CompSci 516
Data Intensive Computing Systems

Lecture 8
External Sorting
and
Operator Algorithms

Instructor: Sudeepa Roy

Announcement

- Homework 1
  - Due on Feb 10 (Wednesday), 11:59 pm
    - deadline extended for Chinese New Year
  - Check out clarifications and Q/A on Piazza
    - especially on “collaborators” in Q5b

What will we learn?

- Last lecture:
  - Hash-based indexing
    - Static and dynamic (extendible hashing, linear hashing)
  - Overview on query evaluation
- Next:
  - External Sorting
  - How individual relational operators are implemented in a DBMS

Reading Material

- [RG]
  - External sorting: Chapter 13
  - Query evaluation and operator algorithms: Chapter 12.2-12.5, 13, 14.1-14.3
- [GUW]
  - Chapter 14

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

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

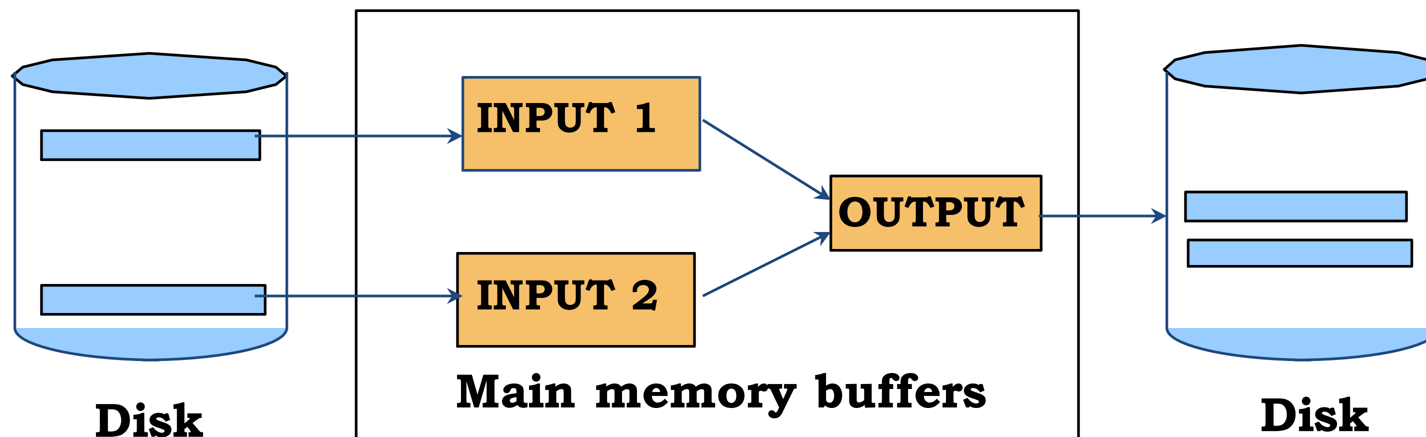
quick review of mergesort on whiteboard

A simple 2-way sort

- Not too practical, but useful to learn basic concepts for external sorting
- Utilizes only 3 pages of main memory
- Several sorted sub-files are generated at intermediate steps
- Each sorted sub-file is called a run
  - each run can contain multiple pages

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 them using one output page, write them to disk
  - repeat 2^{k-1} times
  - three buffer pages used
- Pass 2, 3, 4, ….. continue

