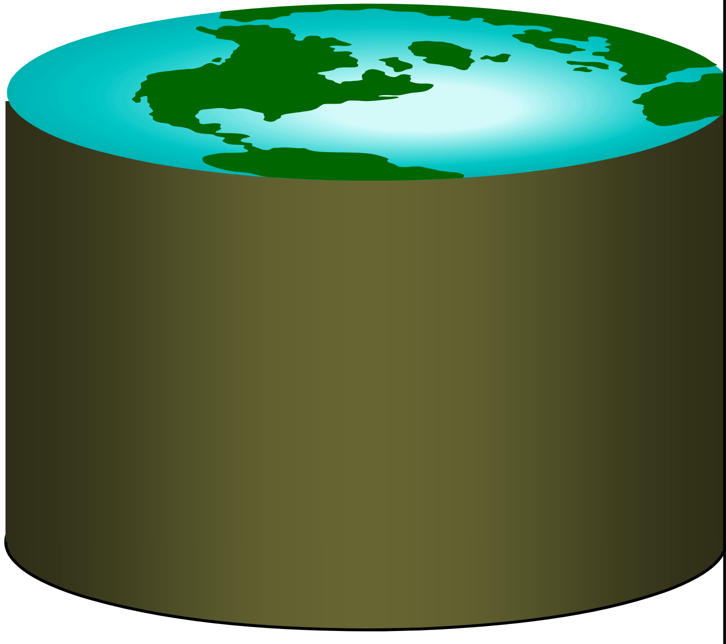
Database Management System

Sumayyea Salahuddin (Lecturer)
Dept. of Computer Systems Eng.
UET Peshawar



Objectives

- **File Organization & Indexing**
  - Indexes
  - B+ Tree Indexes
  - Comparison
- **Query Processing: Sorting & Joins**
- **Query Optimization**

File Organizations and Indexing

If you don't find it in the index, look very carefully through the entire catalogue.

-- Sears, Roebuck, and Co., (Consumer's Guide, 1897)

Buffer Management and Files

- **Storage of Data**
  - Fields, either fixed or variable length...
  - Stored in Records...
  - Stored in Pages...
  - Stored in Files

- **If data won't fit in RAM, store on Disk**
  - Need Buffer Pool to hold pages in RAM
  - Different strategies decide what to keep in pool

File Organization

- **How to keep pages of records on disk**
- ***but* must support operations:**
  - scan all records
  - search for a record id “RID”
  - search for record(s) with certain values
  - insert new records
  - delete old records

Alternative File Organizations

Many alternatives exist, tradeoffs for *each:*

- Heap files:
  - Suitable when typical access is file scan of all records.
- Sorted Files:

  –In this method, As the name itself suggest whenever a new record has to be inserted, it is always inserted in a sorted (ascending or descending) manner. Sorting of records may be based on any primary key or any other key. Suppose a new record R2 has to be inserted in the sequence, then it will be inserted at the end of the file and then it will sort the sequence .

  - Best for retrieval in *search key* order
  - Also good for search based on *search key*
- Indexes: Organize records via trees or hashing.
  - Like sorted files, speed up searches for *search key* fields
  - Updates are much faster than in sorted files.

Indexes

- **Sometimes need to retrieve records by the *values in one or more fields*, e.g.,**
  - Find all students in the "CS" department
  - Find all students with a gpa > 3
- **An *index* on a file is a:**
  - Disk-based data structure
  - Speeds up selections on the *search key fields* for the index.
  - Any subset of the fields of a relation can be index search key
  - *Search key* is not the same as *key*
    - (e.g. doesn't have to be unique ID).
- **An index**
  - Contains a collection of *data entries*
  - Supports efficient retrieval of all records with a given search key value **k**.

First Question to Ask About Indexes

- **What kinds of selections do they support?**
  - Selections of form field <op> constant
  - Equality selections (op is =)
  - Range selections (op is one of <, >, <=, >=, BETWEEN)
  - More exotic selections:
    - 2-dimensional ranges ("east of Berkeley and west of Truckee and North of Fresno and South of Eureka")
      - Or n-dimensional
    - 2-dimensional distances ("within 2 miles of Soda Hall")
      - Or n-dimensional
    - Ranking queries ("10 restaurants closest to VLSB")
    - Regular expression matches, genome string matches, etc.
    - One common n-dimensional index: R-tree
      - Supported in Oracle and Informix
      - See http://gist.cs.berkeley.edu for research on this topic

Index Classification

- **What selections does it support**

- **Representation of data entries in index**
  - what info is the index storing? 3 alternatives:
    - Data record with key value **k**
    - <**k**, rid of data record with search key value **k**>
    - <**k**, list of rids of data records with search key **k**>

- **Clustered vs. Unclustered Indexes**
- **Single Key vs. Composite Indexes**
- **Tree-based, hash-based, other**

Alternatives for Data Entry **k*** in Index

- **Three alternatives:**
  - Actual data record (with key value **k**)
  - <**k**, rid of matching data record>
  - <**k**, list of rids of matching data records>

- **Choice is orthogonal to the indexing technique.**
  - techniques: B+ trees, hash-tables, R trees, …
  - Typically, index contains auxiliary information that directs searches to the desired data entries

- **Can have multiple (different) indexes per file.**
  - E.g. file sorted by *age*, with a hash index on *salary* and a B+tree index on *name*.

Alternatives for Data Entries (Contd.)

- **Alternative 1:**
  Actual data record (with key value k**)**
  - Index structure *is* file organization for data records (like Heap files or sorted files).
  - At most one index on a table can use Alternative 1.
  - Saves pointer lookups
  - Can be expensive to maintain with insertions and deletions.

Alternatives for Data Entries (Contd.)

Alternative 2

<k, rid of matching data record>

and Alternative 3

<k, list of rids of matching data records>

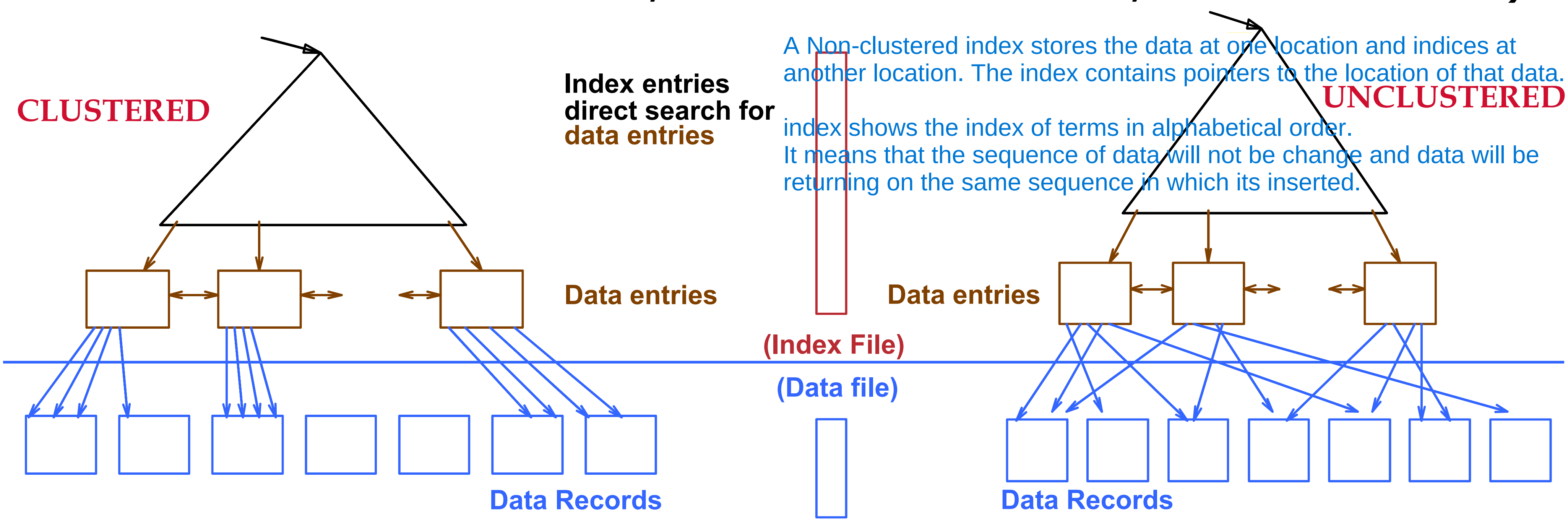
- Easier to maintain than Alt 1.
- At most one index can use Alternative 1; any others must use Alternatives 2 or 3.
- Alternative 3 more compact than Alternative 2, but leads to *variable sized data* entries even if search keys are of fixed length.
- Even worse, for large rid lists the data entry might have to span multiple pages!

Index Classification

- ***Clustered* vs. *unclustered*:**
  - If order of data records is the same as, or `close to', order of index data entries, then called *clustered index*.
- **A file can be clustered on at most one search key.**
- **Cost to retrieve data records with index varies *greatly* based on whether index clustered or not!**
- **Alternative 1 implies clustered, *but not vice-versa*.**

Clustered vs. Unclustered Index

- **Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file.**
  - To build clustered index, first sort the Heap file (with some free space on each block for future inserts).
  - Overflow blocks may be needed for inserts. (Thus, order of data recs is `close to', but not identical to, the sort order.)



Unclustered vs. Clustered Indexes

- **What are the tradeoffs????**
- **Clustered Pros**
  - Efficient for range searches
  - May be able to do some types of compression
  - Possible locality benefits (related data?)

