

# CompSci 516

# Database Systems

## Lecture 9

### Index Selection and External Sorting

Instructor: Sudeepa Roy

# Announcements

- Private project threads created on piazza
  - Please use these threads (and not emails) for all communications on your project
- Project proposal deadline extended until 9/28 Thursday 5 pm
- HW2 to be posted soon
  - Will receive email with your CS email ID for AWS accounts from Keping/Yilin

# Today

- Index selection
- External sort

# Reading Material

- Index: as in Lecture 7/8
- External sorting:
- [RG]
  - External sorting: Chapter 13
- [GUW]
  - Chapter 15.4.1

Acknowledgement:

The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.

# Selection of Indexes

# Different File Organizations

We need to understand the importance of appropriate file organization and index

Search key =  $\langle \text{age}, \text{sal} \rangle$

Consider following options:

- Heap files
  - random order; insert at end-of-file
- Sorted files
  - sorted on  $\langle \text{age}, \text{sal} \rangle$
- Clustered B+ tree file
  - search key  $\langle \text{age}, \text{sal} \rangle$
- Heap file with unclustered B<sup>+</sup>-tree index
  - on search key  $\langle \text{age}, \text{sal} \rangle$
- Heap file with unclustered hash index
  - on search key  $\langle \text{age}, \text{sal} \rangle$

# Possible Operations

- Scan
  - Fetch all records from disk to buffer pool
- Equality search
  - Find all employees with age = 23 and sal = 50
  - Fetch page from disk, then locate qualifying record in page
- Range selection
  - Find all employees with age > 35
- Insert a record
  - identify the page, fetch that page from disk, inset record, write back to disk (possibly other pages as well)
- Delete a record
  - similar to insert

# Understanding the Workload

- A workload is a mix of **queries** and **updates**
- For each query in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
- For each update in the workload:
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
  - The type of update (INSERT/DELETE/UPDATE), and the attributes that 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 each index, what kind of an index should it be?
  - Clustered? Hash/tree?

# More on Choice of Indexes

- One approach:
  - Consider the most important queries
  - Consider the best plan using the current indexes
  - 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 **query evaluation plans**
  - We will learn query execution and optimization later - For now, we discuss simple 1-table queries.
- Before creating an index, must also consider the impact on updates in the workload

# Trade-offs for Indexes

- Indexes can make
  - queries go faster
  - updates slower
- Require disk space, too

# Index Selection Guidelines

- Attributes in WHERE clause are candidates for index keys
  - Exact match condition suggests hash index
  - Range query suggests tree index
  - Clustering is especially useful for range queries
    - can also help on equality queries if there are many duplicates
- Try to choose indexes that benefit as many queries as possible
  - Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering
- Multi-attribute search keys should be considered when a WHERE clause contains several conditions
  - Order of attributes is important for range queries
- Note: clustered index should be used judiciously
  - expensive updates, although cheaper than sorted files

# Examples of Clustered Indexes

- B+ tree index on E.age can be used to get qualifying tuples
- How selective is the condition?
  - everyone > 40, index not of much help, scan is as good
  - Suppose 10% > 40. Then?
- Depends on if the index is clustered
  - otherwise can be more expensive than a linear scan
  - if clustered, 10% I/O (+ index pages)

What is a good indexing strategy?

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

Which attribute(s)?  
Clustered/Unclustered?  
B+ tree/Hash?

# Examples of Clustered Indexes

## Group-By query

- Use *E.age* as search key?
  - Bad If many tuples have *E.age* > 10 or if not clustered....
  - ...using *E.age* index and sorting the retrieved tuples by *E.dno* may be costly
- Clustered *E.dno* index may be better
  - First group by, then count tuples with age > 10
  - good when age > 10 is not too selective
- Note: the first option is good when the WHERE condition is highly selective (few tuples have age > 10), the second is good when not highly selective

What is a good indexing strategy?

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

Which attribute(s)?  
Clustered/Unclustered?  
B+ tree/Hash?

# Examples of Clustered Indexes

What is a good indexing strategy?