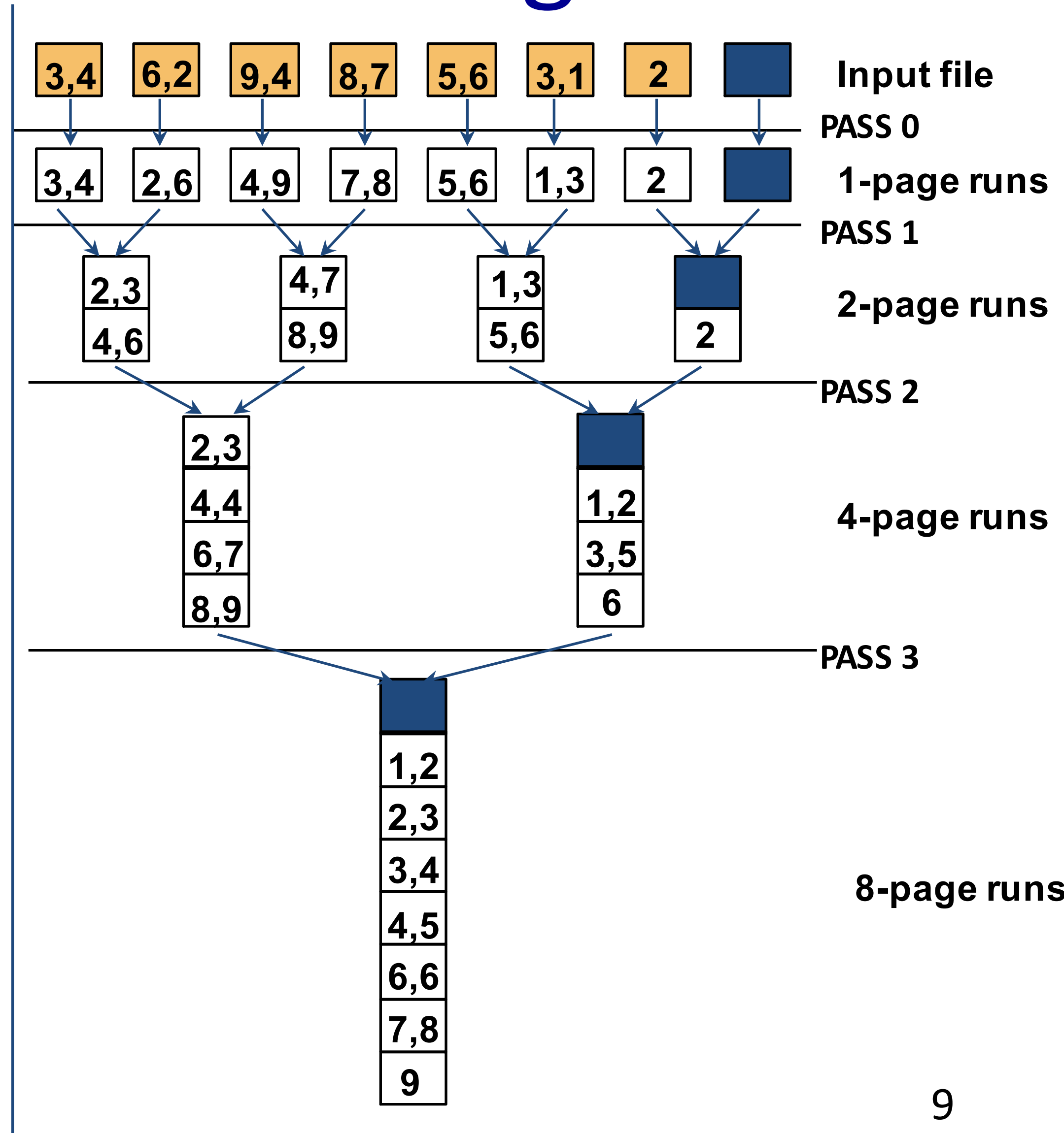


Two-Way External Merge Sort

- Each pass we read + write each page in file.
- N pages in the file
- => the number of passes $= \lceil \log_2 N \rceil + 1$
- So toal cost is:

