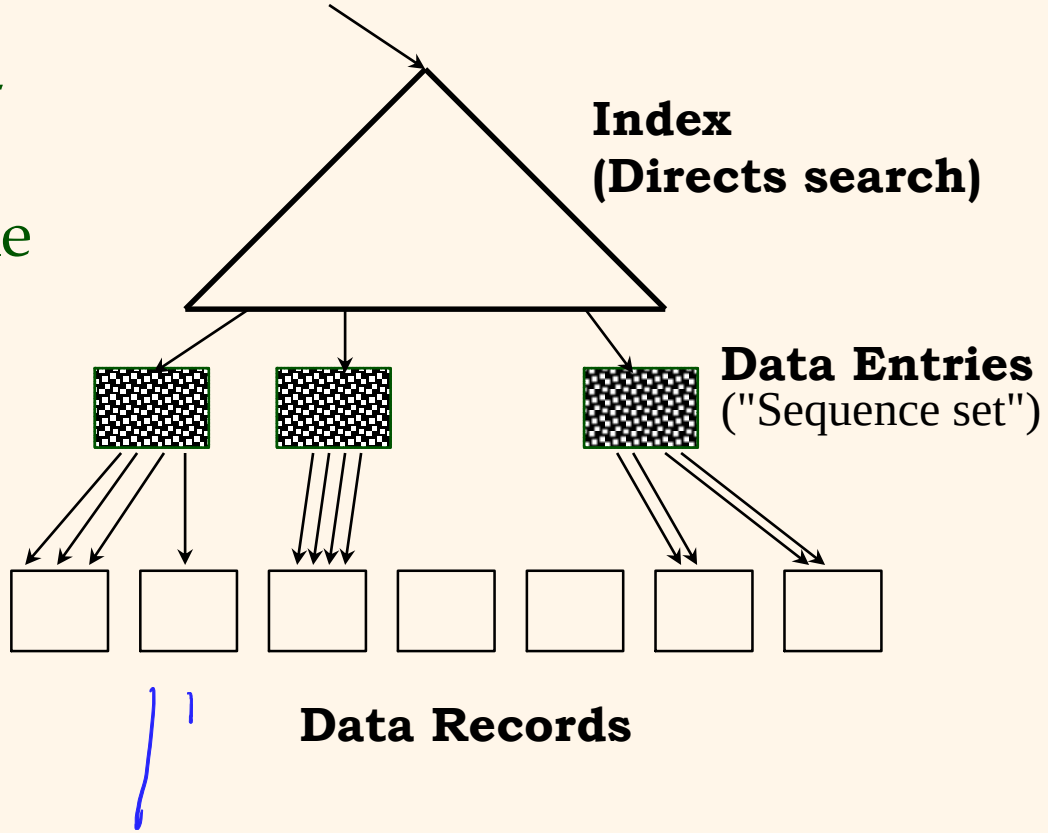
Instructor: Chen Li

# *B+ Tree as "Sorted Access Path"*

- Scenario: Table to be retrieved in some order has a B+ tree index on the ordering column(s).
- Idea: Retrieve records in order by traversing the B+ tree's leaf pages.
- ***Q: Is this a good idea?***
- Cases to consider:
  1. B+ tree is clustered → ***Good idea!***
  2. B+ tree is not clustered → ***Can be a very bad idea!***

# *Case 1: Clustered B+ Tree*

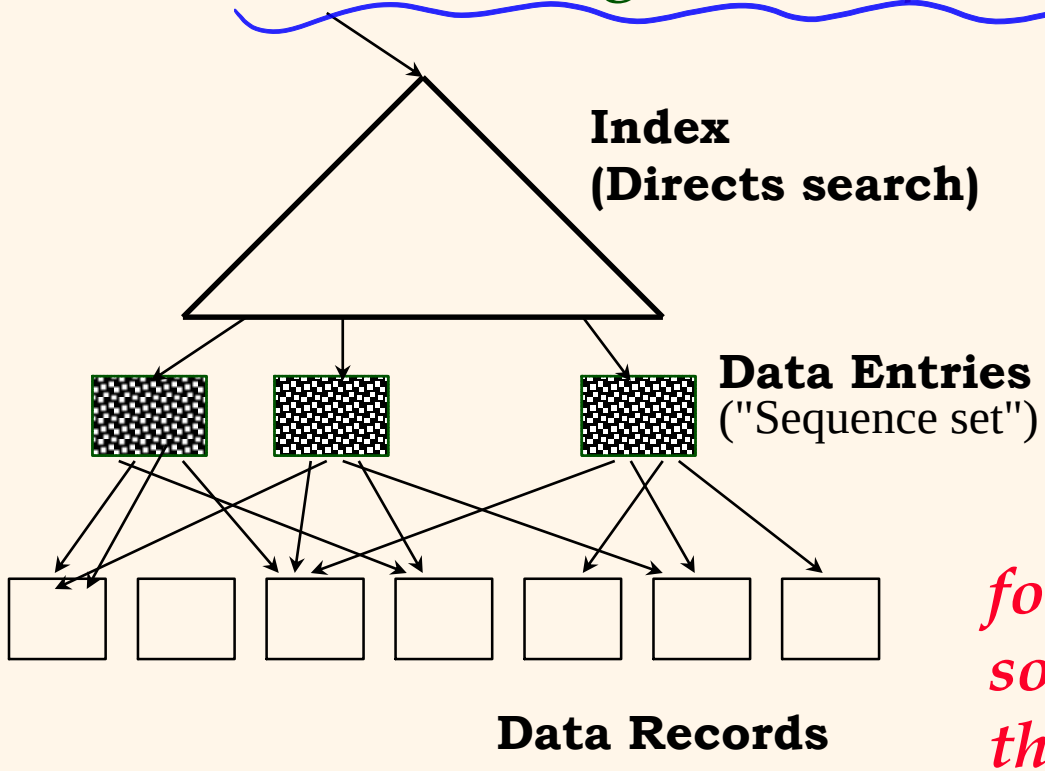
- Cost: Descend to the left-most leaf, then just scan leaves, as they contain the records. (Alternative 1)
- If rids (Alternative 2) are used? Additional cost to fetch data records, but *each page fetched just once* thanks to clustering.