## Equality queries and duplicates

- Clustering on *E.hobby* helps
  - hobby not a candidate key, several tuples possible
- Does clustering help now?
  - (eid = key)
  - Not much
  - at most one tuple satisfies the condition

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

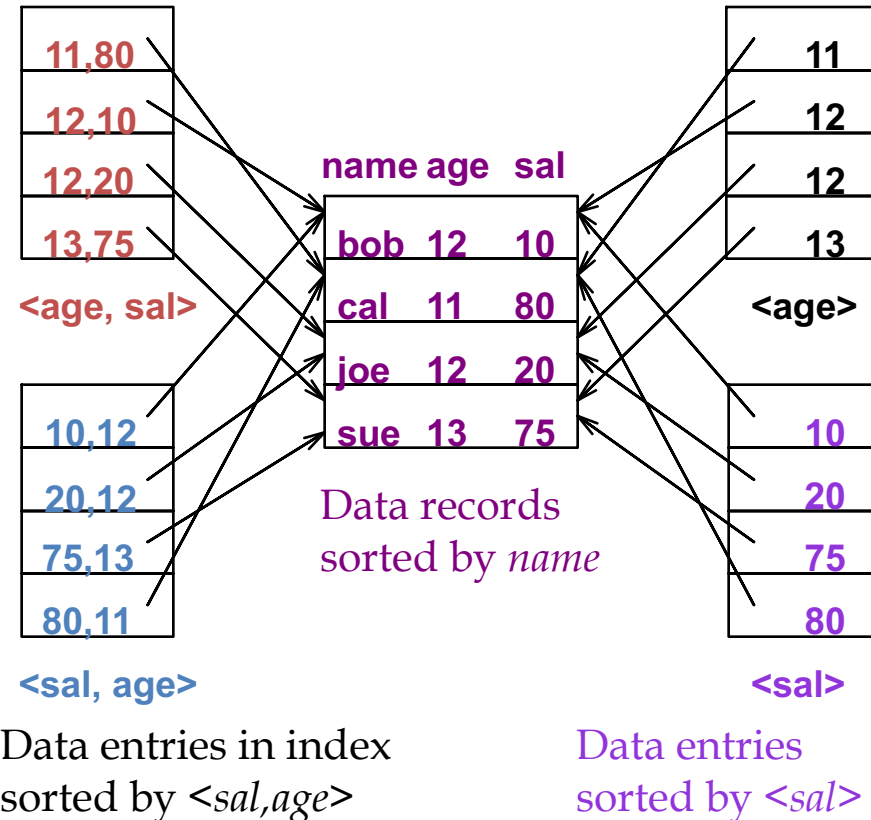
Which attribute(s)?  
Clustered/Unclustered?  
B+ tree/Hash?

```
SELECT E.dno
FROM Emp E
WHERE E.eid=50
```

# 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  $\langle \text{sal}, \text{age} \rangle$  index:
  - age=20 and sal =75
- **Range query:** Some field value is not a constant. E.g.:
  - sal > 10 – which combination(s) would help?
  - $\langle \text{age}, \text{sal} \rangle$  does not help
  - B+tree on  $\langle \text{sal} \rangle$  or  $\langle \text{sal}, \text{age} \rangle$  helps
  - has to be a prefix

Examples of composite key indexes using lexicographic order.





# Composite Search Keys

- To retrieve Emp records with  $age=30$  AND  $sal=4000$ , an index on  $\langle age, sal \rangle$  would be better than an index on  $age$  or an index on  $sal$ 
  - first find  $age = 30$ , among them search  $sal = 4000$
- If condition is:  $20 < age < 30$  AND  $3000 < sal < 5000$ :
  - Clustered tree index on  $\langle age, sal \rangle$  or  $\langle sal, age \rangle$  is best.
- If condition is:  $age=30$  AND  $3000 < sal < 5000$ :
  - Clustered  $\langle age, sal \rangle$  index much better than  $\langle sal, age \rangle$  index
  - more index entries are retrieved for the latter
- Composite indexes are larger, updated more often

# Index-Only Plans

- A number of queries can be answered without retrieving any tuples from one or more of the relations involved if a suitable index is available