$$2N\left(\lceil \log_2 N \rceil + 1\right)$$

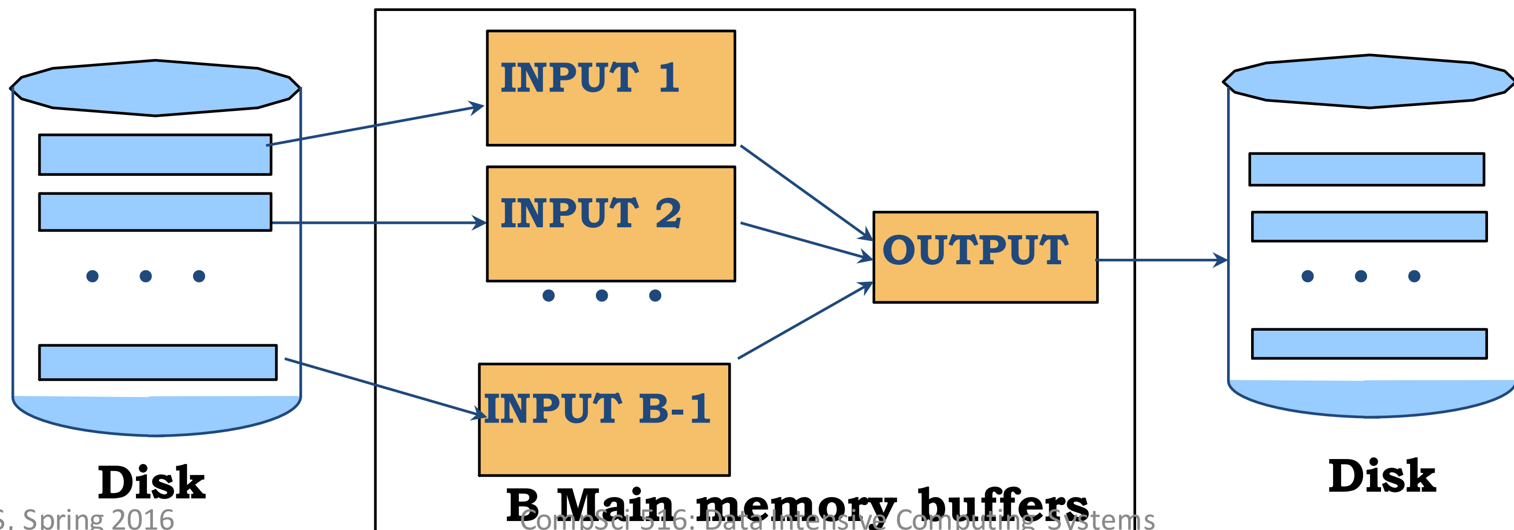
- Idea: Divide and conquer: sort subfiles and merge



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 ⌈N/B⌉ 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 * (# 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