***Always better than external sorting!***

# *Case 2: Unclustered B+ Tree*

- Implies Alternative (2) for data entries; each entry contains *RID* of a data record. In general, will be doing *one I/O per data record*!




Need lots of disk IO

**Trick:** ***Scanning for the RIDs 1st and sorting before using them can help...!***

# *Index Selection: Workload Analysis*

- For each query in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes appear in selection/join conditions, and how *selective* are these conditions (few *vs.* many results)?
- For each update in the workload:
  - Which attributes are involved in selection/join conditions? Again, how selective are these conditions likely to be?
  - What is the type of update (INSERT/DELETE/UPDATE), and which attributes are affected?

# *Choice of Indexes*

- What indexes should we create?
  - Which relations should have indexes? What field(s) should be the search key? Should we build several indexes for a given file/table?
- For each index, what kind of index should it be?
  - Clustered? Hashed? B+ tree? ...?

# *Choice of Indexes (cont.)*

- One approach: Consider the most important queries in turn. Consider the best query evaluation plan given the current indexes, and see if a better plan is possible with an additional index. If so, create it.
  - Obviously, this implies that we must understand how a DBMS evaluates queries and creates its query plans!
  - For today, we'll focus mostly on simple 1-table queries.
- Before creating an index, must also consider its potential impact on updates in the workload!
  - Trade-off: Indexes can make queries go faster, but make updates go slower. They require disk space, too.

# *Index Selection Guidelines*

- Attributes in WHERE clause are candidates for index keys.
  - Exact match condition suggests a hashed index.
  - Range query suggests a tree index.
    - Clustering especially useful for range queries; might also help on equality queries if there are many duplicates. (*Q:* See why?)
- Multi-attribute search keys should be considered when a WHERE clause contains several conditions.
  - Order of attributes is important for range queries.
  - Such indexes can actually enable index-only query plans for important queries.
    - Note: For index-only strategies, clustering is not important!!
- Try to choose indexes that will benefit as many queries as possible. Since only one index per relation can be clustered, choose it based on important queries that can benefit the most from clustering.

# *Examples of Clustered Indexes*

- B+ tree index on E.age can be used to get qualifying tuples.
  - How selective is the condition?
  - And is the index clustered?

```
SELECT  E.dno
FROM  Emp E
WHERE  E.age>40
```

# *Examples of Clustered Indexes*

- Consider the GROUP BY query.
  - If many tuples have *E.age* > 10, using *E.age* index and sorting the retrieved tuples (for grouping) may be costly.
  - Clustered *E.dno* index may be better!

```
SELECT  E.dno,  COUNT (*)
FROM  Emp E
WHERE  E.age>10
GROUP BY E.dno
```

# *Examples of Clustered Indexes*

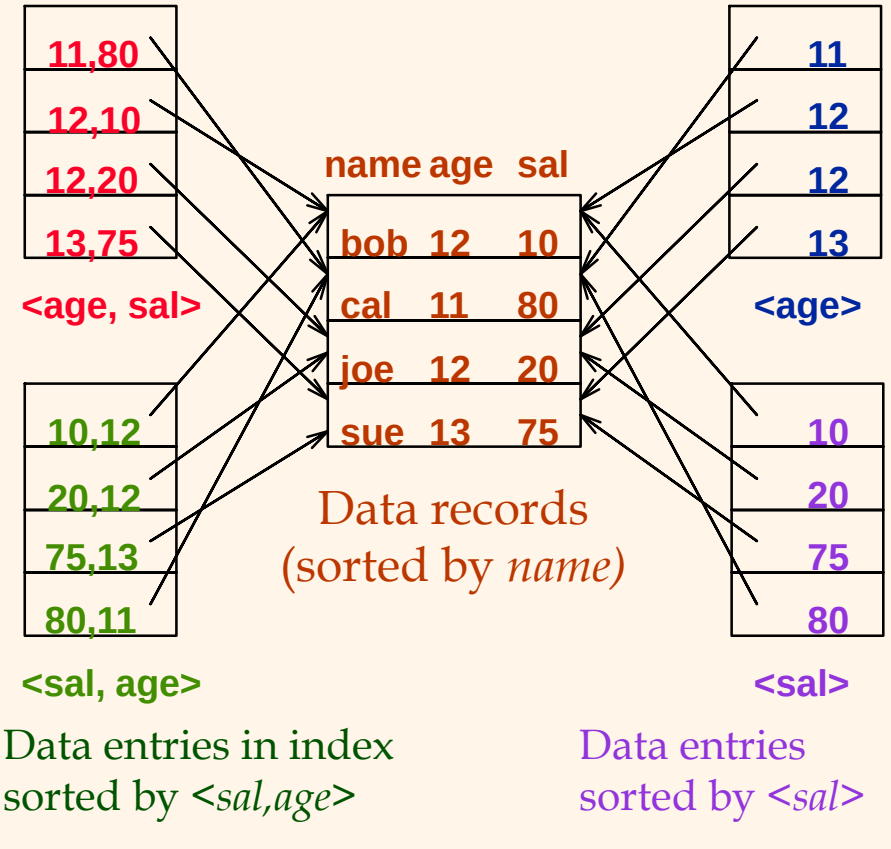
- Equality with duplicates:
  - Clustering on *E.hobby* helps!

```
SELECT  E.dno
FROM  Emp E
WHERE  E.hobby='Stamps'
```