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

*<E.dno>*

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

*<E.dno,E.sal>*

*Tree index!*

*<E.age,E.sal>*

*Tree index!*

- For index-only strategies, clustering is not important

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

# External Sorting

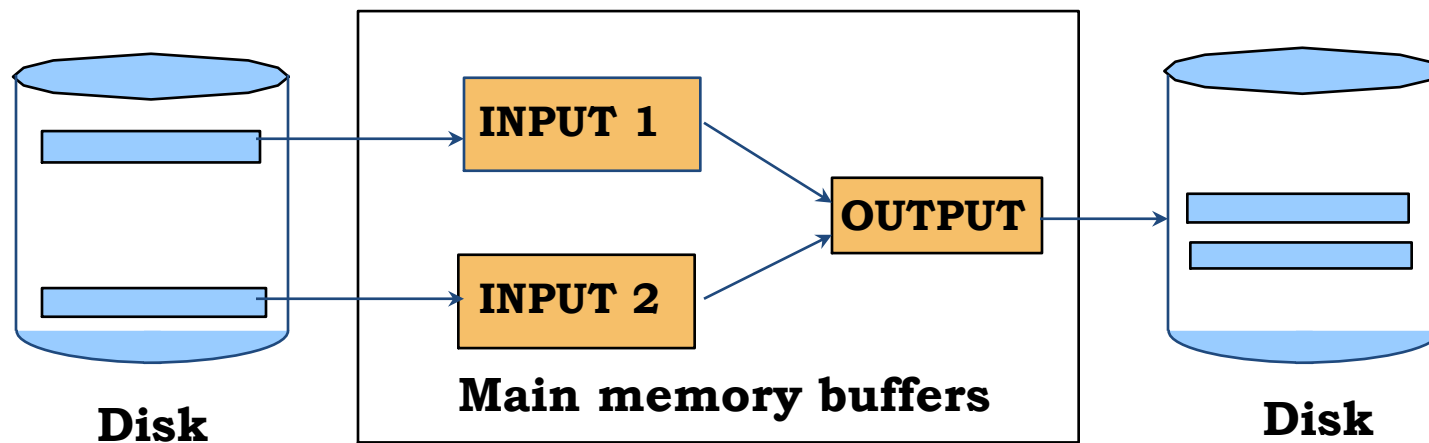
# Why Sort?

- A classic problem in computer science
- Data requested in sorted order
  - e.g., find students in increasing gpa order
- Sorting is first step in bulk loading B+ tree index
- Sorting useful for eliminating duplicate copies in a collection of records
- Sort-merge join algorithm involves sorting
- **Problem: sort 1Gb of data with 1Mb of RAM**
  - need to minimize the cost of disk access