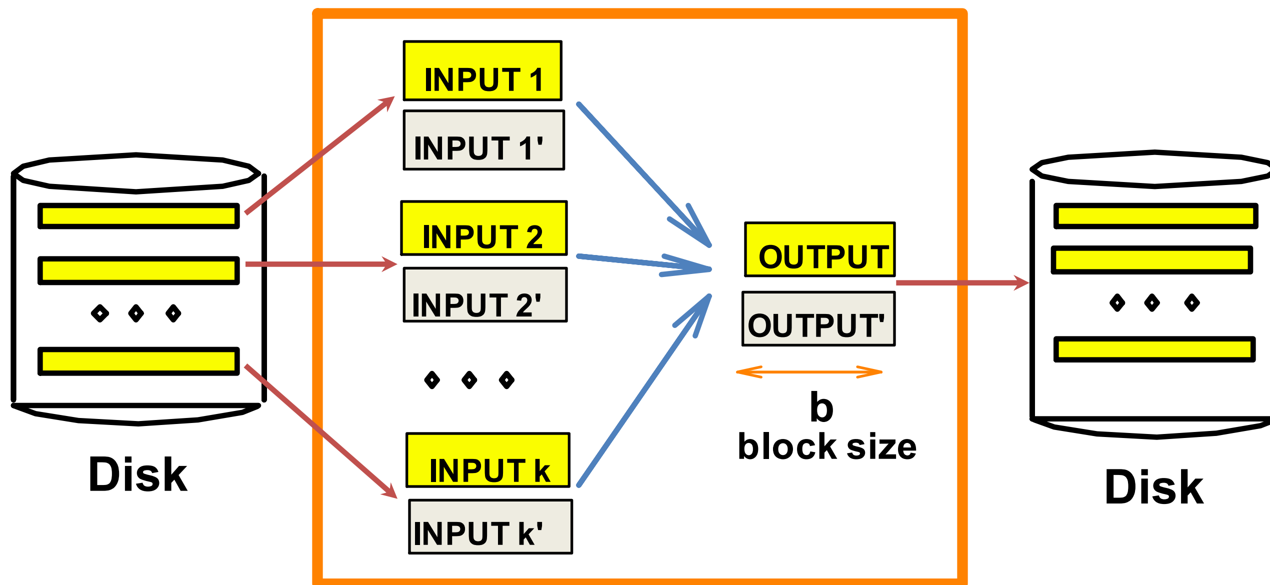
Number of Passes of Optimized Sort

N	B=1,000	B=5,000	B=10,000
100	1	1	1
1,000	1	1	1
10,000	2	2	1
100,000	3	2	2
1,000,000	3	2	2
10,000,000	4	3	3
100,000,000	5	3	3
1,000,000,000	5	4	3