# *Indexes with Composite Search Keys*

- *Composite Search Keys*: Search on a combination of fields.
  - Equality query: Every field value is equal to a constant value. E.g. wrt <sal,age> index:
    - age=20 and sal=75
  - Range query: Some field value is a range, not a constant. E.g. again wrt <sal,age> index:
    - age=20; or age=20 and sal > 10
- Data entries in index sorted by search key(s) to support such range queries.
  - Lexicographic order

Various possible composite key indexes using lexicographic order.



# *Composite Search Keys*

- To retrieve Emp records with *age*=30 AND *sal*=4000, an index on <*age,sal*> would be better than an index on *age* or *sal* alone.
  - Choice of index key orthogonal to clustering, etc.
- If condition is: 20<*age*<30 AND 3000<*sal*<5000:
  - Clustered tree index on <*age,sal*> or <*sal,age*> is best.
- If condition is: *age*=30 AND 3000<*sal*<5000:
  - Clustered <*age,sal*> index much better than <*sal,age*> index! (See why?)
- Composite indexes generally larger, and may be updated more often.

# *Index-Only Query Plans* = Without retrieving record from heap file

- A number of queries can be answered w/o retrieving *any* (!) tuples from one or more of the relations involved, if a suitable index is available

*<E.dno> index*

```
SELECT  E.dno, COUNT(*)
FROM  Emp E
GROUP BY  E.dno
```

*<E.dno,E.sal> Tree index!*

```
SELECT  E.dno, MIN(E.sal)
FROM  Emp E
GROUP BY  E.dno
```

*<E. age,E.sal>* or *<E.sal, E.age> Tree index!*

```
SELECT AVG(E.sal)
FROM  Emp E
WHERE  E.age=25 AND
  E.sal BETWEEN 3000 AND 5000
```

# *Index-Only Plans (Contd.)*

- Here, index-only plans possible if the key is <dno,age> or we have a tree index with key <age,dno>
  - What is in each index entry?
  - Which is better?
  - What if we consider the second query?

```
SELECT  E.dno,  COUNT (*)
FROM  Emp E
WHERE  E.age=30
GROUP BY E.dno
```

```
SELECT  E.dno,  COUNT (*)
FROM  Emp E
WHERE  E.age>30
GROUP BY E.dno
```

# *Files and Indexes: Summary*

- Many alternative file organizations exist, with each being appropriate in some situations.
- If selection queries are frequent, building an *index* is important to avoid file scans.
  - Hash-based indexes (only) good for equality search. (Though could still scan index leaves...)
  - Sorted files and tree-based indexes best for range search; also good for equality search. (Files rarely kept sorted in practice; B+ tree index is better.)
- Index is a collection of data entries plus a way to quickly find entries with given key values.

# *Summary (cont.)*

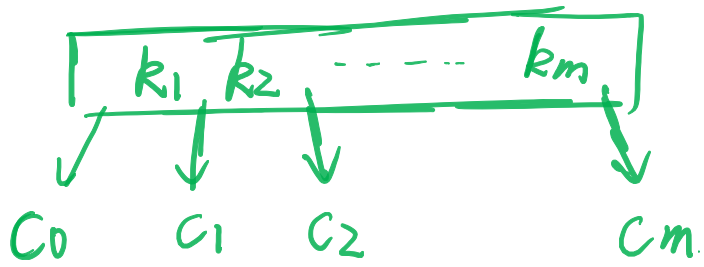
- Data entries can be actual data records, <key, rid> pairs, or <key, rid-list> pairs.
  - Choice orthogonal to *indexing technique* used to locate data entries with a given key value.
- Can have several indexes on a given file of data records, each with a different search key.
- Indexes can be classified as clustered vs. unclustered, primary *vs*. secondary, and dense *vs*. sparse. Differences have important consequences for index utility/performance.