quick review of mergesort on whiteboard

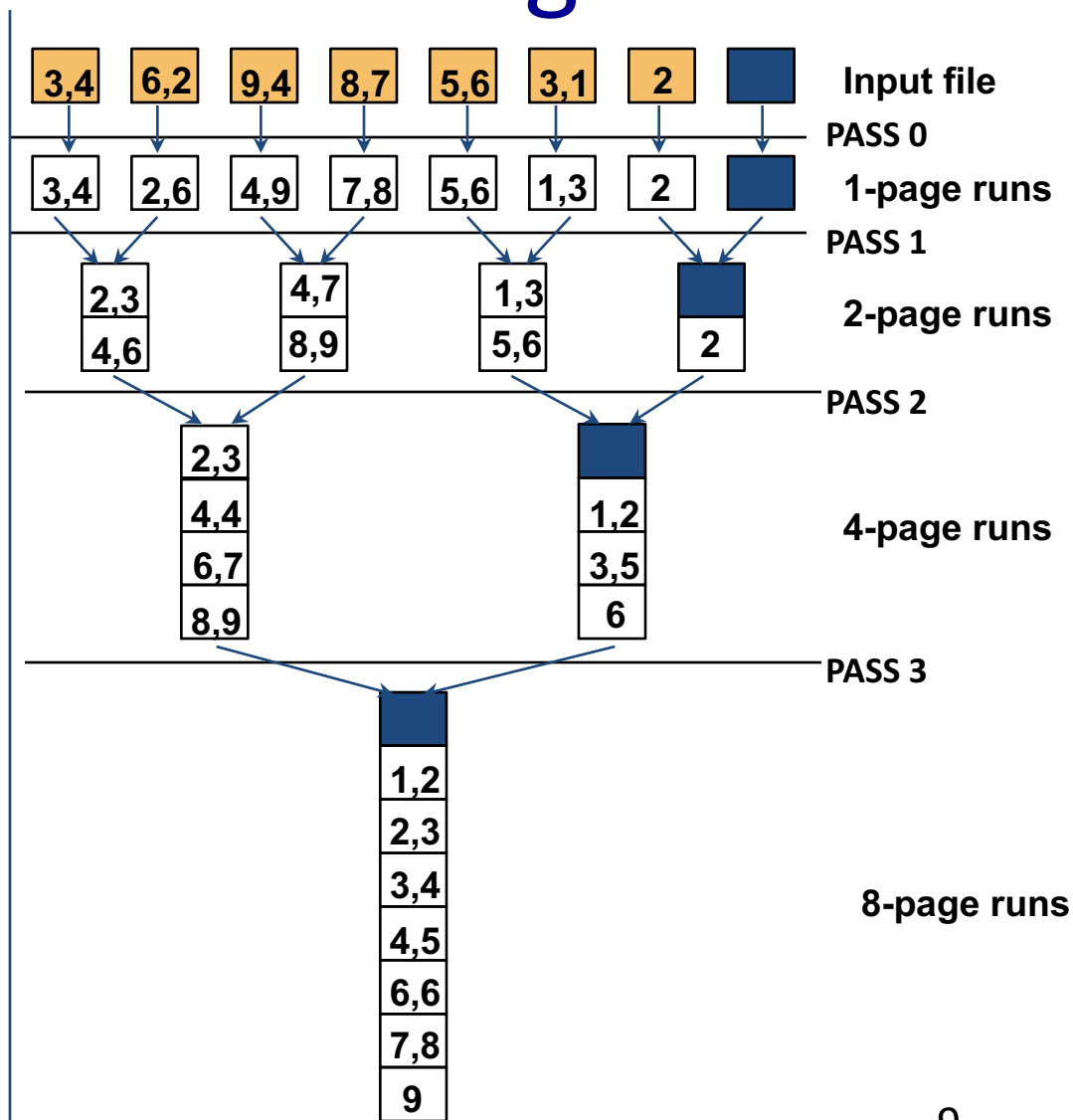
# 2-Way Sort: Requires 3 Buffers

- Suppose  $N = 2^k$  pages in the file
- Pass 0: Read a page, sort it, write it.
  - repeat for all  $2^k$  pages
  - only one buffer page is used
- Pass 1:
  - Read two pages, sort (merge) them using one output page, write them to disk
  - repeat  $2^{k-1}$  times
  - three buffer pages used
- Pass 2, 3, 4, ..... continue