☛ *Block size = 32, initial pass produces runs of size 2B.*

Double Buffering

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



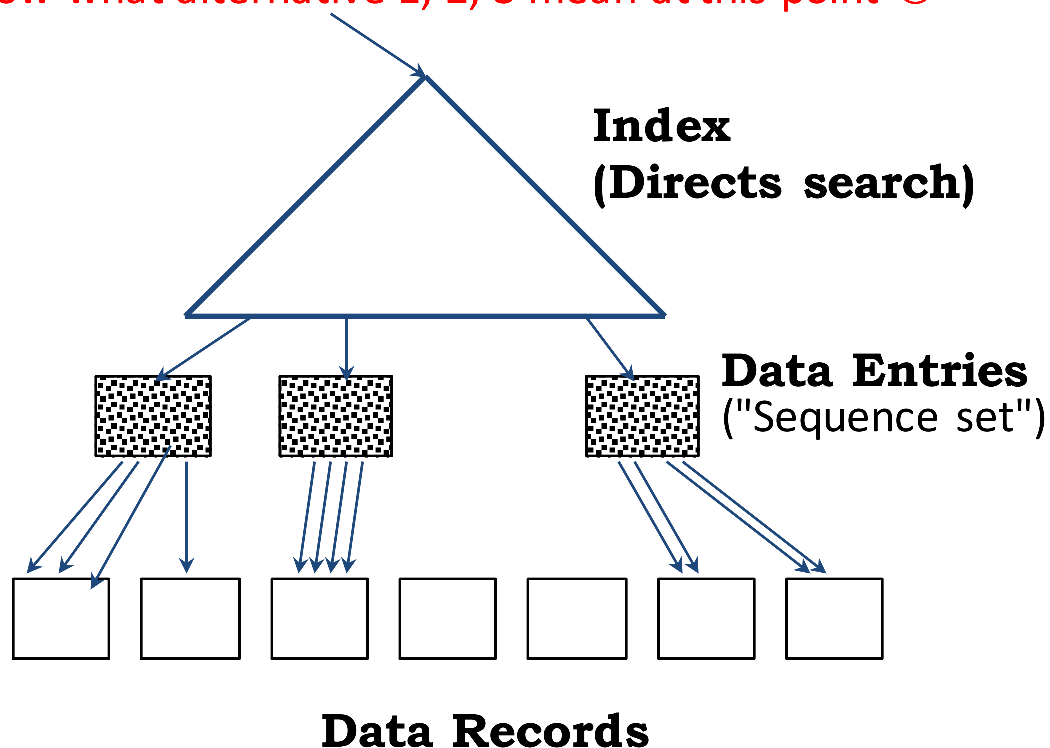
Using B+ Trees for Sorting

- 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

you should know what alternative 1, 2, 3 mean at this point ☺

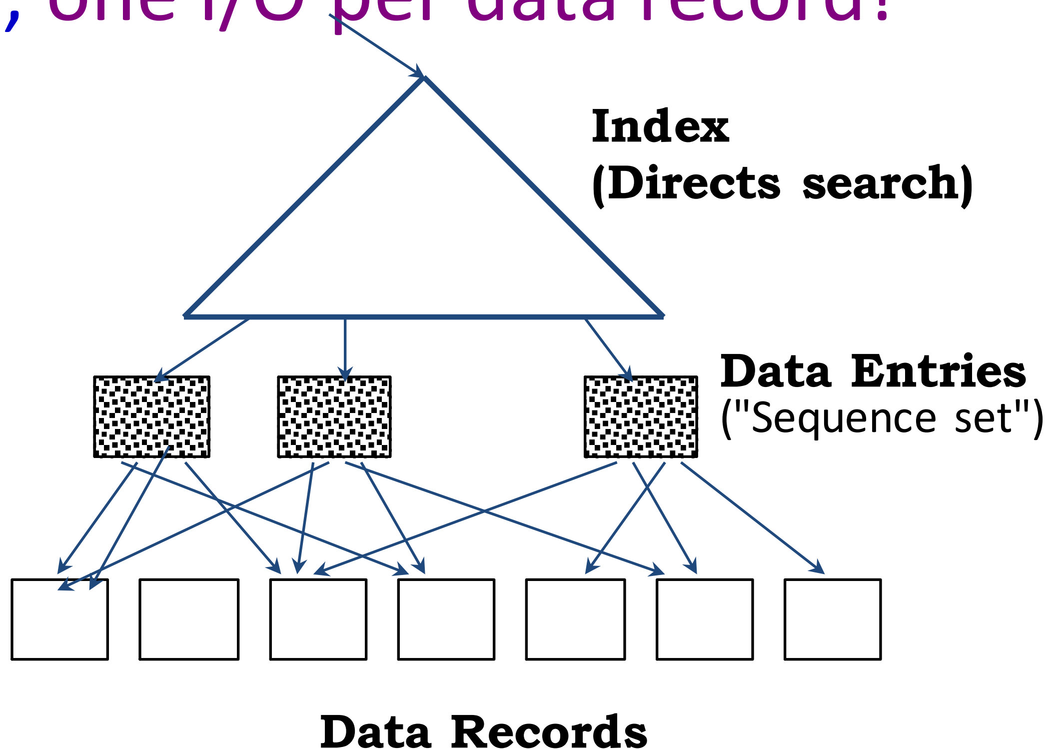
- Cost: root to the left-most leaf, then retrieve all leaf pages (Alternative 1)
- If Alternative 2 is used? Additional cost of retrieving data records: each page fetched just once.



☛ *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!



External Sorting vs. Unclustered Index

N	Sorting	p=1	p=10	p=100
100	200	100	1,000	10,000
1,000	2,000	1,000	10,000	100,000
10,000	40,000	10,000	100,000	1,000,000
100,000	600,000	100,000	1,000,000	10,000,000
1,000,000	8,000,000	1,000,000	10,000,000	100,000,000
10,000,000	80,000,000	10,000,000	100,000,000	1,000,000,000

Unclustered: pN
Clustered: N

- ☛*N* = *#data pages*
- ☛*p*: **# of records per page**
- ☛ ***B=1,000 and block size=32 for sorting***
- ☛ ***p=100 is the more realistic value.***

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

Operator Algorithms

Review from last lecture

- Recall Relational Algebra = Logical Query Plan
- Now Algorithms will be attached to each operator = Physical Query Plan
- An access path is a method of retrieving tuples:
  1. File scan, or
  2. index that matches a selection (in the query)
- Selectivity of an access path:
  - the number of pages retrieved for an access path
  - includes data pages + index pages
  - More selective: fewer pages