# *Summary (cont.)*

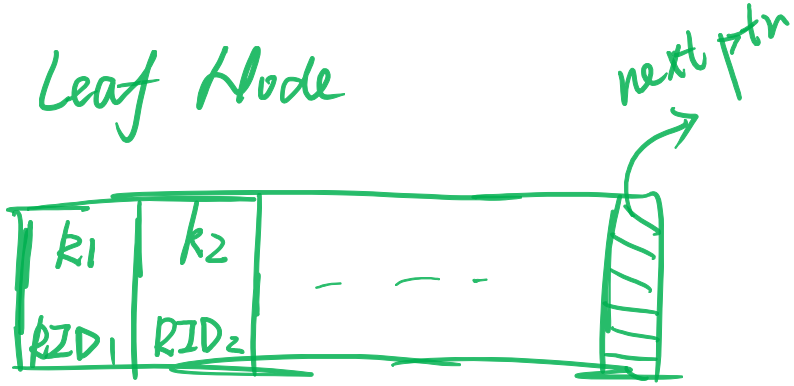
- Understanding the nature of the *workload* for the application, and the performance goals, is essential to good physical DB design (*a.k.a.* index selection).
  - What are the important queries and updates? What attributes/relations are involved?
- Indexes must be chosen to speed up important queries (and perhaps some updates too!).
  - Maintenance overhead on updates to indexed fields.
  - Choose indexes that help many queries, if possible.
  - Build indexes in support of index-only strategies.
  - Clustering is an important decision; only one index on any given relation can be clustered!
  - Order of fields in composite keys can be very important.

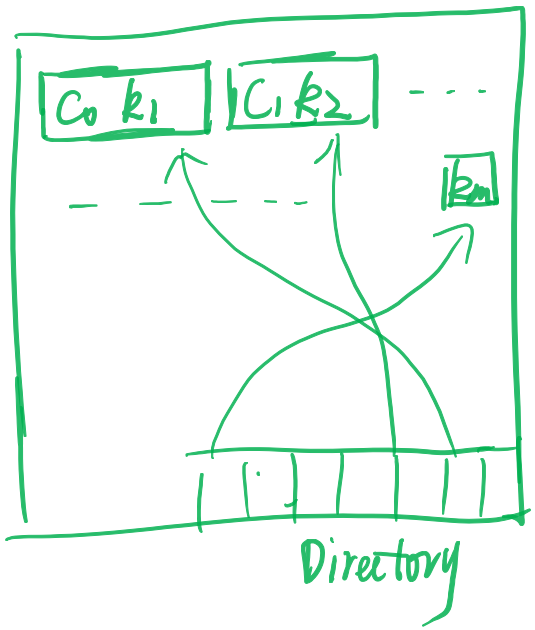
# Page format:

## 1) Intermediate Node:

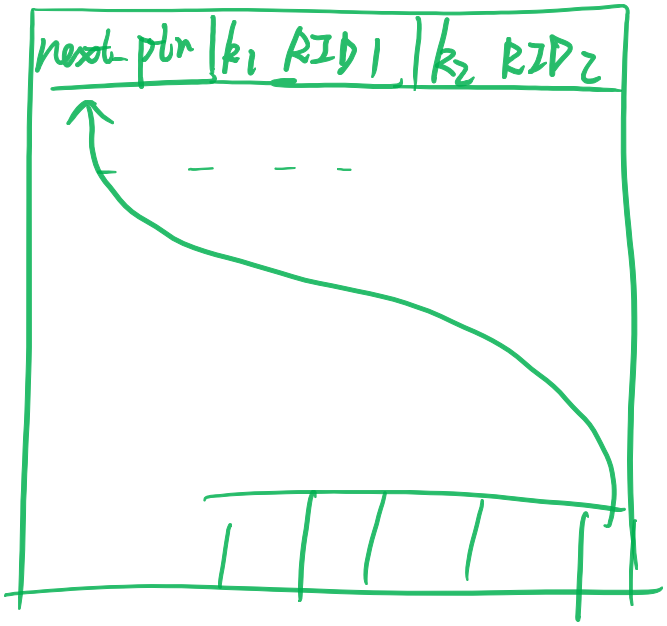



## 2) Leaf Node

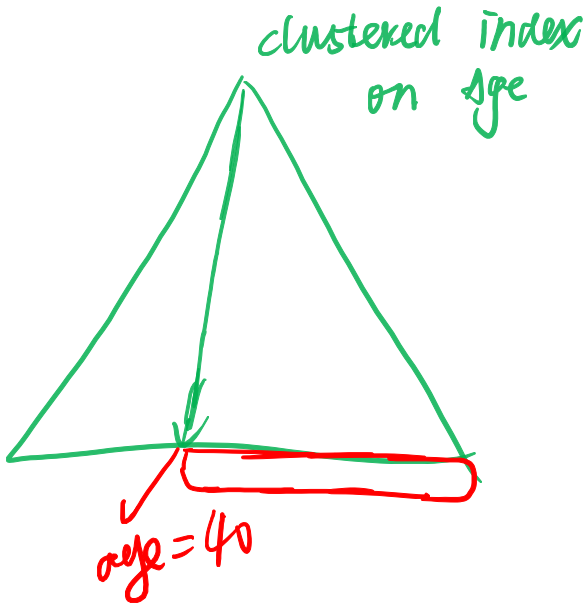


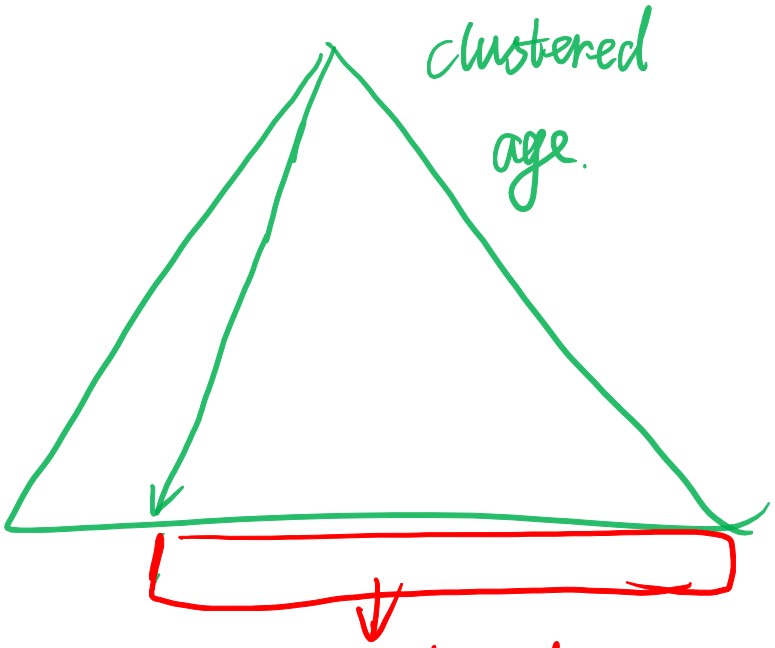

Treat $C_i, k_{i+1}$ as a record.



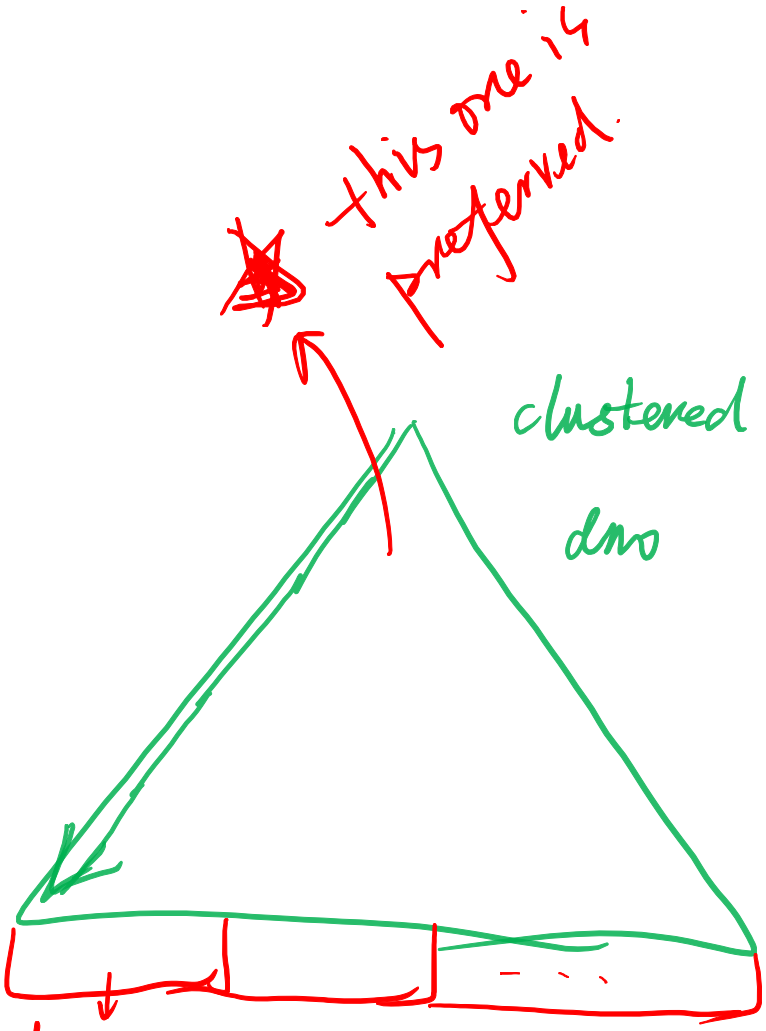
Q1 : Select Dno
From Emp
where age > 40



$Q_2$ = Select dno, count(*)
From Emp
Where age > 10
Group by dno.