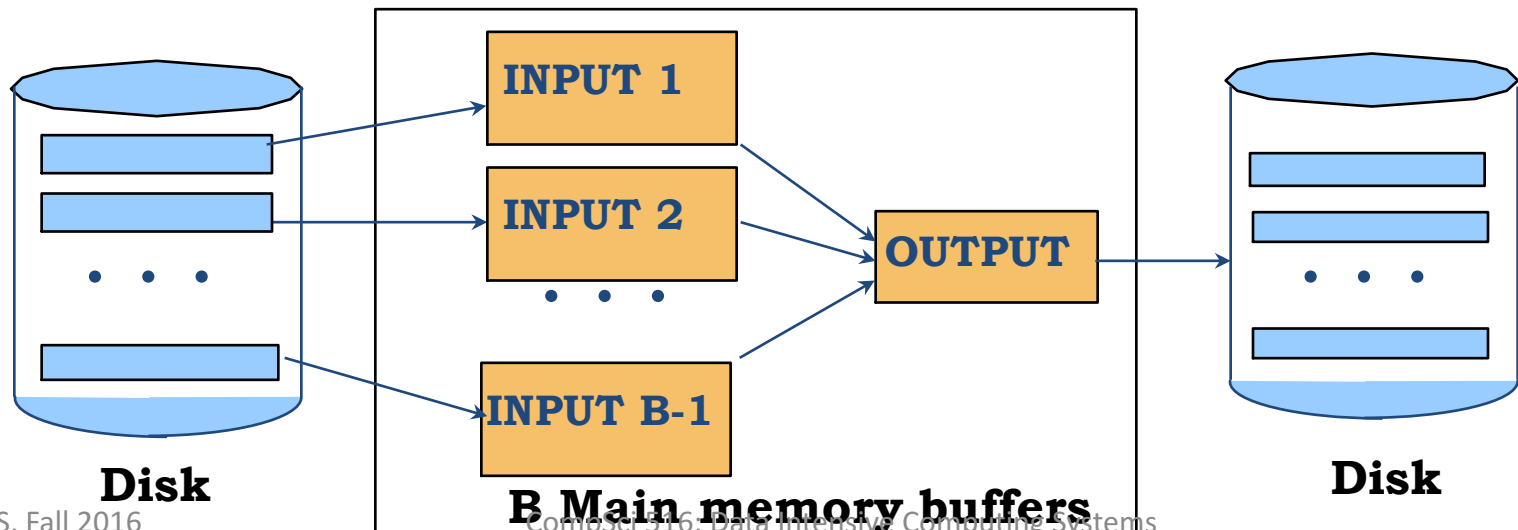
# Two-Way External Merge Sort

- Each sorted sub-file is called a **run**
  - each run can contain multiple pages
- Each pass we read + write each page in file.
- $N$  pages in the file,
- => the number of passes  $= \lceil \log_2 N \rceil + 1$
- So total cost is:  $2N(\lceil \log_2 N \rceil + 1)$
- Not too practical, but useful to learn basic concepts for external sorting



# General External Merge Sort

- Suppose we have more than 3 buffer pages.
- How can we utilize them?
- To sort a file with  $N$  pages using  $B$  buffer pages:
  - Pass 0: use  $B$  buffer pages:
    - Produce  $\lfloor N/B \rfloor$  sorted runs of  $B$  pages each.
  - Pass 1, 2, ..., etc.: merge  $B-1$  runs to one output page
    - keep writing to disk once the output page is full



# Cost of External Merge Sort

- Number of passes:  $1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$
- Cost =  $2N * (\text{\# of passes})$  – why 2 times?
- E.g., with 5 buffer pages, to sort 108 page file:
- Pass 0: sorting 5 pages at a time
  - $\lceil 108/5 \rceil = 22$  sorted runs of 5 pages each (last run is only 3 pages)
- Pass 1: 4-way merge
  - $\lceil 22/4 \rceil = 6$  sorted runs of 20 pages each (last run is only 8 pages)
- Pass 2: 4-way merge
  - (but 2-way for the last two runs)
  - $\lceil 6/4 \rceil = 2$  sorted runs, 80 pages and 28 pages
- Pass 3: 2-way merge (only 2 runs remaining)
  - Sorted file of 108 pages