- **Clustered Cons**
  - Expensive to maintain (on the fly or sloppy with reorganization)

Hash-Based Indexes

- **Good for equality selections.**
  - Index is a collection of *buckets.* Bucket = *primary* page plus zero or more *overflow* pages.
  - *Hashing function* **h**:

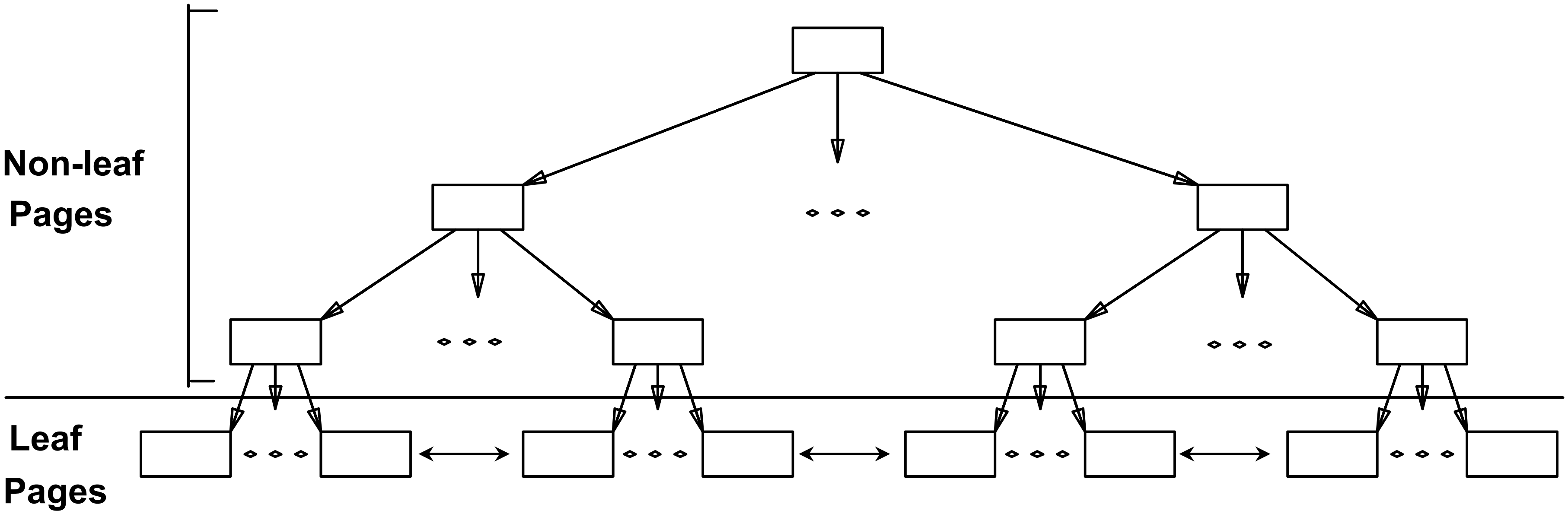
    cost is number of pages in bucket

    – **h**(*r*) = bucket in which record *r* belongs.

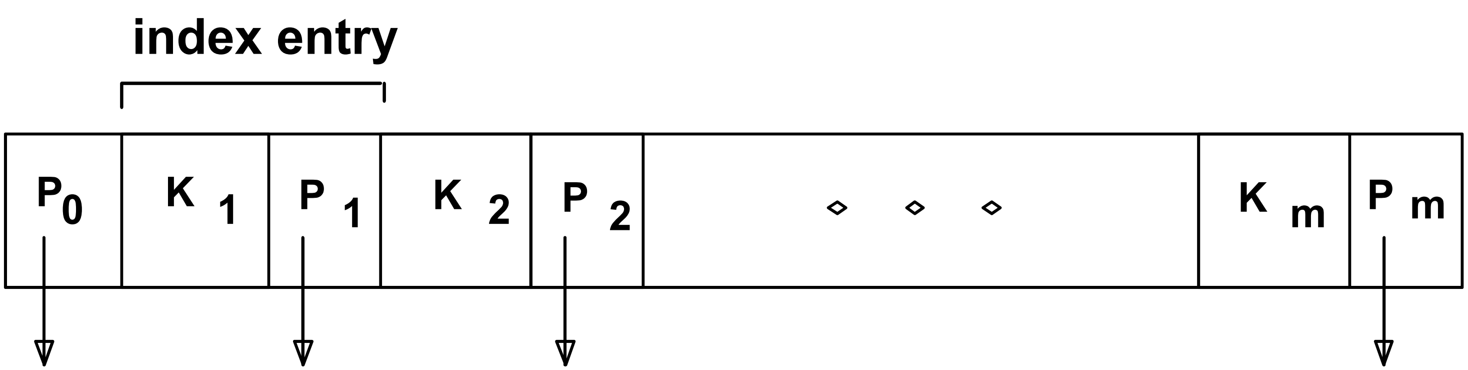
    – **h** looks at the *search key* fields of *r.*

- **If Alternative (1) is used, the buckets contain the data records; otherwise, they contain <key, rid> or <key, rid-list> pairs.**

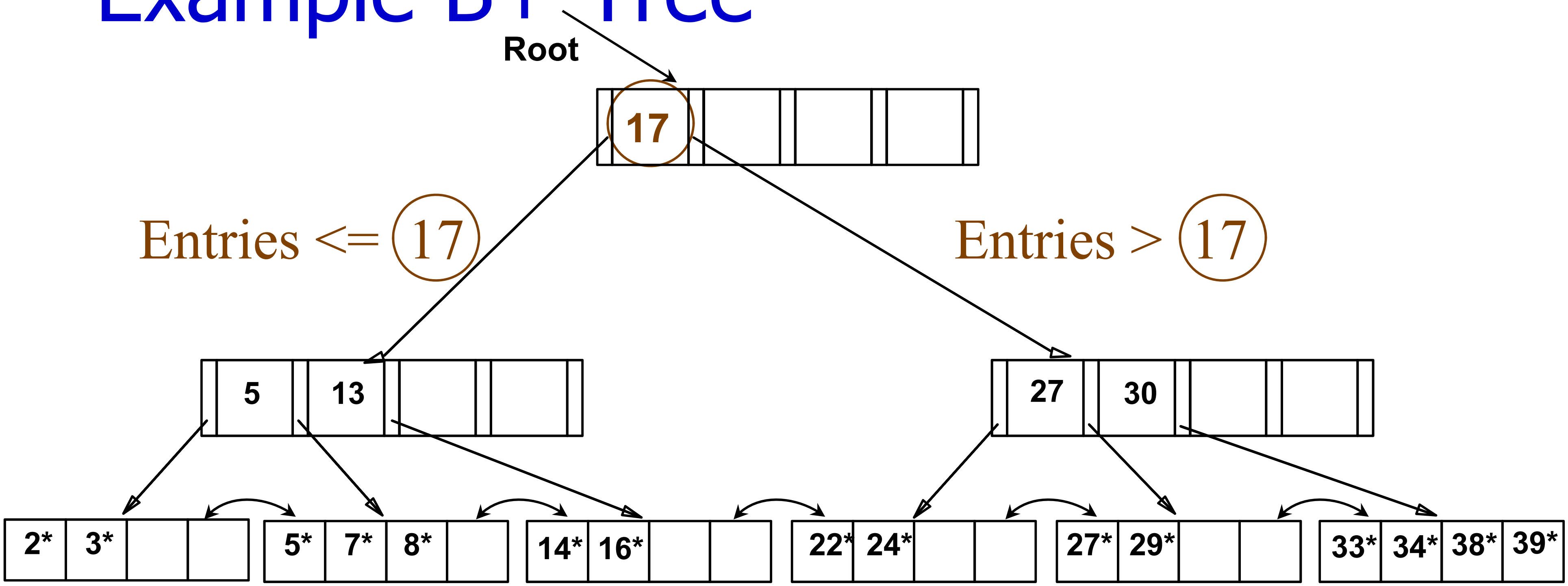
B+ Tree Indexes



- Leaf pages contain *data entries*, and are chained (prev & next)
- Non-leaf pages contain *index entries* and direct searches:



Example B+ Tree



- **Find 28*? 29*? All > 15* and < 30***
- **Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes.**
  - And change sometimes bubbles up the tree

Comparing File Organizations

- **Heap files (random order; insert at eof)**
- **Sorted files, sorted on *<age, sal>***
- **Clustered B+ tree file, Alternative (1), search key *<age, sal>***
- **Heap file with unclustered B + tree index on search key *<age, sal>***
- **Heap file with unclustered hash index on search key *<age, sal>***

Operations to Compare

- **Scan: Fetch all records from disk**
- **Fetch all records in sorted order**
- **Equality search**
- **Range selection**
- **Insert a record**
- **Delete a record**

Cost Model for Analysis

I/O cost 150,000 times more than hash function

- We ignore CPU costs, for simplicity

B: **The number of data pages**

R: **Number of records per page**

F: **Fanout of B-tree**

Average-case analysis; based on several simplistic assumptions.

□ *Good enough to show the overall trends!*

Assumptions in Our Analysis

- **Heap Files:**
  - Equality selection on key; exactly one match.
- **Sorted Files:**
  - Files compacted after deletions.
- **Indexes:**
  - Alt (2), (3): data entry size = 10% size of record
  - Hash: No overflow buckets.
    - 80% page occupancy => File size = 1.25 data size
  - Tree: 67% occupancy (this is typical).
    - Implies file size = 1.5 data size

I/O Cost of Operations

B: Number of data pages (packed)
R: Number of records per page
S: Time required for equality search

	Heap File
Scan all records	B
Get all in sort order	4B
Equality Search	0.5 B
Range Search	B
Insert	2
Delete	0.5B + 1

I/O Cost of Operations

B: Number of data pages (packed)
R: Number of records per page
S: Time required for equality search

	Sorted File
Scan all records	**B**
Get all in sort order	**B**
Equality Search	**$\log_2 B$**
Range Search	**S + # matching pages**
Insert	**S + B**
Delete	**S + B**

I/O Cost of Operations

B: Number of data pages (packed)
R: Number of records per page
F: Fanout of B-Tree
S: Time required for equality search

	Clustered Tree
Scan all records	**1.5 B**
Get all in sort order	**1.5 B**
Equality Search	**$\log_F$ (1.5 B)**
Range Search	**S + #matching pages**
Insert	**S + 1**
Delete	**0.5B + 1**

I/O Cost of Operations

B: Number of data pages (packed)
R: Number of records per page
F: Fanout of B-Tree
S: Time required for equality search

	Unclustered Tree
Scan all records	1.5 B
Get all in sort order	4B
Equality Search	$\log_F (1.5\ B) + 1$
Range Search	S + #matching records
Insert	S + 2
Delete	S + 2

I/O Cost of Operations

B: Number of data pages (packed)
R: Number of records per page
S: Time required for equality search

	Hash Index
Scan all records	1.25 B
Get all in sort order	4B
Equality Search	2
Range Search	1.25 B
Insert	4
Delete	S + 2

I/O Cost of Operations

B: The number of data pages
R: Number of records per page
F: Fanout of B-Tree
S: Time required for equality search

	Heap File	Sorted File	Clustered Tree	Unclustered Tree	Hash Index
Scan all records	B	B	1.5 B	1.5 B	1.25 B
Get all in sort order	4B	B	1.5 B	4B	4B
Equality Search	0.5 B	$\log_2$ B	$\log_F$ (1.5 B)	$\log_F$ (1.5 B) + 1	2
Range Search	B	S + #matching pages	S + #matching pages	S + #matching records	1.25 B
Insert	2	S + B	S + 1	S + 2	4
Delete	0.5B + 1	S + B	0.5B + 1	S + 2	S + 2

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.

- **Multi-attribute search keys should be considered when a WHERE clause contains several conditions.**
  - Order of attributes is important for range queries.
  - Such indexes sometimes enable index-only strategies
    - For index-only strategies, clustering is not important!

- **Choose indexes that benefit as many queries as possible.**
- **Since only one index can be clustered per table, choose it based on important queries that would 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?
  - Is the index clustered?
- **Consider the GROUP BY query.**
  - If many tuples have *E.age* > 10, using *E.age* index and sorting the retrieved tuples may be costly.
  - Clustered *E.dno* index may be better!
- **Equality queries and duplicates:**
  - Clustering on *E.hobby* helps!

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

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

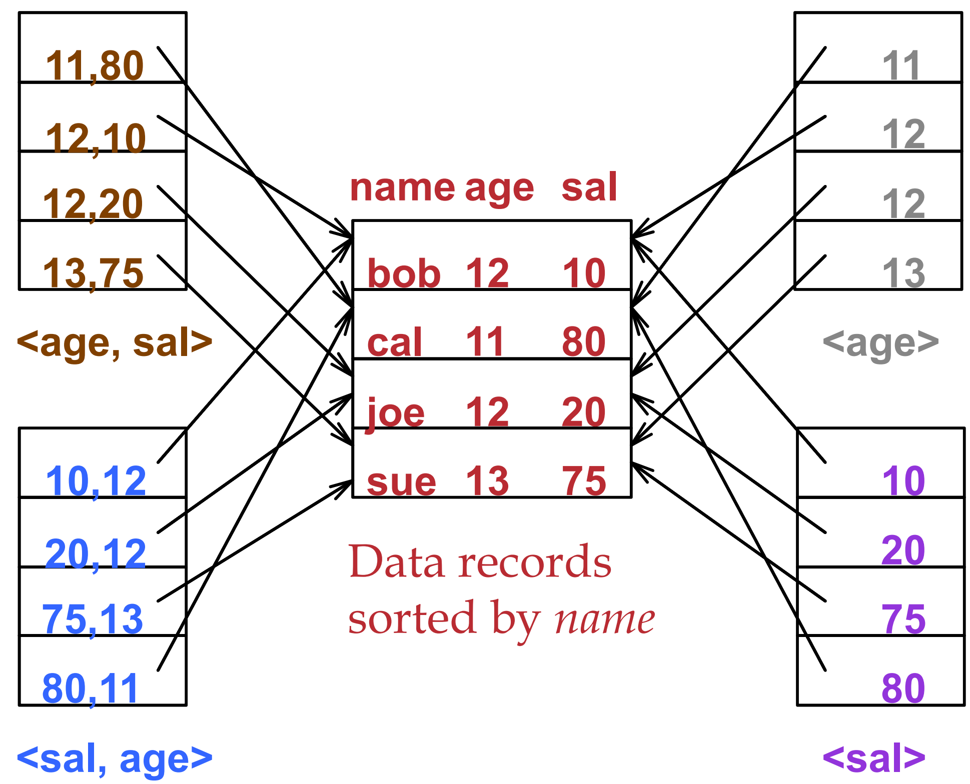
```
SELECT  E.dno
FROM  Emp E
WHERE  E.hobby=Stamps
```

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 not a constant. E.g.:
    - age =20; or age=20 and sal > 10

- **Data entries in index sorted by search key to support range queries.**
  - Lexicographic order, or
  - Spatial order.

Examples of composite key indexes using lexicographic order.



Data entries in index sorted by <*sal,age*>

Data entries sorted by <*sal*>

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 an index on *sal*.**
  - 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!
- **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.

<E.dno>

```
SELECT D.mgr
FROM Dept D, Emp E
WHERE D.dno=E.dno
```

<E.dno,E.eid>
Tree index!

```
SELECT D.mgr, E.eid
FROM Dept D, Emp E
WHERE D.dno=E.dno
```

<E.dno>

```
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!

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

Index-Only Plans (Contd.)

- **Index-only plans are possible if the key is <dno,age> or we have a tree index with key <age,dno>**
  - 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
```

Summary

- **Alternative file organizations, tradeoffs for each**
- **If selection queries are frequent, sorting the file or building an *index* is important.**
  - Hash-based indexes only good for equality search.
  - 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 (Contd.)

- **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**
  - clustered, unclustered
  - B-tree, hash table, etc.

Summary (Contd.)

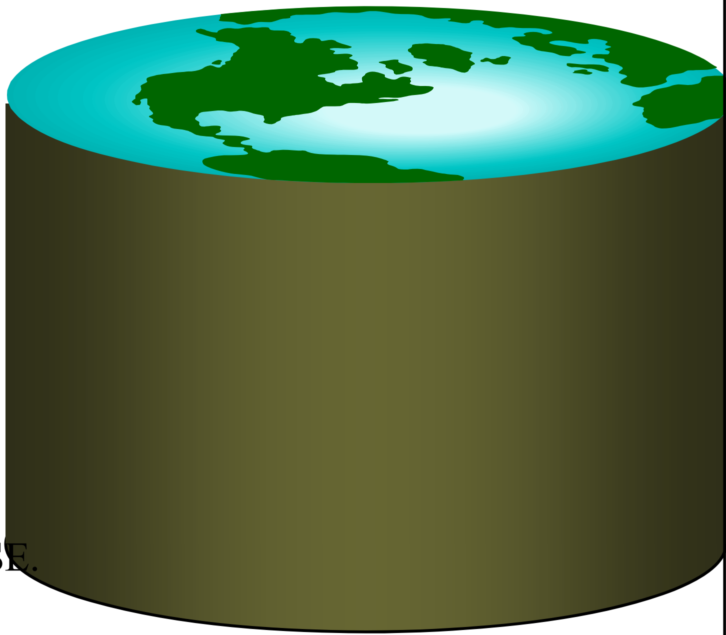
- **Understanding the nature of the *workload* for the application, and the performance goals, is essential to developing a good design.**
  - What are the important queries and updates? What attributes/relations are involved?

- **Indexes must be chosen to speed up important queries (and perhaps some updates!).**
  - Index maintenance overhead on updates to key fields.
  - Choose indexes that can help many queries, if possible.
  - Build indexes to support index-only strategies.
  - Clustering is an important decision; only one index on a given relation can be clustered!
  - Order of fields in composite index key can be important.

Query Processing: Joins & Sorting

One of the advantages of being disorderly is that one is constantly making exciting discoveries.

-- A. A. Milne



Questions

- **We learned that the same query can be written many ways.**
  - How does DBMS decide which is best?
- **We learned about tree & hash indexes.**
  - How does DBMS know when to use them?
- **Sometimes we need to sort data.**
  - How to sort more data than will fit in memory?

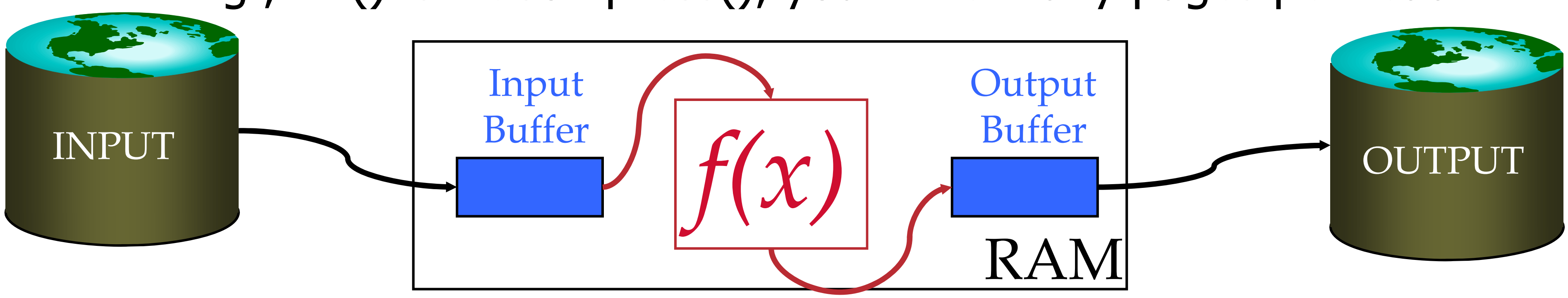
Why Sort?

- **A classic problem in computer science!**
- **Database needs it in order**
  - e.g., find students in increasing *gpa* order
  - first step in *bulk loading* B+ tree index.
  - eliminating *duplicates*
  - aggregating related groups of tuples
  - *Sort-merge* join algorithm involves sorting.
- **Problem: sort 1Gb of data with 1Mb of RAM.**
  - why not virtual memory?

Heap sort keeps the data permanently in a partially sorted condition and so is comparable to Insertion sort. But it is substantially quicker and has a worst case of O(n log n) compared with O(n2) for Insertion Sort.

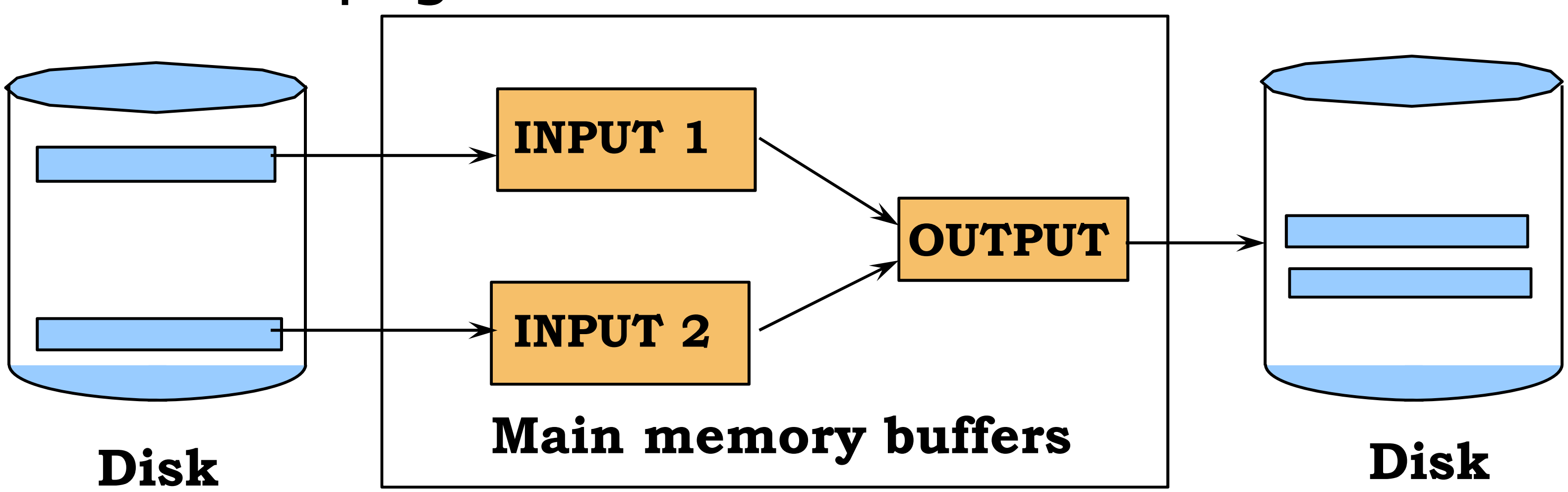
Streaming Data Through RAM

- **An important detail for sorting & other DB operations**
- **Simple case:**
  - Compute *f(x)* for each record, write out the result
  - Read a page from INPUT to Input Buffer
  - Write *f(x)* for each item to Output Buffer
  - When Input Buffer is consumed, read another page
  - When Output Buffer fills, write it to OUTPUT
- **Reads and Writes are *not* coordinated**
  - E.g., if f() is Compress(), you read many pages per write.
  - E.g., if f() is DeCompress(), you write many pages per read.



2-Way Sort

- **Pass 0: Read a page, sort it, write it.**
  - only one buffer page is used (as in previous slide)
- **Pass 1, 2, …, etc.:**
  - requires 3 buffer pages
  - merge pairs of runs into runs twice as long
  - three buffer pages used.



also called Concatenate sort.

Two-Way External Merge Sort

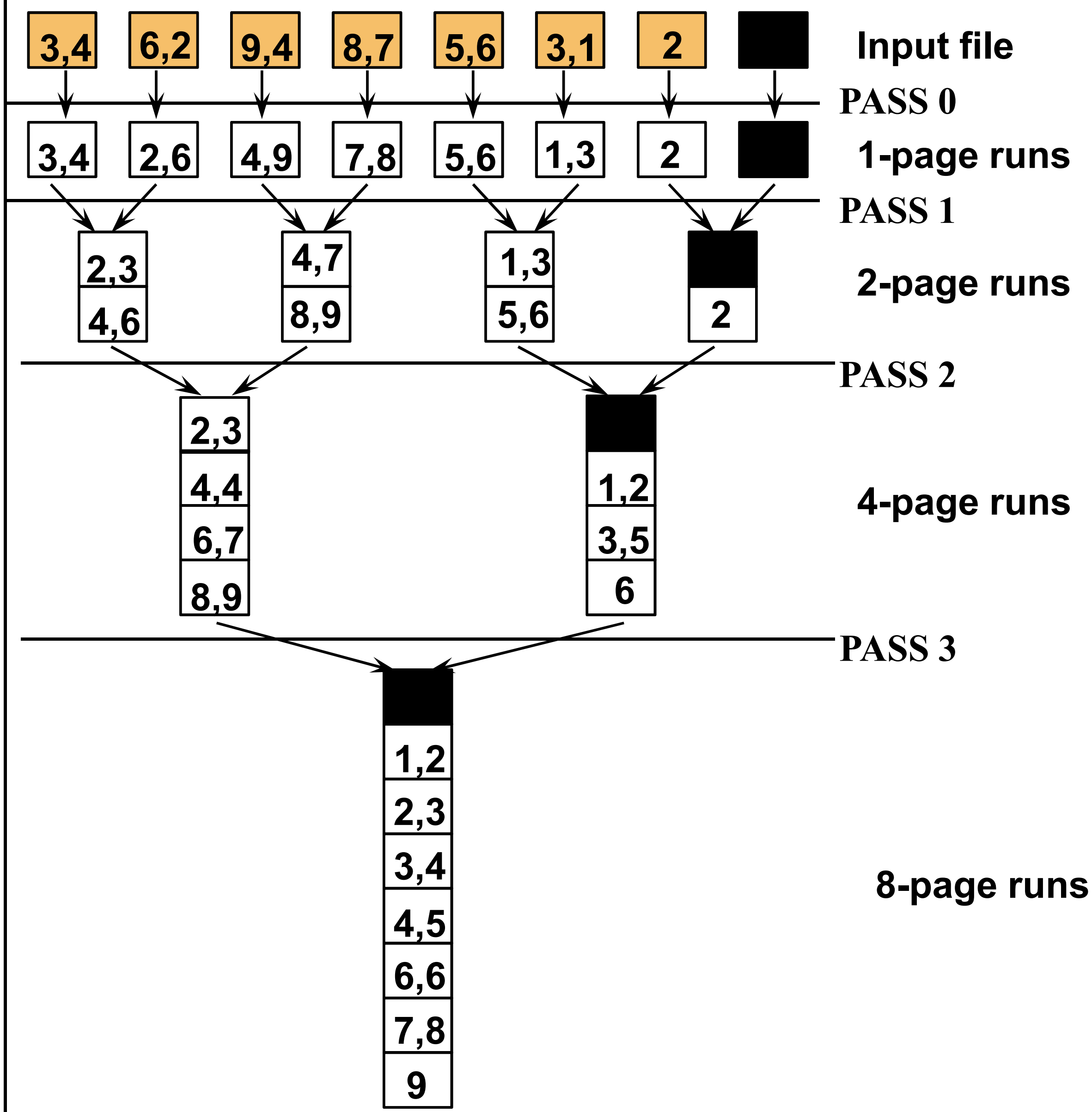
- **Each pass we read + write each page in file.**
- **N pages in the file => the number of passes**

$$= \lceil \log_2 N \rceil + 1$$

- **So total cost is:**

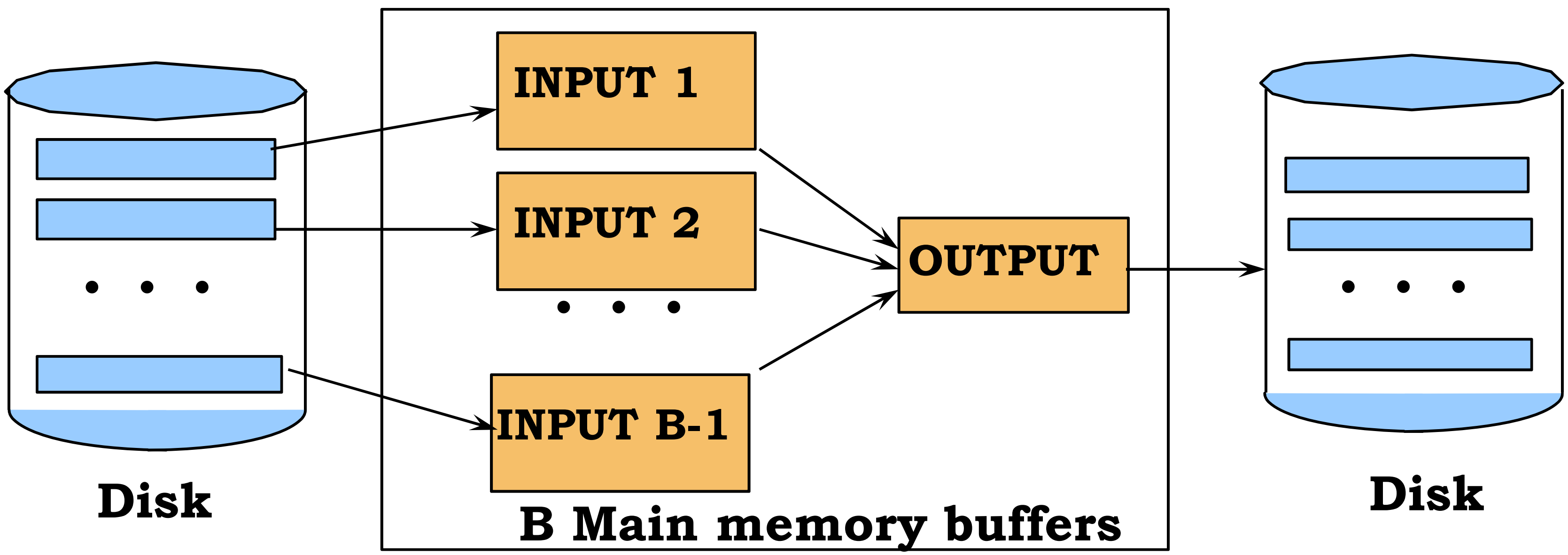
$$2N\left(\lceil \log_2 N \rceil + 1\right)$$

- ***Idea:*** *Divide and conquer:* **sort subfiles and merge**



General External Merge Sort

- *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 $\lceil N / B \rceil$ sorted runs of *B* pages each.
  - Pass 1, 2, …, etc.: merge *B-1* runs.



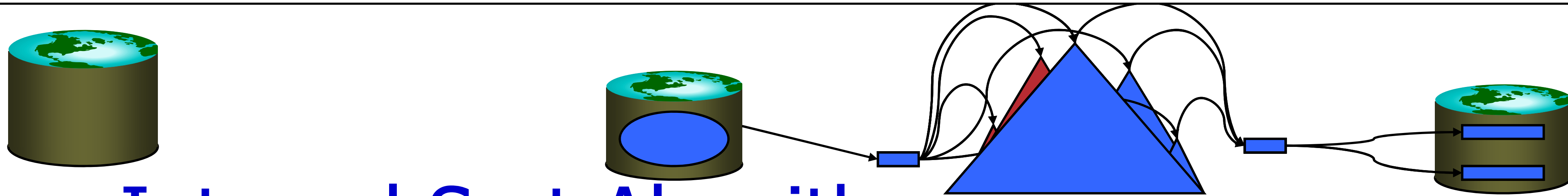
Cost of External Merge Sort

- **Number of passes:** $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$
- **Cost = 2N * (# of passes)**
- **E.g., with 5 buffer pages, to sort 108 page file:**
  – Pass 0: $\lceil 108 / 5 \rceil$ = 22 sorted runs of 5 pages each (last run is only 3 pages)
- **Now, do four-way (B-1) merges**
  – Pass 1: $\lceil 22 / 4 \rceil$ = 6 sorted runs of 20 pages each (last run is only 8 pages)
  – Pass 2: 2 sorted runs, 80 pages and 28 pages
  – Pass 3: Sorted file of 108 pages

Number of Passes of External Sort

(I/O cost is 2N times number of passes)

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



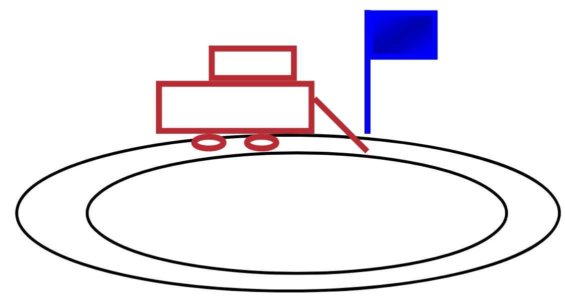
Internal Sort Algorithm

- **Quicksort is a fast way to sort in memory.**
- **Alternative: "tournament sort" (a.k.a. "heapsort", "replacement selection")**
- **Keep two heaps in memory, H1 and H2**

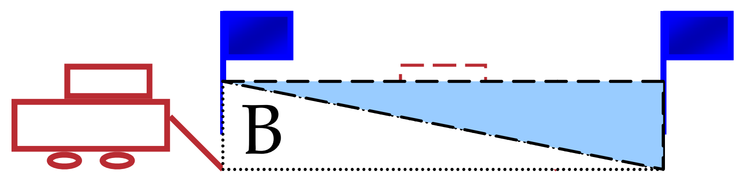
```
read B-2 pages of records, inserting into H1;
while (records left) {
    m = H1.removemin();  put m in output buffer;
    if (H1 is empty)
        H1 = H2;  H2.reset();  start new output run;
    else
        read in a new record r (use 1 buffer for input
          pages);
        if (r < m)  H2.insert(r);
        else        H1.insert(r);
}
H1.output();  start new run;  H2.output();
```

More on Heapsort

- **Fact: average length of a run is *2(B-2)***
  - The "snowplow" analogy



- **Worst-Case:**
  - What is min length of a run?
  - How does this arise?
- **Best-Case:**
  - What is max length of a run?
  - How does this arise?



- **Quicksort is faster, but ... longer runs often means fewer passes!**

I/O for External Merge Sort

- **Actually, doing I/O a page at a time**
  - Not an I/O per record
- **In fact, read a *block (chunk)* of pages sequentially!**
- **Suggests we should make each buffer (input/output) be a *chunk* of pages.**
  - But this will reduce fan-out during merge passes!
  - 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.*

Sorting Records!

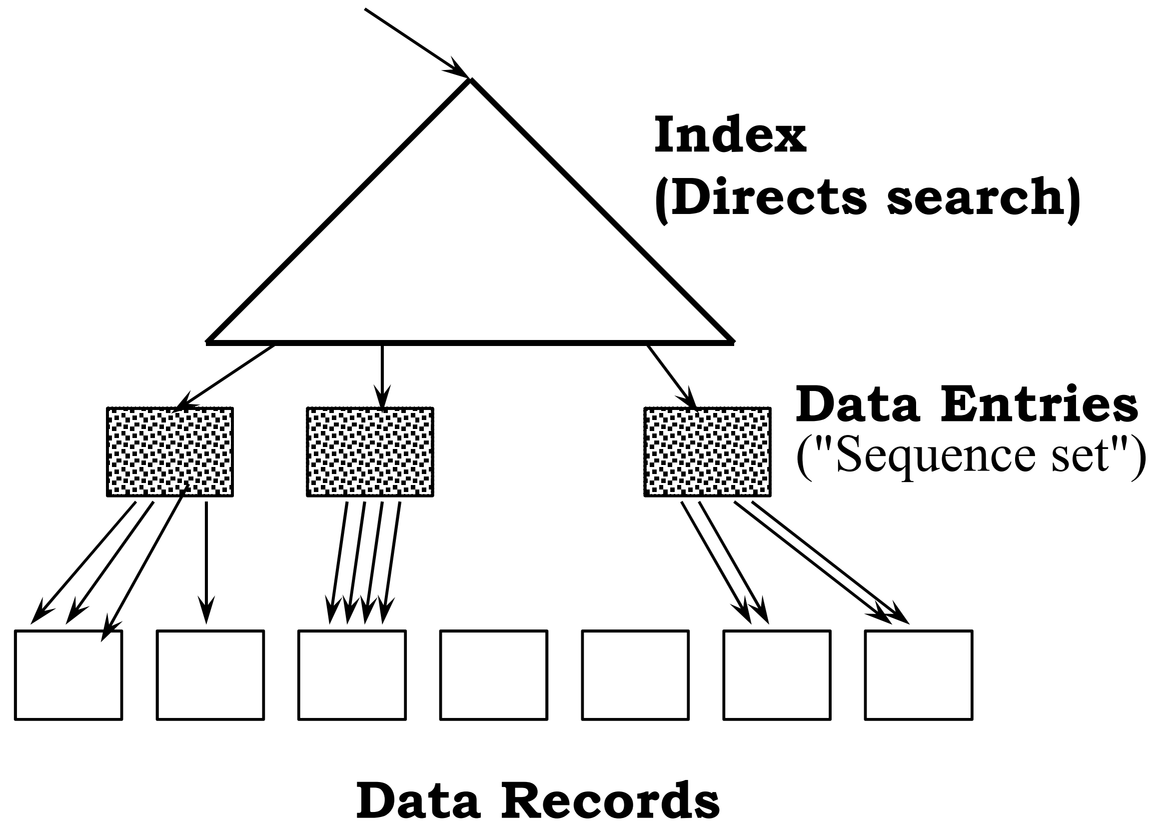
- **Sorting has become a blood sport!**
  - Parallel sorting is the name of the game ...
- **Minute Sort: how many 100-byte records can you sort in a minute?**
  - Typical DBMS: 10MB (~100,000 records)
  - Current World record: 21.8 ***GB***
    - 64 dual-processor Pentium-III PCs (1999)
- **Penny Sort: how many can you sort for a penny?**
  - Current world record: 12GB
    - 1380 seconds on a $672 Linux/Intel system (2001)
    - $672 spread over 3 years = 1404 seconds/penny
- **See http://research.microsoft.com/barc/SortBenchmark/**

Using B+ Trees for Sorting

- **Scenario: Table to be sorted has B+ tree index on sorting column(s).**
- **Idea: Can retrieve 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

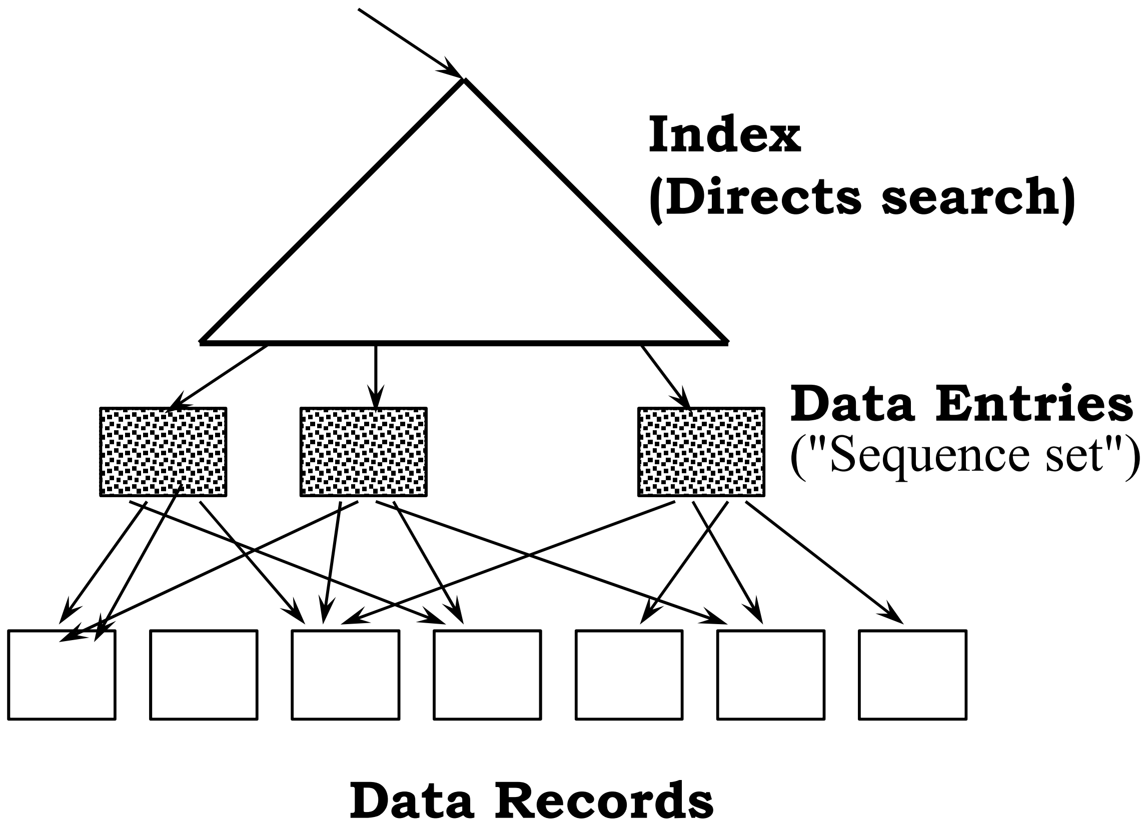
- **Cost: root to the left-most leaf, then retrieve all leaf pages (Alternative 1)**
- **If Alternative 2 is used? Additional cost of retrieving data records: each page fetched just once.**



 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

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

Sorting - Review

- **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 passes rarely more than 2 or 3.

Sorting – Review (cont)

- **Choice of internal sort algorithm may matter:**
  - Quicksort: Quick!
  - Heap/tournament sort: slower (2x), longer runs
- **The best sorts are wildly fast:**
  - Despite 40+ years of research, we're still improving!
- **Clustered B+ tree is good for sorting; unclustered tree is usually very bad.**

A Related Topic: Joins

- **How does DBMS join two tables?**
- **Sorting is one way...**
- **Database must choose best way for each query**

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 total.
- **Sailors:**
  - Each tuple is 50 bytes long,
  - 80 tuples per page,
  - 500 pages total.

Equality Joins With One Join Column

```
SELECT  *
FROM    Reserves R1, Sailors S1
WHERE   R1.sid=S1.sid
```

- **In algebra: $R \bowtie S$. Common! Must be carefully optimized. $R \times S$ is large; so, $R \times S$ followed by a selection is inefficient.**
- **Assume: M tuples in R, p_R tuples per page, N tuples in S, p_S tuples per page.**
  - In our examples, R is Reserves and S is Sailors.
- **We will consider more complex join conditions later.**
- ***Cost metric*: # of I/Os. We will ignore output costs.**

Simple Nested Loops Join

foreach tuple r in R do
    foreach tuple s in S do
        if $r_i == s_j$ then add <r, s> to result

- **For each tuple in the *outer* relation R, we scan the entire *inner* relation S.**
  - Cost: M + p_R * M * N = 1000 + 100*1000*500 I/Os.
- **Page-oriented Nested Loops join: For each *page* of R, get each *page* of S, and write out matching pairs of tuples <r, s>, where r is in R-page and S is in S-page.**
  - Cost: M + M*N = 1000 + 1000*500
  - If smaller relation (S) is outer, cost = 500 + 500*1000

Index Nested Loops Join

foreach tuple r in R do
    foreach tuple s in S where $r_i == s_j$ do
        add <r, s> to result

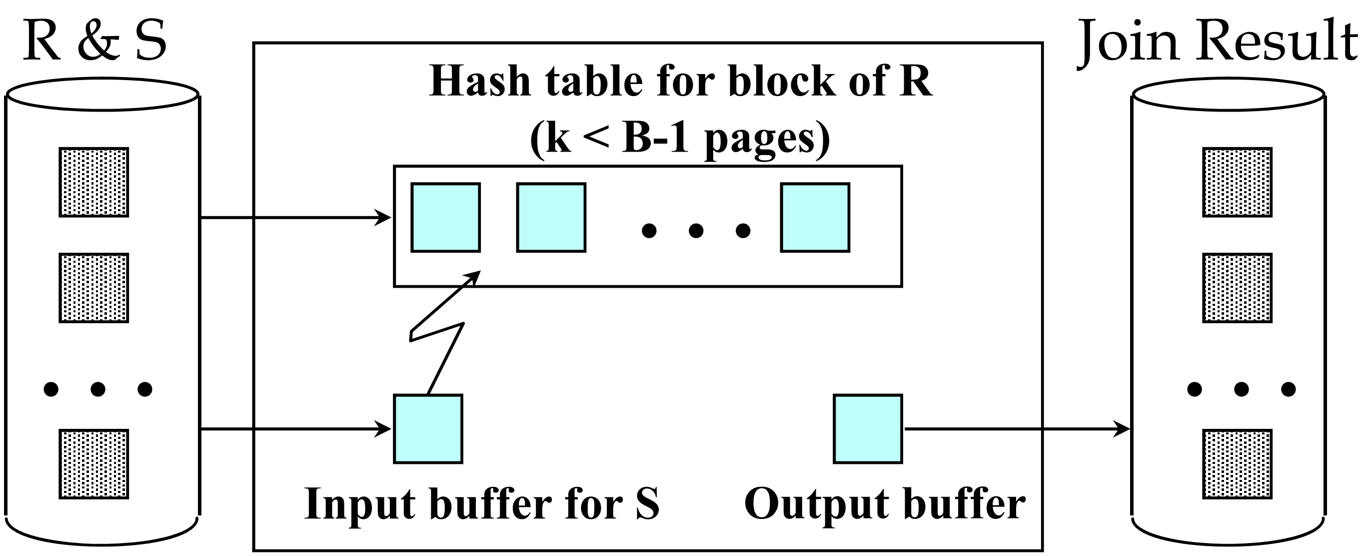
- **If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.**
  - Cost: M + (($M*p_R$) * cost of finding matching S tuples)
- **For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples (assuming Alt. (2) or (3) for data entries) depends on clustering.**
  - Clustered index: 1 I/O (typical), unclustered: upto 1 I/O per matching S tuple.

Examples of Index Nested Loops

- **Hash-index (Alt. 2) on *sid* of Sailors (as inner):**
  - Scan Reserves: 1000 page I/Os, 100*1000 tuples.
  - For each Reserves tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching Sailors tuple. Total: 220,000 I/Os.
- **Hash-index (Alt. 2) on *sid* of Reserves (as inner):**
  - Scan Sailors: 500 page I/Os, 80*500 tuples.
  - For each Sailors tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching Reserves tuples. Assuming uniform distribution, 2.5 reservations per sailor (100,000 / 40,000). Cost of retrieving them is 1 or 2.5 I/Os depending on whether the index is clustered.

Block Nested Loops Join

- **Use one page as an input buffer for scanning the inner S, one page as the output buffer, and use all remaining pages to hold ``block" of outer R.**
  - For each matching tuple r in R-block, s in S-page, add <r, s> to result. Then read next R-block, scan S, etc.



Examples of Block Nested Loops

- **Cost: Scan of outer + #outer blocks * scan of inner**
  - #outer blocks = $\lceil \# \; of \;\; pages \; of \;\; outer \; / \; blocksize \rceil$
- **With Reserves (R) as outer, and 100 pages of R:**
  - Cost of scanning R is 1000 I/Os; a total of 10 *blocks*.
  - Per block of R, we scan Sailors (S); 10*500 I/Os.
  - If space for just 90 pages of R, we would scan S 12 times.
- **With 100-page block of Sailors as outer:**
  - Cost of scanning S is 500 I/Os; a total of 5 blocks.
  - Per block of S, we scan Reserves; 5*1000 I/Os.
- **With *sequential reads* considered, analysis changes: may be best to divide buffers evenly between R and S.**

Sort-Merge Join $(R \bowtie_{i=j} S)$

- **Sort R and S on the join column, then scan them to do a ``merge" (on join col.), and output result tuples.**
  - Advance scan of R until current R-tuple >= current S tuple, then advance scan of S until current S-tuple >= current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value in Ri (*current R group*) and all S tuples with same value in Sj (*current S group*) *match*; output <r, s> for all pairs of such tuples.
  - Then resume scanning R and S.
- **R is scanned once; each S group is scanned once per matching R tuple. (Multiple scans of an S group are likely to find needed pages in buffer.)**

Example of Sort-Merge Join

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

- **Cost: M log M + N log N + (M+N)**
  - The cost of scanning, M+N, could be M*N (very unlikely!)
- **With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500.**

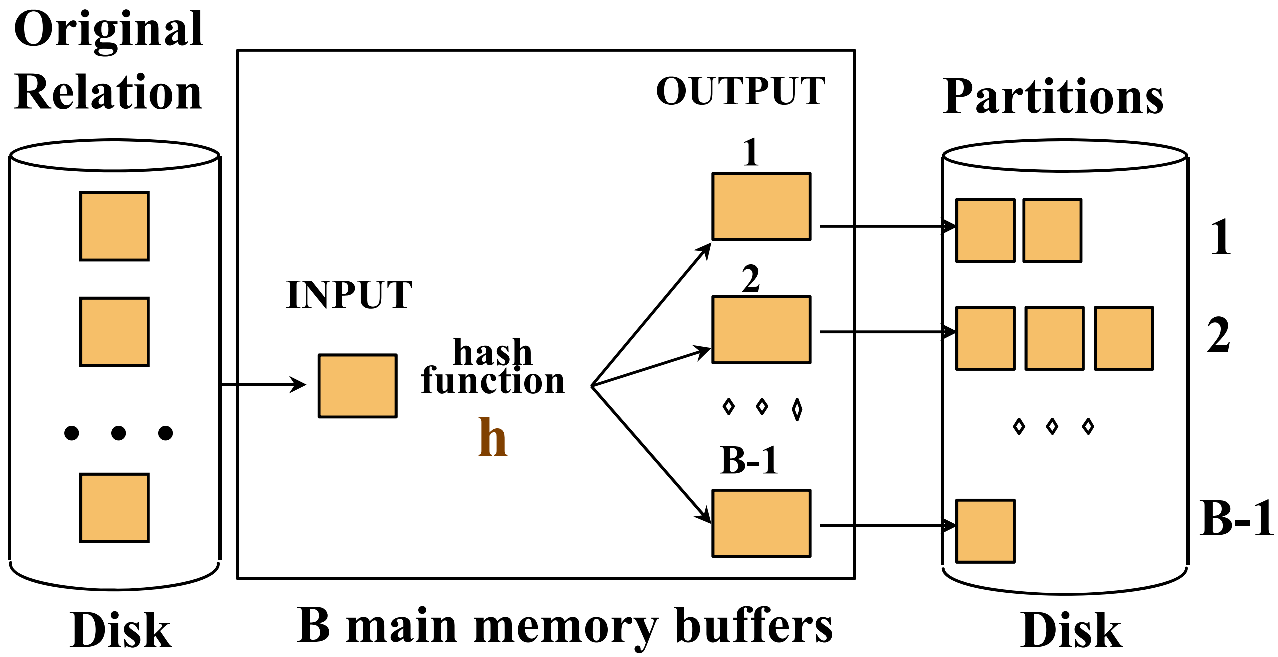
(*BNL cost: 2500 to 15000 I/Os*)

Refinement of Sort-Merge Join

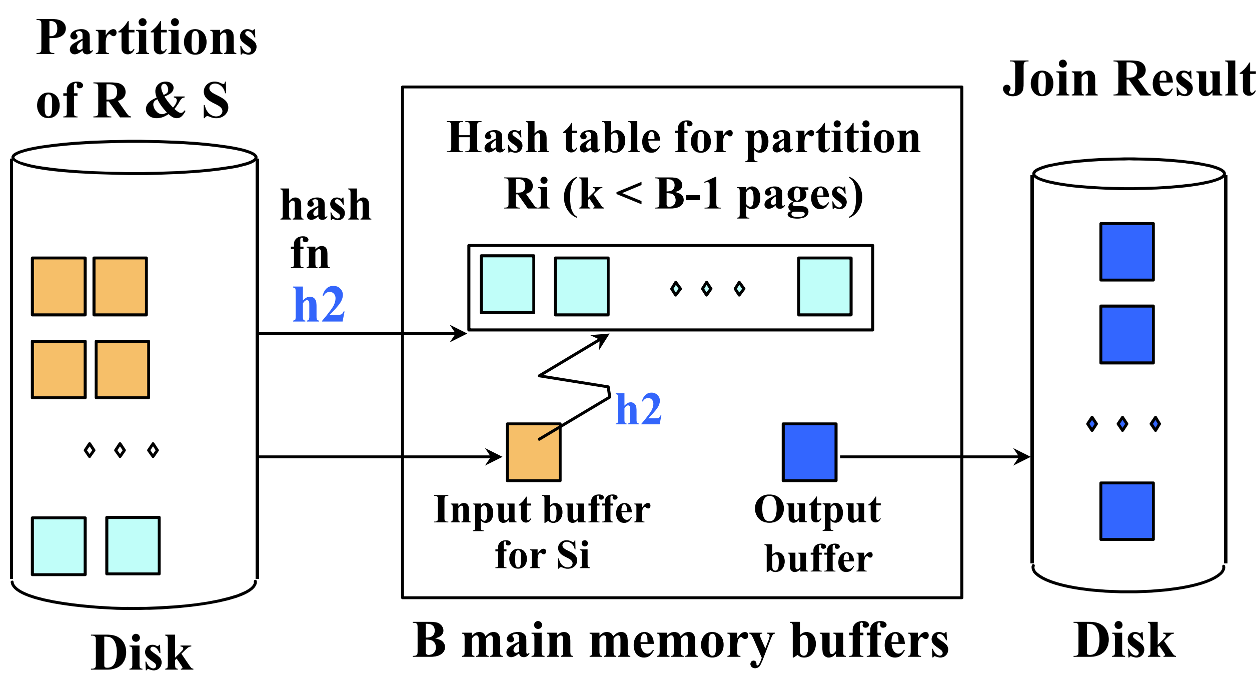
- **We can combine the merging phases in the *sorting* of R and S with the merging required for the join.**
  - With B > $\sqrt{L}$, where *L* is the size of the larger relation, using the sorting refinement that produces runs of length 2B in Pass 0, #runs of each relation is < B/2.
  - Allocate 1 page per run of each relation, and `merge' while checking the join condition.
  - Cost: read+write each relation in Pass 0 + read each relation in (only) merging pass (+ writing of result tuples).
  - In example, cost goes down from 7500 to 4500 I/Os.
- **In practice, cost of sort-merge join, like the cost of external sorting, is *linear*.**

Hash-Join

- **Partition both relations using hash fn h: R tuples in partition i will only match S tuples in partition i.**



- Read in a partition of R, hash it using h2 (<> h!). Scan matching partition of S, search for matches.



Observations on Hash-Join

- **#partitions k < B-1 (why?), and B-2 > size of largest partition to be held in memory. Assuming uniformly sized partitions, and maximizing k, we get:**
  - k= B-1, and M/(B-1) < B-2, i.e., B must be > $\sqrt{M}$
- **If we build an in-memory hash table to speed up the matching of tuples, a little more memory is needed.**
- **If the hash function does not partition uniformly, one or more R partitions may not fit in memory. Can apply hash-join technique recursively to do the join of this R-partition with corresponding S-partition.**

Cost of Hash-Join

- **In partitioning phase, read+write both relns; 2(M+N). In matching phase, read both relns; M+N I/Os.**
- **In our running example, this is a total of 4500 I/Os.**
- **Sort-Merge Join vs. Hash Join:**
  - Given a minimum amount of memory (*what is this, for each?*) both have a cost of 3(M+N) I/Os. Hash Join superior on this count if relation sizes differ greatly. Also, Hash Join shown to be highly parallelizable.
  - Sort-Merge less sensitive to data skew; result is sorted.

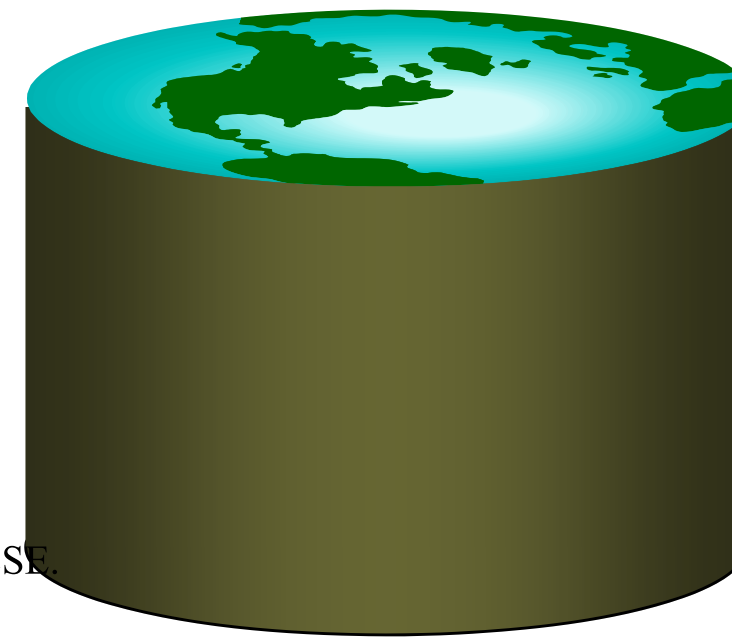
General Join Conditions

- **Equalities over several attributes (e.g., *R.sid=S.sid* AND *R.rname=S.sname*):**
  - For Index NL, build index on *<sid, sname>* (if S is inner); or use existing indexes on *sid* or *sname*.
  - For Sort-Merge and Hash Join, sort/partition on combination of the two join columns.
- **Inequality conditions (e.g., *R.rname < S.sname*):**
  - For Index NL, need (clustered!) B+ tree index.
    - Range probes on inner; # matches likely to be much higher than for equality joins.
  - Hash Join, Sort Merge Join not applicable.
  - Block NL quite likely to be the best join method here.

Conclusions

- **Database needs to run queries fast**
- **Sorting efficiently is one factor**
- **Choosing the right join another factor**
- **Next time: optimizing all parts of a query**

Query Optimization



Review – the Big Picture

- **Data Modelling**
  - Relational
  - E-R
- **Storing Data**
  - File Indexes
  - Buffer Pool Management
- **Query Languages**
  - SQL
  - Relational Algebra
  - Relational Calculus
  - Formal Query Languages permit Query Optimization

Review – Query Processing

- **External Sorting with Merge sort**
  - It is possible to sort most tables in 2 passes
- **Join Algorithms**
  - Nested Loops
  - Indexed Nested Loops
  - Sort-Merge Join
  - Hash Join

Review – Cost of Join Methods

- **Blocked Nested Loops**

  $M + \lceil M / B \rceil * N$

- **Indexed Nested Loops**

  M + (($M*p_R$) * cost to find matching tuples)

- **Sort-Merge Join**

  between 3(M+N) and M*N

- **Hash Join**

  3(M+N) and higher, especially with skewed data

Here: Finding a Better Query

- **Query Languages based on formal foundation**
- **Just as regular algebra expressions can be rewritten, so can relational algebra:**

  $A*B = B*A$, $A(B + C) = AB + AC$, etc.

  $A \times B = B \times C$, etc.

Relational Algebra Equivalences

- **Choose different join orders**
- **`push' selections and projections ahead of joins.**
- ***Selections*: (*Cascade*)** $\sigma_{c1 \wedge \ldots \wedge cn}(R) \equiv \sigma_{c1}(\ \ldots\ \sigma_{cn}(R))$

  $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$ (Commute)

- *Projections:* $\pi_{a1}(R) \equiv \pi_{a1}(\ \ldots(\pi_{an}(R)))$ (Cascade)

- *Joins:* $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$ (Associative)

  $(R \bowtie S) \equiv (S \bowtie R)$ (Commute)

- Show that: $R \bowtie (S \bowtie T) \equiv (T \bowtie R) \bowtie S$

Optimization in a Nutshell

- **Consider access paths to source tables**
  - Tuple Scan
  - Indexes
  - Partitioning

- **Consider equivalent algebra formulas**
  - Estimate costs based on statistics in DBMS

Or, in more detail…

- ***Plan: Tree of R.A. ops, with choice of alg for each op.***
  - Each operator typically implemented using a \`pull' interface: when an operator is \`pulled' for the next output tuples, it \`pulls' on its inputs and computes them.
- **Two main issues:**
  - For a given query, what plans are considered?
    - Algorithm to search plan space for cheapest (estimated) plan.
  - How is the cost of a plan estimated?
- **Ideally: Want to find best plan.**
- **Practically: Avoid worst plans!**
- **We will study the System R approach.**

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,
  - M = 1000 pages total.
- **Sailors:**
  - Each tuple is 50 bytes long,
  - 80 tuples per page,
  - N = 500 pages total.

Statistics and Catalogs

- **Need information about the relations and indexes involved.** *Catalogs* **typically contain at least:**
  - # tuples (NTuples), # pages (NPages) for each relation.
  - # distinct key values (NKeys) and NPages for each index.
  - Index height, low/high key values (Low/High) for each tree index.

- **Catalogs updated periodically.**
  - Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok.

- **More detailed information sometimes stored.**
  - e.g., histograms of the values in some fields

Access Paths

- An <u>access path</u> is a method of retrieving tuples:
  - File scan, or index that matches a selection (in the query)
- A tree index <u>*matches*</u> (a conjunction of) terms that involve only attributes in a *prefix* of the search key.
  - E.g., Tree index on *<a, b, c>* matches the selection *a=5 AND b=3*, and *a=5 AND b>6*, but not *b=3*.
- A hash index <u>*matches*</u> (a conjunction of) terms that has a term *attribute = value* for every attribute in the search key of the index.
  - E.g., Hash index on *<a, b, c>* matches *a=5 AND b=3 AND c=5*; but it does not match *b=3, or a=5 AND b=3, or a>5 AND b=3 AND c=5*.

A Note on Complex Selections

(day<8/9/94 AND rname='Paul') OR bid=5 OR sid=3

- **Selection conditions are first converted to *conjunctive normal form* (CNF):**

  (day<8/9/94 OR bid=5 OR sid=3) AND (rname='Paul' OR bid=5 OR sid=3)

- **We only discuss case with no ORs; see text if you are curious about the general case.**

One Approach to Selections

- **Find the *most selective access path*,**
- **retrieve tuples using it, and**
- **apply any remaining terms that don't match index**
  - index or file scan that estimate will need the fewest I/Os.
  - terms that match index reduce the number of tuples *retrieved*;
  - other terms discard already retrieved tuples, but do not affect I/Os

- **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.
  - Similarly, a hash index on <*bid, sid*> could be used; *day<8/9/94* must then be checked.

Using an Index for Selections

- **Cost depends on #qualifying tuples, and clustering.**
  - Cost of finding qualifying data entries (typically small) plus cost of retrieving records (could be large w/o clustering).
  - In example, assuming uniform distribution of names, about 10% of tuples qualify (100 pages, 10000 tuples). With a clustered index, cost is little more than 100 I/Os; if unclustered, upto 10000 I/Os!

```
SELECT  *
FROM    Reserves R
WHERE   R.rname < 'C%'
```

Projection

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

- The expensive part is removing duplicates.
  - SQL systems don’t remove duplicates unless the DISTINCT is specified.
- Sorting Approach: Sort on <sid, bid> and remove duplicates. (Can optimize by dropping unwanted information while sorting.)
- Hashing Approach: Hash on <sid, bid> to create partitions. Load partitions into memory one at a time, build in-memory hash structure, and eliminate duplicates.
- If there is an index with both R.sid and R.bid in the search key, may be cheaper to sort data entries!

Join: Index Nested Loops

foreach tuple r in R do
foreach tuple s in S where $r_i == s_j$ do
add <r, s> to result

- **If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.**
  - Cost: M + (($M*p_R$) * cost of finding matching S tuples)
- **For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples (assuming Alt. (2) or (3) for data entries) depends on clustering.**
  - Clustered index: 1 I/O (typical), unclustered: up to 1 I/O per matching S tuple.

Examples of Index Nested Loops

- **Hash-index (Alt. 2) on *sid* of Sailors (as inner):**
  - Scan Reserves: 1000 page I/Os, 100*1000 tuples.
  - For each Reserves tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching Sailors tuple. Total: 220,000 I/Os.
- **Hash-index (Alt. 2) on *sid* of Reserves (as inner):**
  - Scan Sailors: 500 page I/Os, 80*500 tuples.
  - For each Sailors tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching Reserves tuples. Assuming uniform distribution, 2.5 reservations per sailor (100,000 / 40,000). Cost of retrieving them is 1 or 2.5 I/Os depending on whether the index is clustered.

Join: Sort-Merge (R $\bowtie_{i=j}$ S)

- **Sort R and S on the join column, then scan them to do a ``merge" (on join col.), and output result tuples.**
  - Advance scan of R until current R-tuple >= current S tuple, then advance scan of S until current S-tuple >= current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value in Ri (*current R group*) and all S tuples with same value in Sj (*current S group*) *match*; output <r, s> for all pairs of such tuples.
  - Then resume scanning R and S.
- **R is scanned once; each S group is scanned once per matching R tuple. (Multiple scans of an S group are likely to find needed pages in buffer.)**

Example of Sort-Merge Join

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

- **Cost: M log M + N log N + (M+N)**
  - The cost of scanning, M+N, could be M*N (very unlikely!)
- **With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500.**

Highlights of System R Optimizer

- **Impact:**
  - Most widely used currently; works well for < 10 joins.
- **Cost estimation: Approximate art at best.**
  - Statistics, maintained in system catalogs, used to estimate cost of operations and result sizes.
  - Considers combination of CPU and I/O costs.
- **Plan Space: Too large, must be pruned.**
  - Only the space of *left-deep plans* is considered.
    - Left-deep plans allow output of each operator to be *pipelined* into the next operator without storing it in a temporary relation.
  - Cartesian products avoided.

Cost Estimation

- **For each plan considered, must estimate cost:**
  - Must estimate *cost* of each operation in plan tree.
    - Depends on input cardinalities.
    - We've already discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)
  - Must also estimate *size of result* for each operation in tree!
    - Use information about the input relations.
    - For selections and joins, assume independence of predicates.

Size Estimation and Reduction Factors

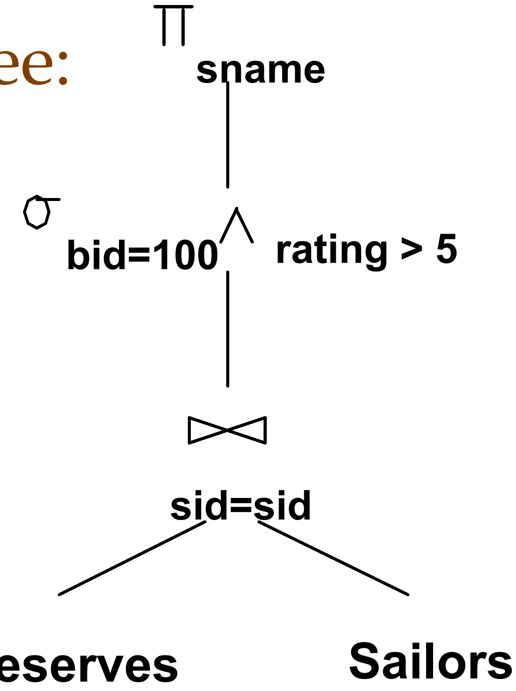
- **Consider a query block:**

```
SELECT  attribute list
FROM  relation list
WHERE  term1 AND ... AND termk
```

- **Maximum # tuples in result is the product of the cardinalities of relations in the FROM clause.**
- ***Reduction factor (RF)* associated with each *term* reflects the impact of the *term* in reducing result size. *Result cardinality* = Max # tuples * product of all RF's.**
  - Implicit assumption that *terms* are independent!
  - Term *col=value* has RF *1/NKeys(I),* given index I on *col*
  - Term *col1=col2* has RF *1/MAX(NKeys(I1), NKeys(I2))*
  - Term *col>value* has RF *(High(I)-value)/(High(I)-Low(I))*

Motivating Example

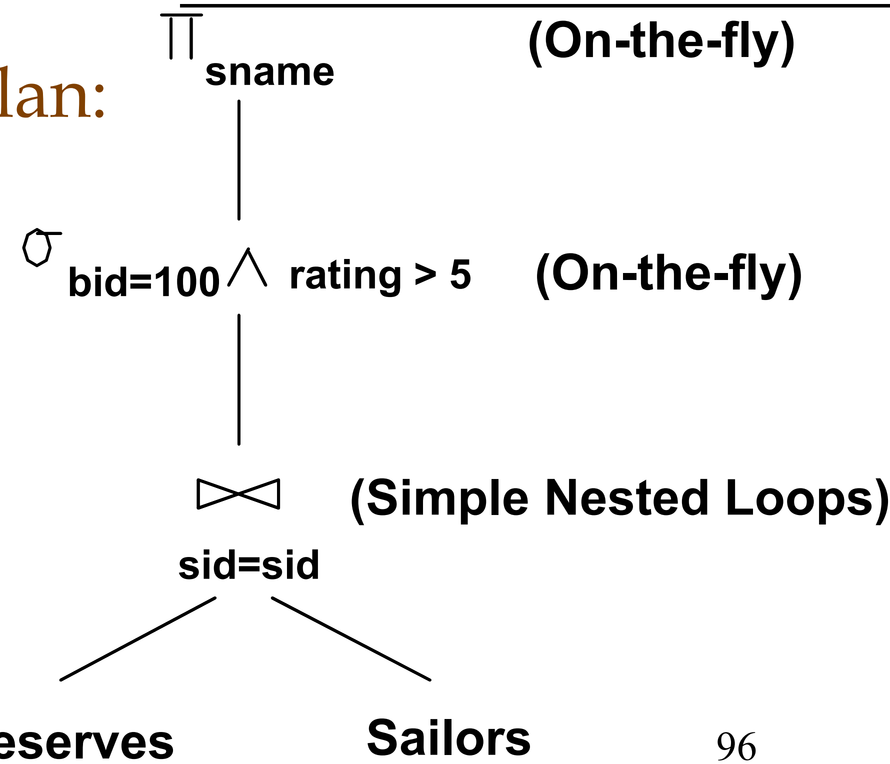
RA Tree:



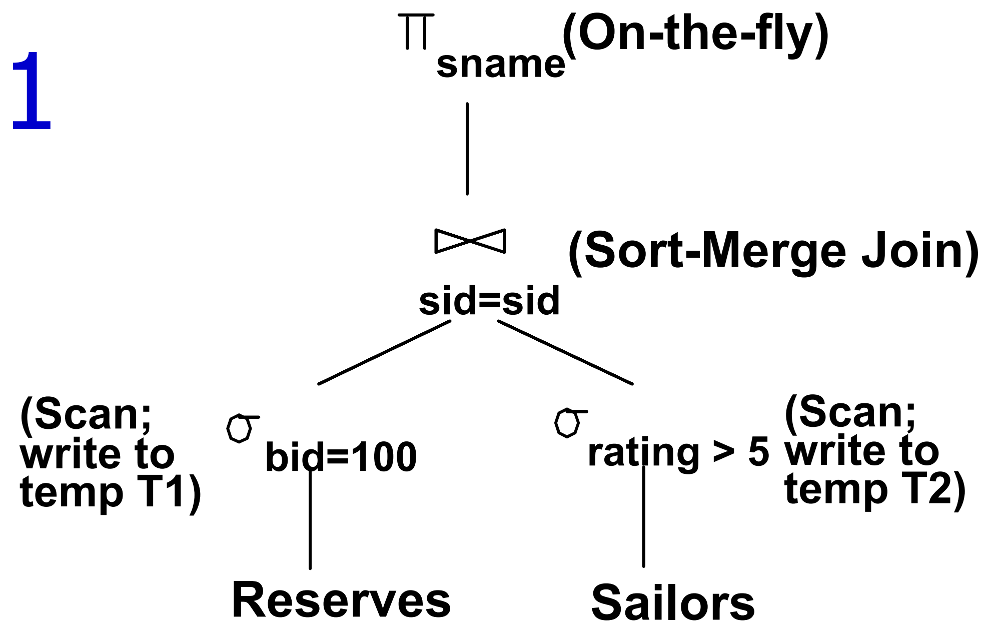

```
SELECT  S.sname
FROM  Reserves R, Sailors S
WHERE  R.sid=S.sid AND
   R.bid=100 AND S.rating>5
```

- **Cost: 500+500*1000 I/Os**
- **By no means the worst plan!**
- **Misses several opportunities: selections could have been `pushed' earlier, no use is made of any available indexes, etc.**
- ***Goal of optimization:* To find more efficient plans that compute the same answer.**

Plan:

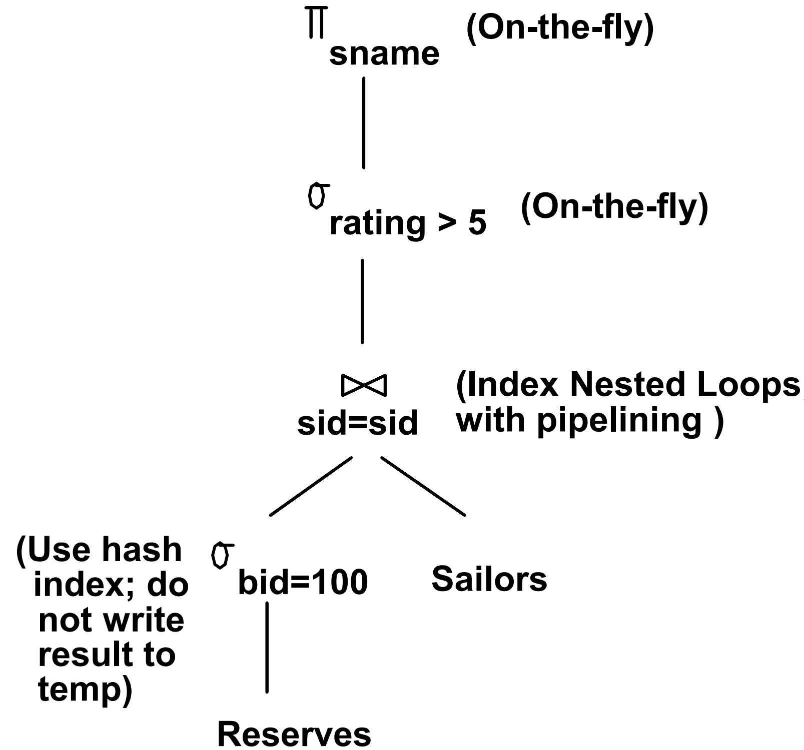


Alternative Plans 1 (No Indexes)



- *Main difference:* ***push selects.***
- **With 5 buffers, cost of plan:**
  - Scan Reserves (1000) + write temp T1 (10 pages, if we have 100 boats, uniform distribution).
  - Scan Sailors (500) + write temp T2 (250 pages, if we have 10 ratings).
  - Sort T1 (2*2*10), sort T2 (2*3*250), merge (10+250)
  - Total: 3560 page I/Os.
- **If we used BNL join, join cost = 10+4*250, total cost = 2770.**
- **If we `push' projections, T1 has only *sid*, T2 only *sid* and *sname*:**
  - T1 fits in 3 pages, cost of BNL drops to under 250 pages, total < 2000.

Alternative Plans 2 With Indexes



- **With clustered index on *bid* of Reserves, we get 100,000/100 = 1000 tuples on 1000/100 = 10 pages.**
- **INL with *pipelining* (outer is not materialized).**

  –Projecting out unnecessary fields from outer doesn't help.

- Join column *sid* is a key for Sailors.
  - –At most one matching tuple, unclustered index on *sid* OK.
- Decision not to push *rating*>5 before the join is based on availability of *sid* index on Sailors.
- Cost: Selection of Reserves tuples (10 I/Os); for each, must get matching Sailors tuple (1000*1.2); total 1210 I/Os.

Summary

- **Several alternative evaluation algorithms for each operator.**
- **Query evaluated by converting to a tree of operators and evaluating the operators in the tree.**
- **Must understand query optimization in order to fully understand the performance impact of a given database design (relations, indexes) on a workload (set of queries).**
- **Two parts to optimizing a query:**
  - Consider a set of alternative plans.
    - Must prune search space; typically, left-deep plans only.
  - Must estimate cost of each plan that is considered.
    - Must estimate size of result and cost for each plan node.
    - *Key issues*: Statistics, indexes, operator implementations.