Relational Operations

- We will consider how to implement:
  - Selection (σ) Selects a subset of rows from relation.
  - Projection (π) Deletes unwanted columns from relation.
  - Join (⋈) Allows us to combine two relations.
  - Set-difference (-) Tuples in reln. 1, but not in reln. 2.
  - Union (∪) Tuples in reln. 1 and in reln. 2.
  - Aggregation (SUM, MIN, etc.) and GROUP BY
- Since each op returns a relation, ops can be composed
- After we cover each operation, we will discuss how to optimize queries formed by composing them (query optimization).

Schema for Examples

Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

- Similar to old schema; *rname* added for variations.
- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.

Algorithms for Selection

Selection: 1
No Index, Unsorted Data

```
SELECT   *
FROM     Reserves R
WHERE    R.rname = 'Joe'
```

Reserves has
- 1000 pages
- each page holds 100 tuples
- each tuple is 40 bytes

Naïve approach
- Scan the entire relation
- Check the condition and build answer set

- Cost = 1000 I/O
- If only a few tuples with 'Joe'
  - expensive, does not use selection

Selection: 2
No Index, Sorted Data

```
SELECT   *
FROM     Reserves R
WHERE    R.rname = 'Joe'
```

Reserves has
- 1000 pages
- each page holds 100 tuples
- each tuple is 40 bytes

- Here, sorted on “rname”
- Locate the first tuple that satisfies the condition
- scan the relation until the condition is no longer satisfied

- Cost of binary search = $\log_2 1000$ = 10 (approx)
- Cost of scan will depend on #satisfying tuples
  - can range from 0 to 1000 (=#pages)

Selection: 3
B+ tree Index

```
SELECT  *
FROM    Reserves R
WHERE   R.rname = 'Joe'
```

Reserves has
- 1000 pages
- each page holds 100 tuples
- each tuple is 40 bytes

- Search the tree to find the first index entry pointing to a qualifying tuple
- Scan the leaves to find all data entries
- Then retrieve the tuples

- Cost of identifying the starting leaf page:
  - typically 2 or 3 I/O
- Cost of scanning leaves will depend on #such data entries
- Cost of retrieving tuples will depend on (if not alternative 1)
  - #qualifying tuples
  - whether the index is clustered (probably just one I/O if all tuples fit in a page)
  - or unclustered (could be one I/O per qualifying tuple)

Selection: 4
Hash Index, Equality

```
SELECT  *
FROM    Reserves R
WHERE   R.rname = 'Joe'
```

Reserves has
- 1000 pages
- each page holds 100 tuples
- each tuple is 40 bytes

- Retrieve the bucket page
- Then retrieve the qualifying tuples

- Cost of retrieving the bucket
  - typically 1 or 2 I/O
- Cost of scanning leaves will depend on #such data entries
- Cost of retrieving tuples will depend on (if not alternative 1)
  - #qualifying tuples
  - whether the index is clustered (probably just one I/O if all tuples fit in a page) – if index on a key, just one tuple and one page
  - or unclustered (could be one I/O per qualifying tuple)

Refinement for Unclustered Index for Selections

1. Find qualifying data entries.
2. Sort the rid's of the data records to be retrieved.
3. Fetch rids in order.

– This ensures that each data page is looked at just once

– however, no. of such pages likely to be higher than with clustering

General Selection

- What if we have more complex selection conditions?
  - instead of attr <op> value
  - we could have logical AND (∧) and OR (∨)

- Two main approaches

Approach 1: Filtering

- Find the most selective access path, retrieve tuples using it, and apply any remaining terms that don't match the index:

- Consider day<8/9/94 AND bid=5 AND sid=3
- A B+ tree index on day can be used
  - then, bid=5 and sid=3 must be checked for each retrieved tuple
- A hash index on <bid, sid> could be used;
  - day<8/9/94 must then be checked.

Approach 2: Intersection

- If we have 2 or more matching indexes that use Alternatives (2) or (3) for data entries):
  - Get sets of rids of data records using each matching index.
  - Then intersect these sets of rids
    - we'll discuss intersection later
  - Retrieve the records and apply any remaining terms.
  - Consider day<8/9/94 AND bid=5 AND sid=3
    - If we have a B+ tree index on day and an index on sid, both using Alternative (2)
    - we can retrieve rids of records satisfying day<8/9/94 using the first
    - rids of recs satisfying sid=3 using the second
    - intersect
    - retrieve records and check bid=5.

Handling Disjunctions in Practice

1. convert the query into a union of queries without OR
2. if same attribute, $A < 5 \vee A > 10$, use a nested query with an IN and an index
3. simply apply the disjunction condition on the retrieved tuples
4. use bitmap
   – see [RG] 14.2.3.

- Most DBMSs do not handle disjunctions too efficiently, we won't discuss them in detail

Algorithms for Projection

Projection

- Two parts
  - Remove some fields (easy)
  - Remove duplicates (hard)

```
SELECT   DISTINCT
         R.sid, R.bid
FROM     Reserves R
```

- The expensive part is removing duplicates.
  - SQL systems don't remove duplicates unless the keyword DISTINCT is specified in a query
  - Then just scan the table or use index (if the key contains all the necessary fields)
  - Otherwise, need to delete duplicates

Projection: 1 Sorting-based

```
SELECT   DISTINCT R.sid, R.bid
FROM     Reserves R
```

Reserves has
- 1000 pages
- each page holds 100 tuples
- each tuple is 40 bytes

- Scan R and eliminate unwanted tuples
- Sort this set with all attributes as the key for sorting
- Scan the sorted result, compare adjacent tuples, discard duplicates

- Improvement:
  - project out unwanted attribute in the first pass of external sorting
  - Eliminate duplicates during merging

Projection: 2A Hashing-based

SELECT DISTINCT R.sid, R.bid
FROM Reserves R

Reserves has
- 1000 pages
- each page holds 100 tuples
- each tuple is 40 bytes

Assume B buffers are available

Step A: Partitioning phase

- Read R using one input buffer
- For each tuple, discard unwanted fields, apply hash function h1 to choose one of B-1 output buffers.
  - Result is B-1 partitions (of tuples with no unwanted fields)
  - Two tuples from different partitions guaranteed to be distinct
- Write each partition back to the disk

- Cost: For partitioning, read R, write out each tuple, but with fewer fields. This is read in next phase.

Projection: 2B Hashing-based

```
SELECT DISTINCT R.sid, R.bid
FROM   Reserves R
```

Reserves has
- 1000 pages
- each page holds 100 tuples
- each tuple is 40 bytes

Assume B buffers are available

Step B: Duplicate elimination phase

For each partition

- Build an in-memory hash table
- Read it one page at a time into memory
- Hash using function h2 (different from h1) on all fields
  - For two tuples in the same bucket, check for duplicates, then discard duplicates.
- Why does h2 have to be different from h1?
  - since h1 hashes the same partition to the same value
- If partition does not fit in memory, can apply hash-based projection algorithm recursively to this partition.

Discussion of Projection

- Sort-based approach is the standard
  - better handling of skew (many duplicates) – hash table may not fit in memory
  - result is sorted
  - external sorting is provided in most DBMS as a utility
- If an index on the relation contains all wanted attributes in its search key, can do index-only scan.
  - Apply projection techniques to data entries (much smaller!)
- If an ordered (i.e., tree) index contains all wanted attributes as prefix of search key, can do even better:
  - Retrieve data entries in order (index-only scan), discard unwanted fields, compare adjacent tuples to check for duplicates.

Next lecture – Join Algorithms