# Number of Passes of External Sort

High B is good, although CPU cost increases

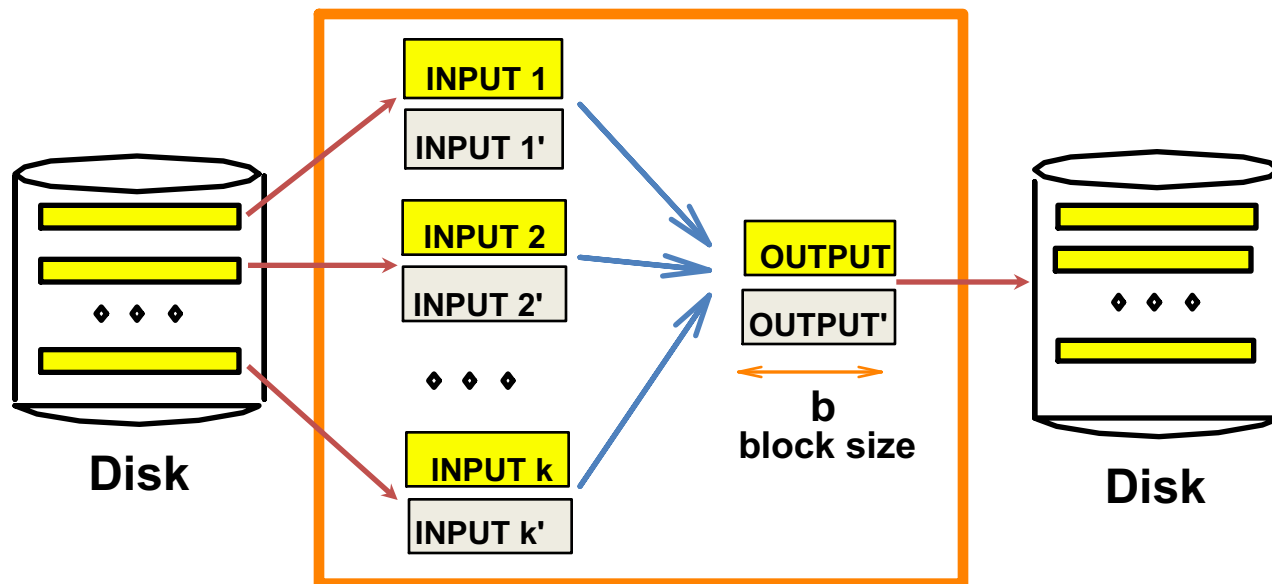
N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

# I/O for External Merge Sort

- If 10 buffer pages
  - either merge 9 runs at a time with one output buffer
  - or 8 runs with two output buffers
- If #page I/O is the metric
  - goal is minimize the #passes
  - each page is read and written in each pass
- If we decide to read a **block** of  $b$  pages sequentially
  - Suggests we should make each buffer (input/output) be a **block** of pages
  - But this will reduce fan-out during merge passes
    - i.e. not as many runs can be merged again any more
  - In practice, most files still sorted in **2-3 passes**

# Double Buffering

- To reduce CPU wait time for I/O request to complete, can prefetch into 'shadow block'.