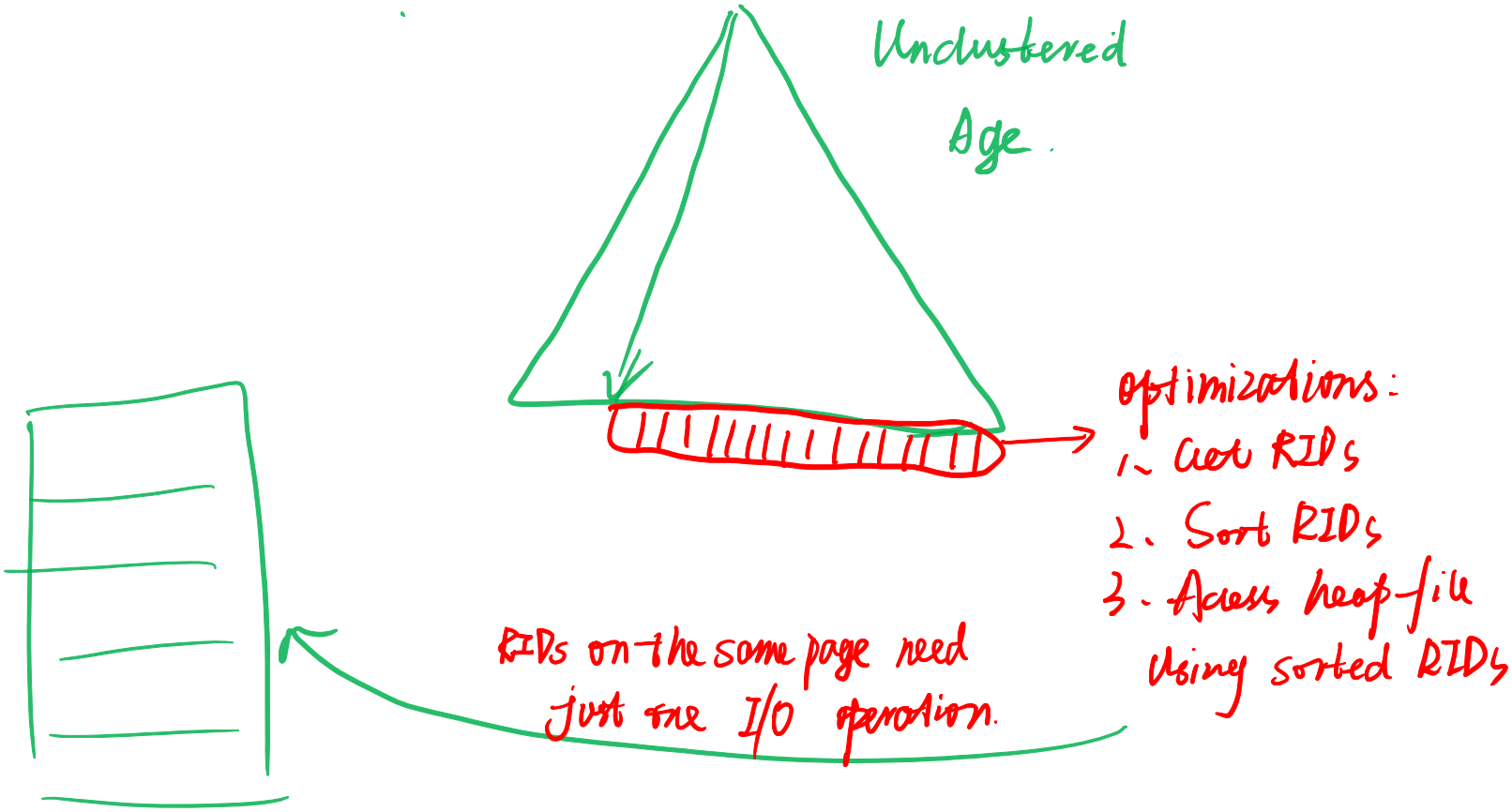

Need another mechanism to do the Group by operation.




Jitter age inside each block.

Q3: Select dno, count(*)
From Emp
Where age > 10

# Unclustered Index



- Composite index

- Index-only

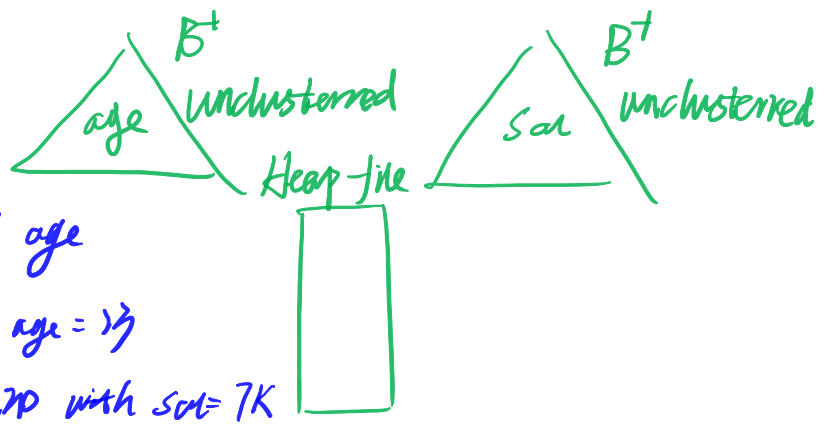
Emp (id, age, sal, ...)