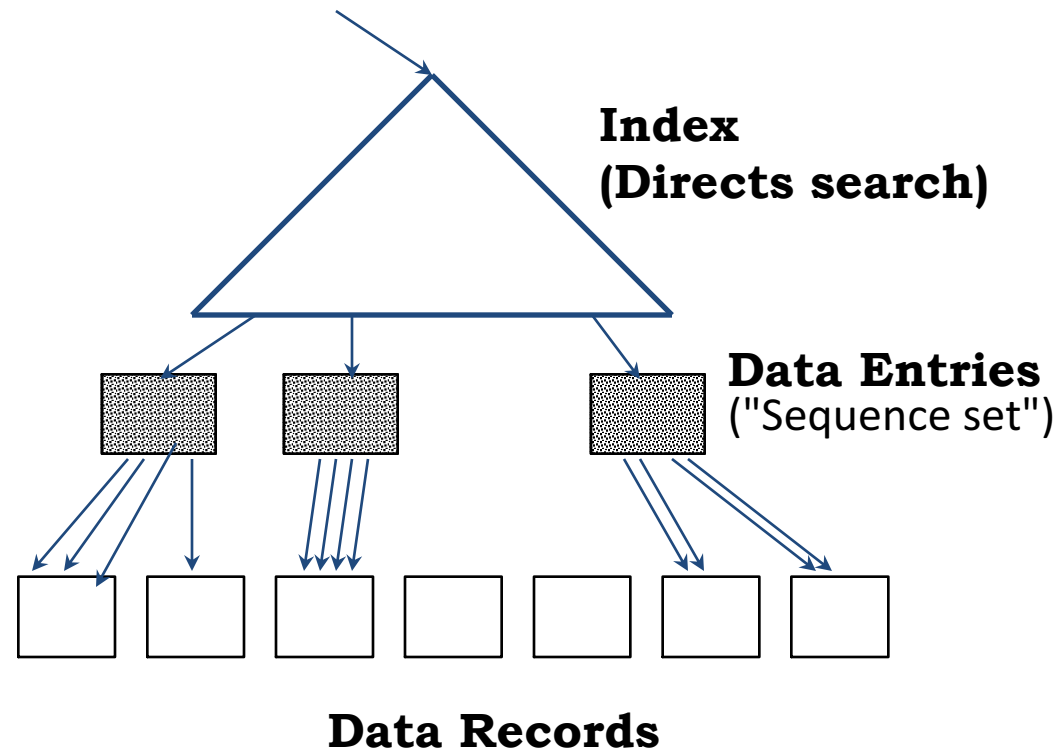
**B main memory buffers, k-way merge**

# Using B+ Trees for Sorting

- Scenario: Table to be sorted has B+ tree index on sorting column(s)
- Idea: Can retrieve data entries (then records) in order by traversing leaf pages.
- Is this a good idea?
- Cases to consider:
  - B+ tree is clustered: Good idea!
  - B+ tree is not clustered: Could be a very bad idea!

# Clustered B+ Tree Used for Sorting

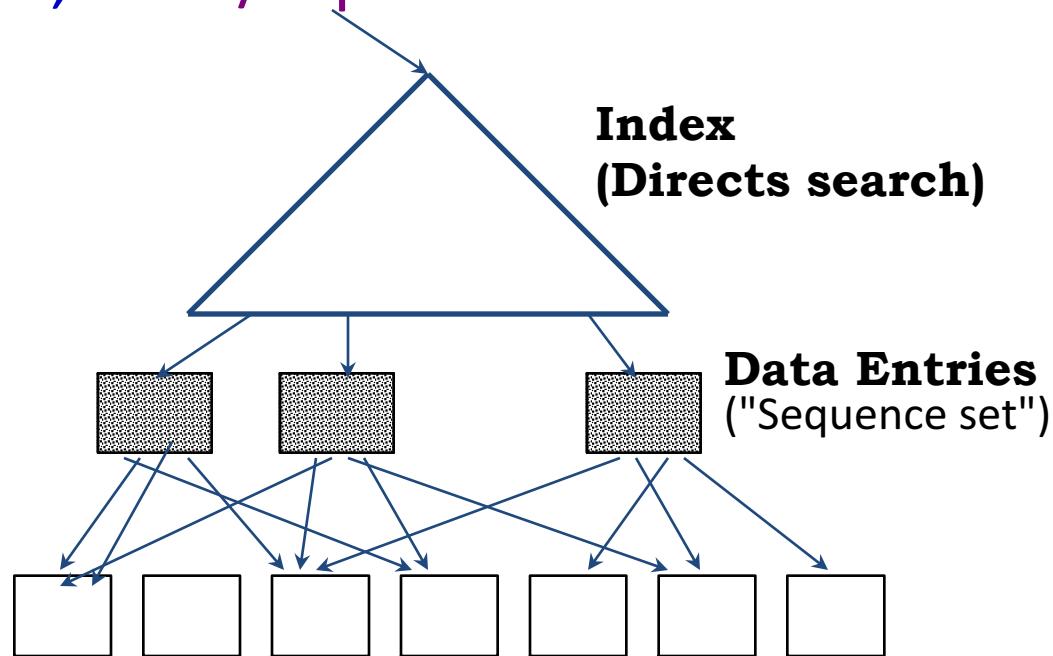
- Cost: root to the left-most leaf, then retrieve all leaf pages (Alternative 1)
- If Alternative 2 is used? Additional cost of retrieving data records: each page fetched just once



➡ *Always better than external sorting!*

# Unclustered B+ Tree Used for Sorting

- Alternative (2) for data entries; each data entry contains *rid* of a data record
- In general, one I/O per data record!



# Summary

- External sorting is important; DBMS may dedicate part of buffer pool for sorting!
- External merge sort minimizes disk I/O cost:
  - Pass 0: Produces **sorted runs** of size B (# buffer pages)
  - Later passes: **merge runs**
  - # of runs merged at a time depends on B, and block size.
  - Larger block size means less I/O cost per page.
  - Larger block size means smaller # runs merged.
  - In practice, # of runs rarely more than 2 or 3