[10,70] [3K-9K]

R1: select *
from Emp
where age = 23
AND sal = 7K

plan1: scan the heap file, check each record with two condition

plan2: Use Age $B^+$ tree, find RID with age = 23
Use rid to retrieve record in heap file
return record with sal = 7K



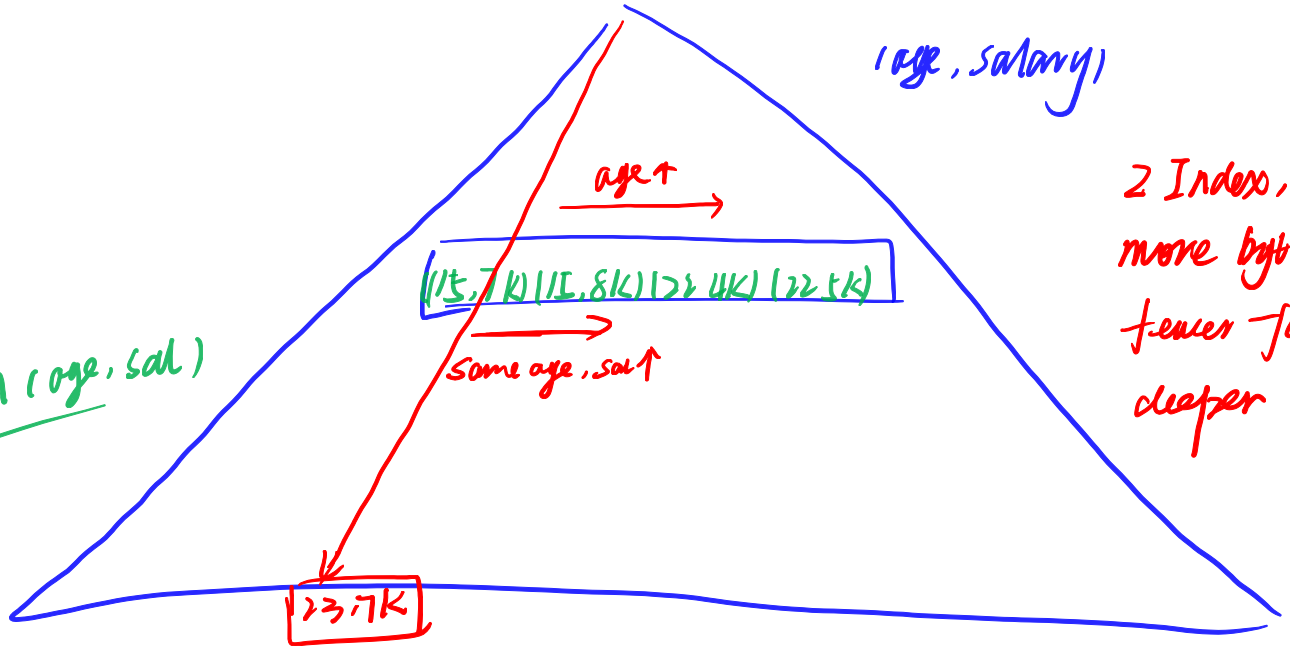

plan3: Similar to plan2. by reversing sal and age

plan4: Use age $B^+$ tree to retrieve RID with age = 23
And then use sal $B^+$ tree to retrieve RID with sal = 7K
Intersect 2 sets.

Composite Index: '<' ⇒ total order of keys ⇒ $B^+$ tree

(age, sal)

$(age_1, sal_1) < (age_2, sal_2)$ if $age_1 < age_2$ or $age_1 = age_2 \wedge sal_1 < sal_2$



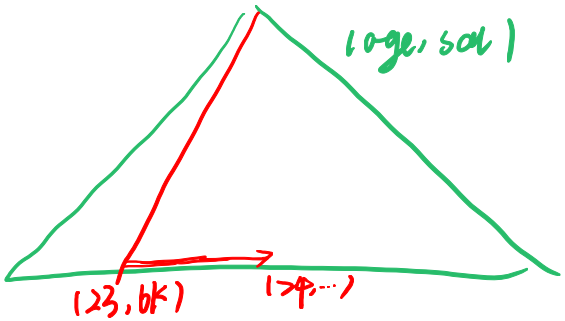

Condition (age, sal)

2 Indexs, each node needs more bytes, tree has fewer fanout and get deeper

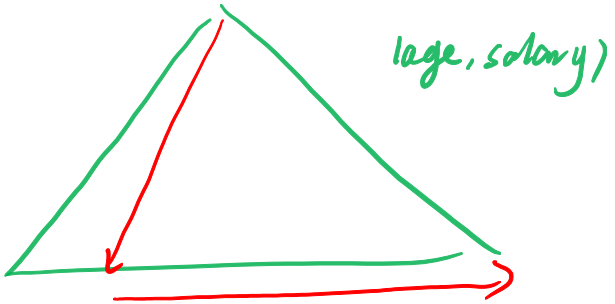
plan 5: Use the $B^+$ tree to retrieve RID for (23,7K) and the access the heap file to retrieve record

| | intermediate node | | leaf node |
|---|---|---|---|
| Single $B^+$ tree | 4 (age) + | 4 (ptr) | 4 + 8 (age) (RID) |
| composite $B^+$ tree | 8 (age, sal) + | 4 (ptr) | 8 + 8 (age, sal) (RID) |

Q2: age=23, sal>6K:



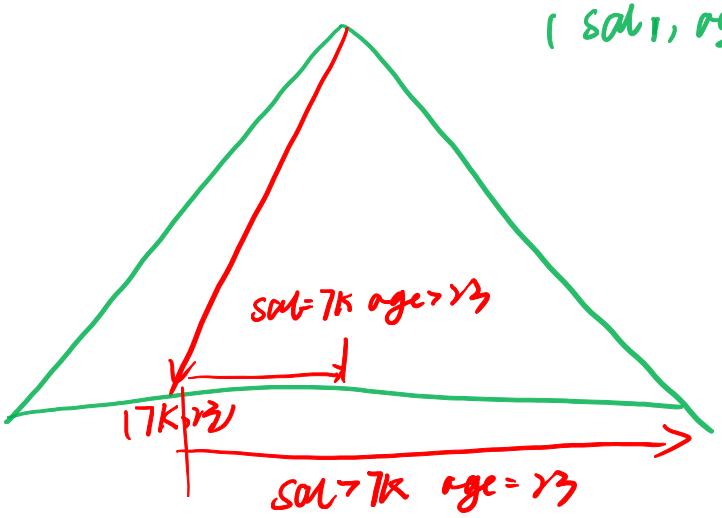

Q3 age>>23, sal=6K




need to scan all record age>>23

Use (sal, age) B+ tree → better

Composite Index: (sal, age)

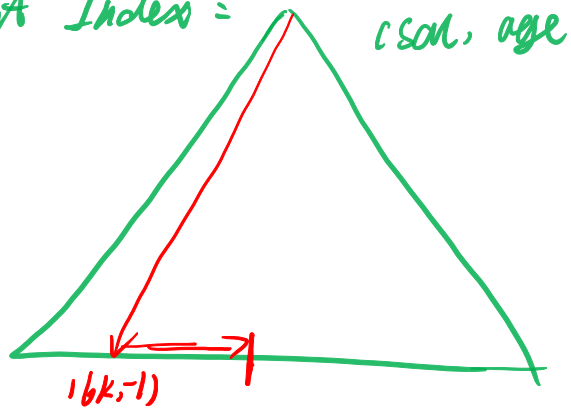



(sal1, age1) < (sal2, age2) if sal1 < sal2 or

sal1 = sal2 and age1 < age2

Order of Composite Indexes is IMPORTANT!

Composit Index = (sal, age)

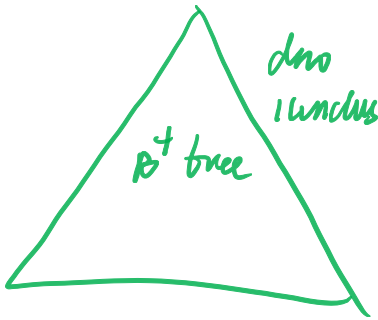


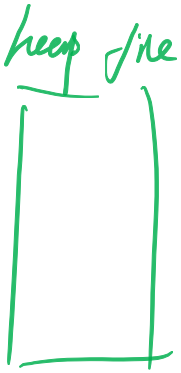

sal = 6k

age = > 3

- Index-only query plan

Q1: select count(*)
from Emp



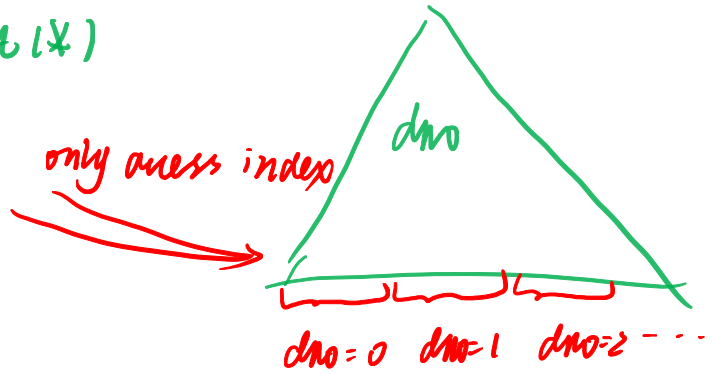

- Scan the leaf node of B+ tree
- do the counting

Deal with 'null' value: Depend on how we index null values in B+ tree.

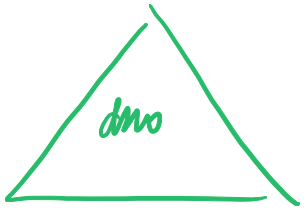
↓

eg. put all null value at beginning

Q2: select dno count(*)
from Emp
group by dno




Q3: select dno, min(sal)
from Emp
group by dno

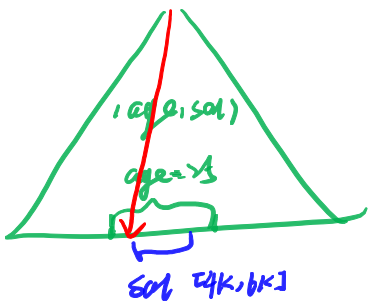



Q4: Select Avg(sal)

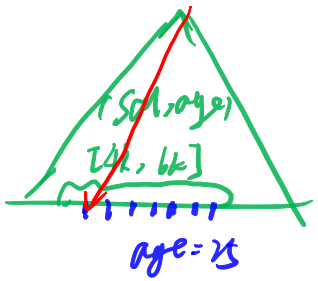
from Emp

where age = 25 & sal in [4k, 6k]

a)




b)



Q5: Select dno min(sal)
from emp
Group by dno: