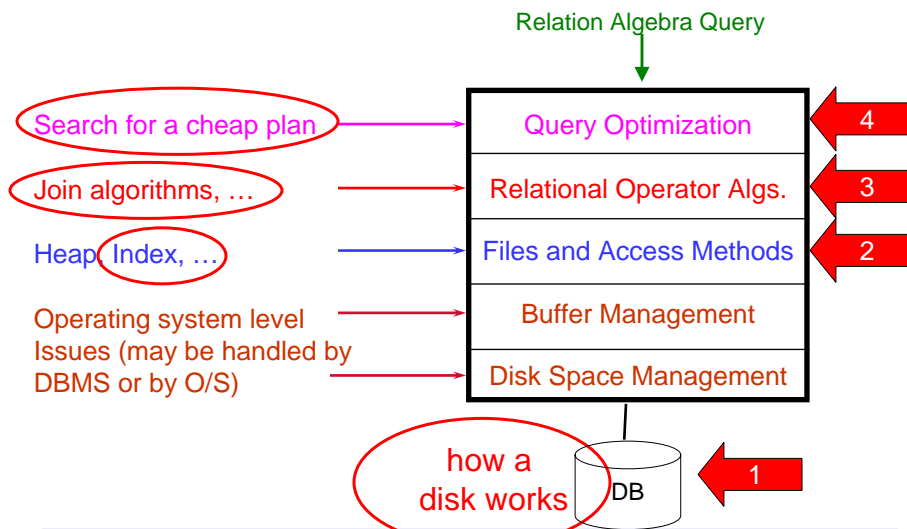


## Lecture 7

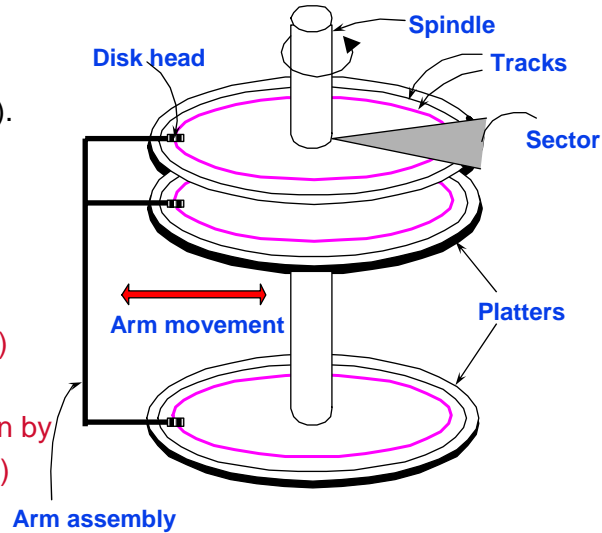
- Storage
  - Disk Architectures
- Indexes
  - Definition
  - Classification
    - Tree (B+) vs. Hash
    - Clustered vs Non-Clustered
    - Sparse vs. Dense
  - Composite Search Keys
- Join Algorithms
  - Nested Loop
  - Sort-Merge
  - Hash

We'll just introduce these ideas  
and we'll start from bottom



## Components of a Disk

- platters are always spinning (say, 2400rps).
- one head reads/writes at any one time.
- to read a record:
  - position arm (seek)
  - engage head
  - wait for data to spin by
  - read (transfer data)

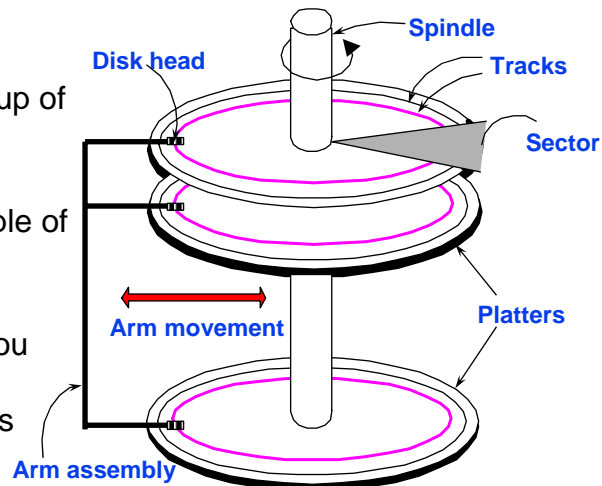


CS386 Introduction to Databases, © Lois Delcambre, David Maier 1998-2006  
Some slides adapted from R. Ramakrishnan, with permission

Slide 3  
Lecture 7

## More terminology

- ❖ Each track is made up of fixed size sectors.
- ❖ Page size is a multiple of sector size.
- ❖ All the tracks that you can reach from one position of the arm is called a cylinder (imaginary!).



CS386 Introduction to Databases, © Lois Delcambre, David Maier 1998-2006  
Some slides adapted from R. Ramakrishnan, with permission

Slide 4  
Lecture 7

## Some Disk Links

### Photos of disks

- Wikipedia

[http://en.wikipedia.org/wiki/Hard\\_disk](http://en.wikipedia.org/wiki/Hard_disk)

- HowStuffWorks (warning: lots of ads)

<http://computer.howstuffworks.com/hard-disk.htm>

### Example disk numbers

- Seagate Barracuda 7200.8

<http://www.bigbruin.com/reviews05/seagate200gb/index.php?file=1>

## Analogy for Disk Access

- Have a library with 10 floors
- Each floor has 50 rows of stacks
- Each row has 20 stacks of shelves
- Each stack holds 400 books

How do you get a particular book?

## Get a Book from the Library

1. Take the elevator to the right floor
2. Walk to the right row
3. Walk down row to right stack
4. Scan the stack for the right book

---

CS386 Introduction to Databases, © Lois Delcambre, David Maier 1998-2006  
Some slides adapted from R. Ramakrishnan, with permission

Slide 7  
Lecture 7

## Access a Record on Disk

1. Take the elevator to the right floor  
Select the right head
2. Walk to the right row  
Seek to the right track
3. Walk down row to right stack  
Rotate to the right page
4. Scan the stack for the right book  
Move the block to memory, check each record

---

CS386 Introduction to Databases, © Lois Delcambre, David Maier 1998-2006  
Some slides adapted from R. Ramakrishnan, with permission

Slide 8  
Lecture 7

## Cost of Accessing Data on Disk

- Time to access (read/write) a disk block:
  - *seek time* (moving arms to position disk head on track)
  - *rotational delay* (waiting for block to rotate under head)
  - *transfer time* (actually moving data to/from disk surface)
- Key to lower I/O cost: **reduce seek/rotation delays!**  
(to reduce transfer time, you need multiple disks)
- Query cost is often measured in the number of page I/Os – often simplified to assume each page I/O costs the same  
Random I/O is more expensive than Sequential I/O

## Disk Drive Statistics

Sector size: 512 bytes

### Seek time

Average 4-10 ms

Track to track .6-1.0 ms

**Average Rotational Delay** - 3 to 6 ms (rotational speed 10,00 RPM to 5,400RPM, half a rotation)

### Transfer Time - Sustained data rate

.3-.1 msec per 8K page, or 25-75 Mbytes/second

### Density

300Gbits/square inch

## 100,000 times slower than memory

- ~100ns for memory, 10ms for disk
- contrast 1 second (to pick up a piece of paper)  
vs.
- driving to San Francisco to get it (over a day there and back)

## However...

Getting one word from memory, 8Kbytes  
from disk

With 2x the time, and sequential read

- Get 10x data from memory
- Get 100x data from disk

## Block (page) size vs. record size

- **Page** –smallest unit of transfer supported by OS
- **Block** – Multiple of page, smallest unit of transfer supported by an application.
- Block and page are often used interchangeably.
- “typical” record size ... maybe a few hundred up to 1,000 bytes
- “typical” block size for DBMS is 8K
- When would we choose block size to be **larger**?
- When would we choose block size to be **smaller**?

## How to minimize the cost of Disk I/Os

- If possible, store a file to be read sequentially as follows:
  - Consecutive pages on same track, followed by
  - Consecutive tracks on same cylinder, followed by
  - Consecutive cylinders adjacent to each other
  - First two incur no seek time or rotational delay, seek for third is only one-track.

Remember:  $\text{disk access time} = \text{seek time} + \text{rotational latency} + \text{transfer time}$   
What is saved with this storage pattern?

- In the book, they assume that all I/O operations take the same amount of time. This is a simplification! Real query optimizers would consider sequential vs. random disk reads – because sequential reads are much faster.

## Index for a File

- An *Index* is a data structure that speeds up selections on the *search key field(s)*
- An index transforms a search key  $k$  into a *data entry  $k^*$* .
- Given  $k^*$ , you can get to the record(s) with the search key  $k$  in one I/O.

## Real-life Indexes

- What is the search key? What is the data entry?
  - Book index
  - Grocery store directory
  - Guide to Pioneer Courthouse Square

## Database Indexes

- Given Emp(ID, name, age, address)
- What are the possible search keys?
- What data structure might be used for the index?
- What could be the format of the data entry  $k^*$ ?

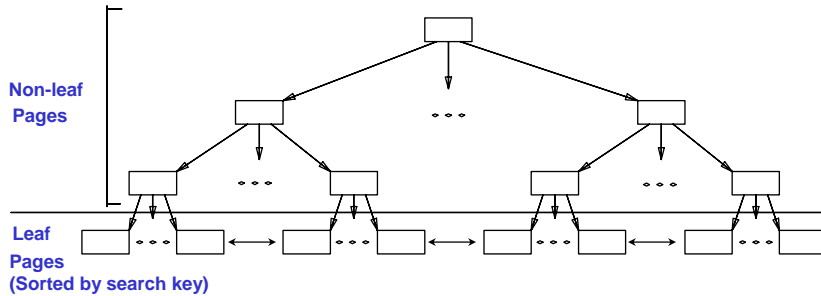
### Nota Bene:

- You can build an index on any subset of the fields of a table.
- You can build more than one index for the same table.
- “*Search key*” is *not* the same as a *key* for the table. Values of a “search key” need not be unique.

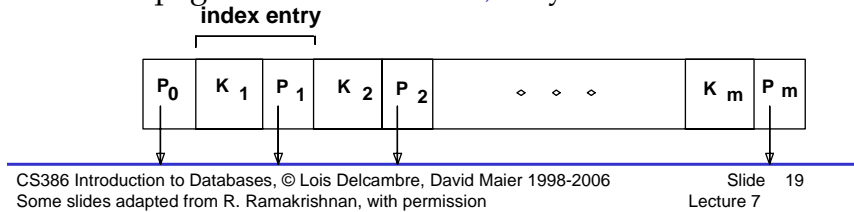
## Tree-Structured Indexes

- Tree-structured indexes support *range searches* and *equality searches*.
  - *ISAM*: static structure (old technology)...index is built just once, when the file is loaded. Uses overflow areas, so the tree can become very unbalanced.
  - *B<sup>+</sup> tree*: dynamic – index is adjusted as records are inserted and deleted in the file. Index remains balanced.

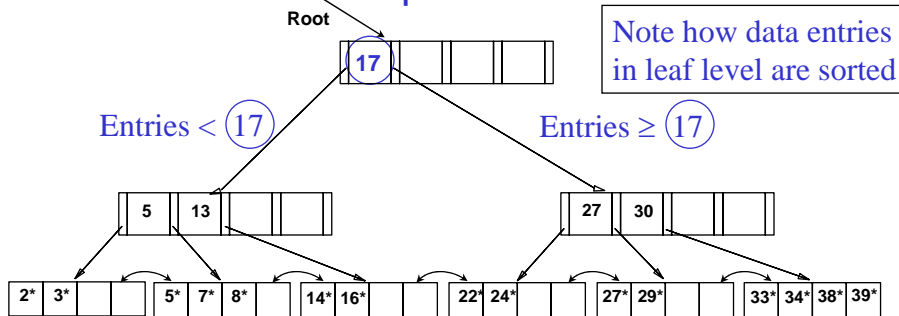
## B+ Tree Indexes



- ❖ Leaf pages contain *data entries*, and are chained (prev & next)
- ❖ Non-leaf pages have *index entries*; only used to direct searches:



## Example B+ Tree



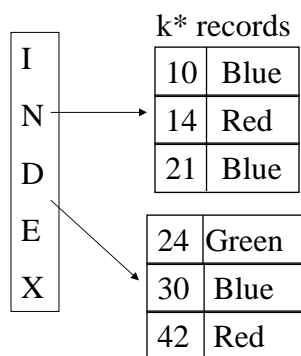
- Find 28\*? 29\*? All > 15\* and < 30\*
- Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes.
  - And change sometimes bubbles up the tree
  - This keeps the tree *balanced*: **each data retrieval takes the same number of I/Os and each page is always at least half full.**

## Alternatives for Data Entry $k^*$ in Index

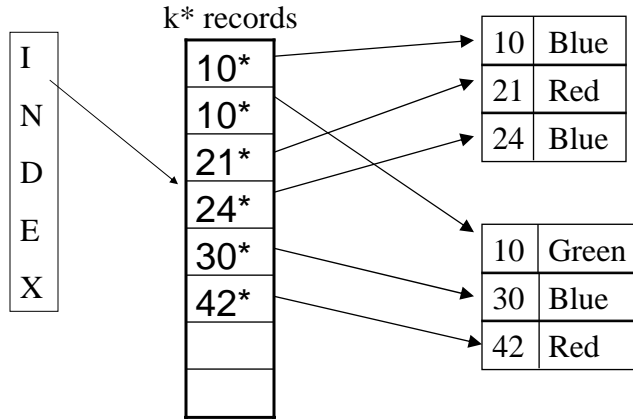
- In a data entry  $k^*$  we can store:
  1. Data record with key value  $k$ , or
  2.  $\langle k, \text{rid of data record with search key value } k \rangle$ , or
  3.  $\langle k, \text{list of rids of data records with search key } k \rangle$
- Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value  $k$ .
  - Examples of indexing techniques: B+ trees, hash-based structures

\*RID = Record id: location in file structure

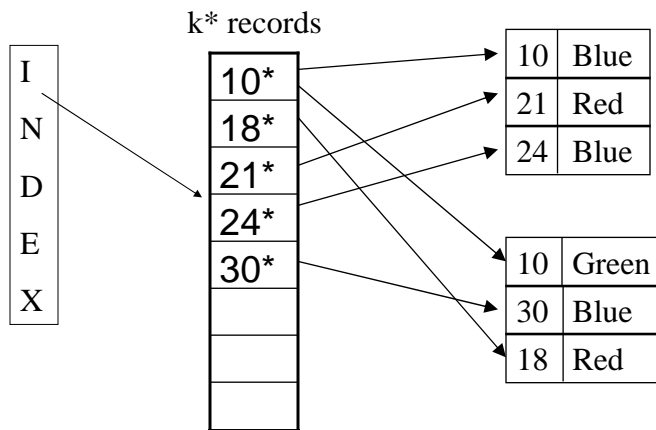
## Alternative 1: $k^*$ holds the data



### Alternative 2: One pointer per $k^*$



### Alternative 3: 1 or more pointers per $k^*$



## Alternatives for Data Entries (Contd.)

- **Alternative 1:**
  - If this is used, index structure is a file organization for data records (instead of a heap file or sorted file).
  - There can be at most one single-attribute index on a given collection of data records using Alternative 1. Why?
  - If data records are very large, # of pages containing data entries is high. If there is one leaf pointer per page, this implies more leaf pointers and perhaps a deeper tree. So Alternative 1 is less attractive in this case.

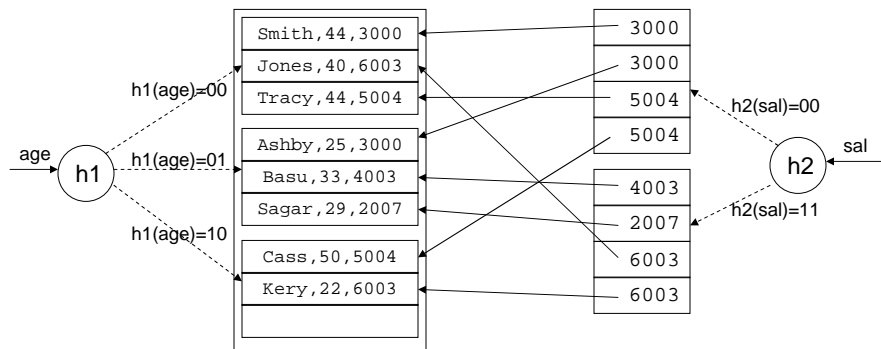
## Alternatives for Data Entries (Contd.)

- **Alternatives 2 and 3:**
  - Alternative 3 is more compact than Alternative 2, but leads to variable-sized data entries even if search keys are of fixed length.

## Hash-Based Indexes

- Good for equality selections.
- Index is a collection of buckets.
  - Bucket = *primary page* plus zero or more *overflow pages*.
  - Buckets contain data entries.
- *Hashing function h*:  $h(r)$  = bucket in which (data entry for) record  $r$  belongs.  $h$  looks at the *search key* fields of  $r$ .
  - No need for “index entries” in this scheme.

## Hash-based Index Examples



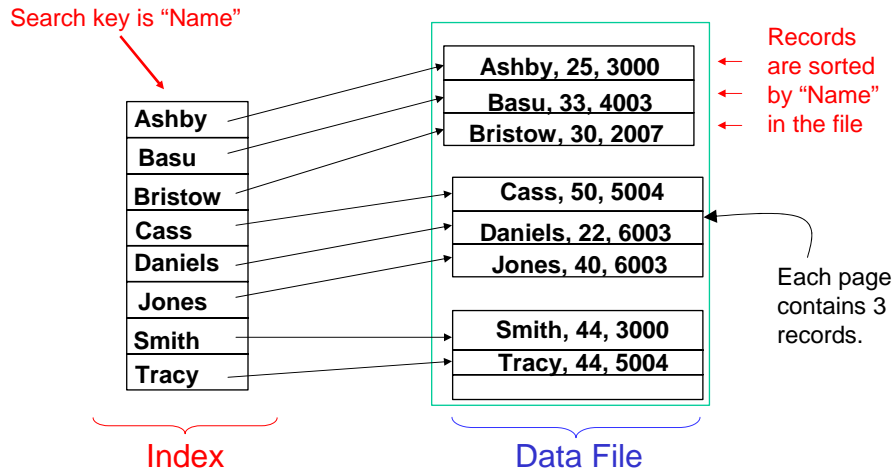
## B+ vs. Hash Indexes

- Typical values
  - Page size 8K, Pointer 4 bytes, key 12 bytes (compression), occupancy 67%. This implies fanout of 343, assume 256 conservatively.
- Typical capacities:
  - Height 4:  $256^4 = 2^{32}$  records
  - Height 3:  $256^3 = 2^{24}$  records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 256 pages = 2 Mbyte
- Therefore direct access costs 2-3 disk probes, not much more than hashing.

## Costs of an Index

- If you define an index in your database, you will incur three **costs**
  - Space to store the index
  - Updates to the search key will be slower
  - The optimizer will take longer because it has more choices
- There is one **advantage** to having an index
  - Some queries run faster (better be sure about this)

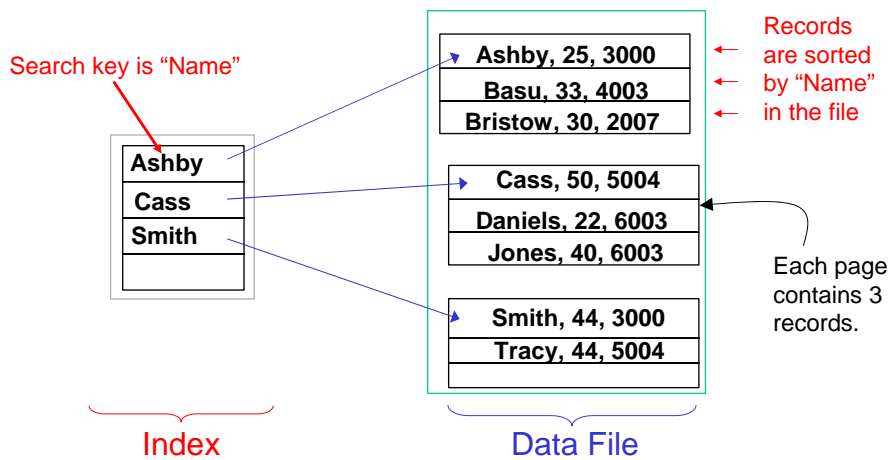
## Clustered Index: Records are sorted based on search key for the index



CS386 Introduction to Databases, © Lois Delcambre, David Maier 1998-2006  
Some slides adapted from R. Ramakrishnan, with permission

Slide 31  
Lecture 7

## Another kind of Clustered Index (Sparse)



CS386 Introduction to Databases, © Lois Delcambre, David Maier 1998-2006  
Some slides adapted from R. Ramakrishnan, with permission

Slide 32  
Lecture 7

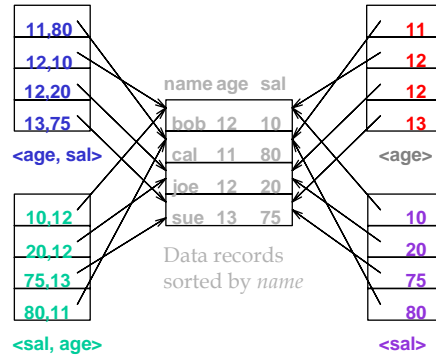


## Using Composite Search Keys

Which indexes can you use for each of these queries?

(use = the data entries are adjacent in the index)

- age = 12
- age = 12 and sal = 20
- age=12 and sal > 10
- age > 12 and sal > 30



### Important refinement for unclustered indexes

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 (though # of such pages likely to be higher than with clustering).

## The simplest join

```
SELECT *
FROM Reserves R, Sailors S
WHERE R.sid=S.sid
```

Sailors				Reserves			
sid	sname	rating	age	sid	bid	day	rname
22	dustin	7	45	28	103	12/4/06	guppy
28	yuppy	9	35	28	103	11/3/06	yuppy
31	lubber	8	55	31	101	9/11/06	dustin
44	guppy	5	35	31	102	9/12/06	lubber
45	rusty	10	35	31	101	10/9/06	lubber
				58	103	12/3/06	dustin

$M = 1000$  pages in  $R$ ,  $p_R = 100$  tuples per page,  
 $N = 500$  pages in  $S$ ,  $p_S = 80$  tuples per page.

CS386 Introduction to Databases, © Lois Delcambre, David Maier 1998-2006  
 Some slides adapted from R. Ramakrishnan, with permission

Slide 37  
 Lecture 7

## Join Algorithms – an Introduction

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

$R \bowtie S$  is very common! And  $R \times S$  followed by a selection is inefficient. So we process joins (rather than cross products) whenever possible. Lots of effort invested in join algorithms.

Assume:  $M$  pages in  $R$ ,  $p_R$  tuples per page,  $N$  pages in  $S$ ,  $p_S$  tuples per page.

In our examples,  $R$  is Reserves and  $S$  is Sailors.

Our algorithms work for any **equijoins**.

CS386 Introduction to Databases, © Lois Delcambre, David Maier 1998-2006  
 Some slides adapted from R. Ramakrishnan, with permission

Slide 38  
 Lecture 7

## Simple Nested Loops Join (very naive)

```

Join on ith column of R and jth column of S
foreach tuple r in R do
    foreach tuple s in S do
        if ri == sj then add <r, s> to result
    
```

- For each **tuple** in the *outer* relation R, we scan the entire *inner* relation S, tuple by tuple.
  - Cost:  $M + (p_R * M) * N = 1000 + 100 * 1000 * 500$  I/Os
  - 50,001,000 I/Os**  $\approx 500,010$  seconds  $\approx 6$  days

**We assume approximately 100 I/Os per second**

M = 1000 pages in R,  $p_R = 100$  tuples per page,

N = 500 pages in S,  $p_S = 80$  tuples per page.

## Simple Nested Loops Join (yes, this is dumb)

Table 1  
on disk

2	...
12	...
6	...
1	...
5	...
27	...

Memory Buffers:

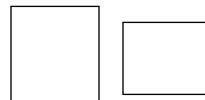
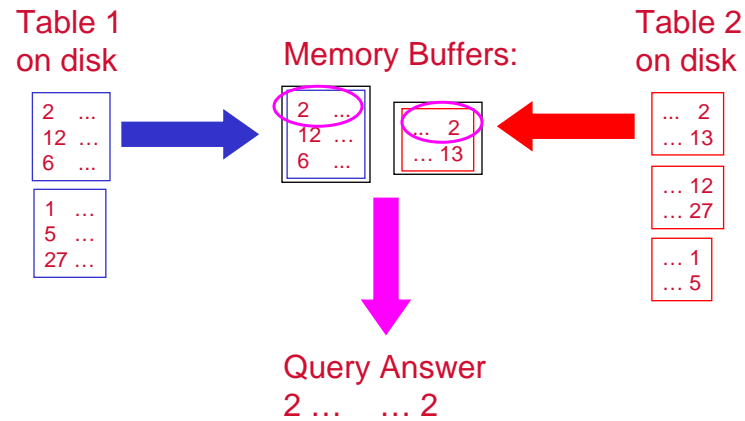


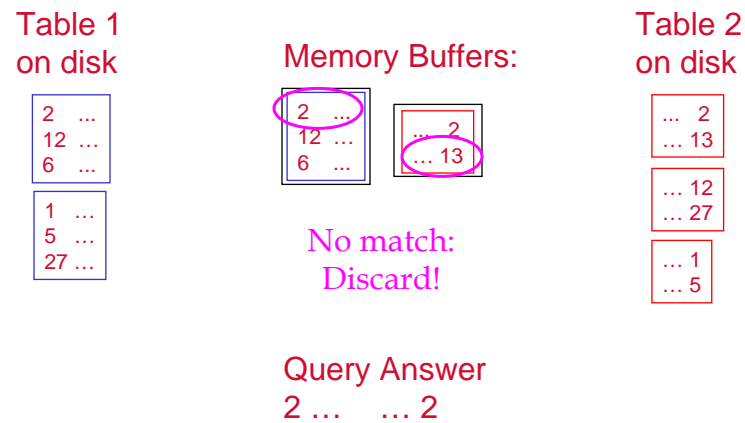
Table 2  
on disk

...	2
...	13
...	12
...	27
...	1
...	5

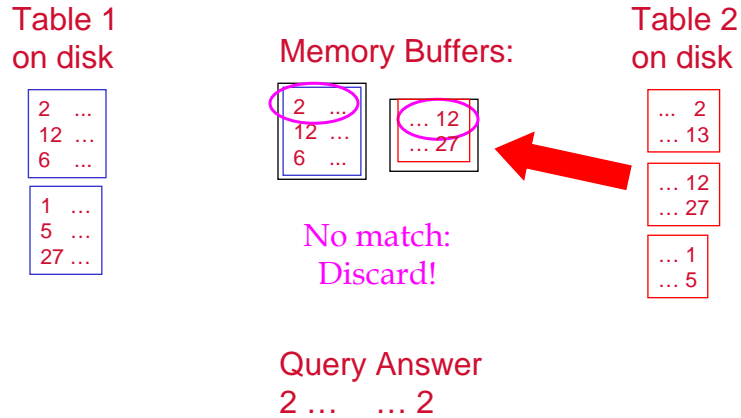
### Simple Nested Loops Join (still dumb)



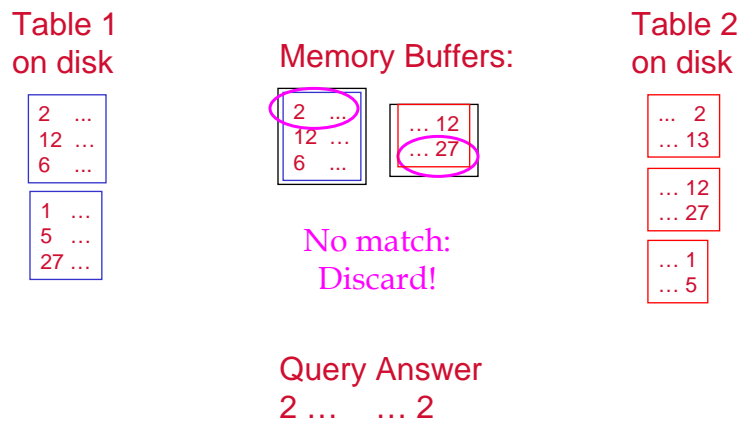
### Simple Nested Loops Join (very, very dumb)



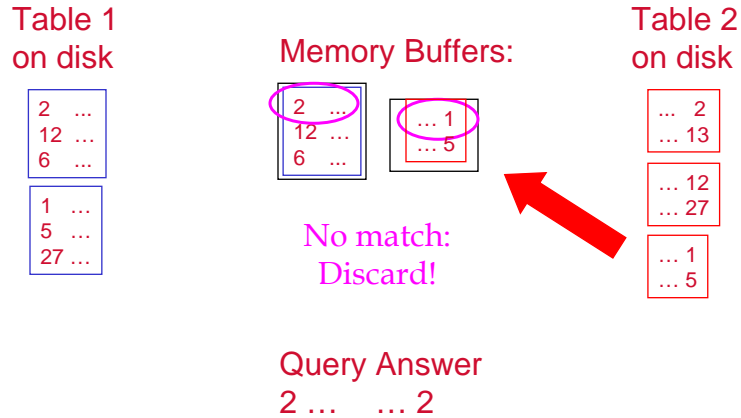
### Simple Nested Loops Join (yawn)



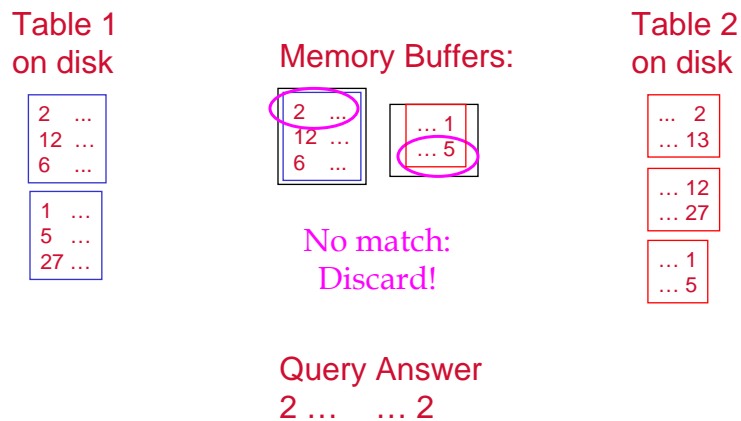
### Simple Nested Loops Join (time for some coffee)



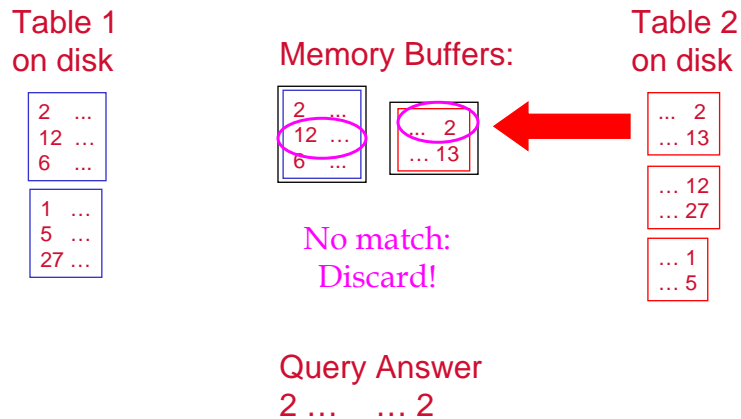
### Simple Nested Loops Join (zzzzz...)



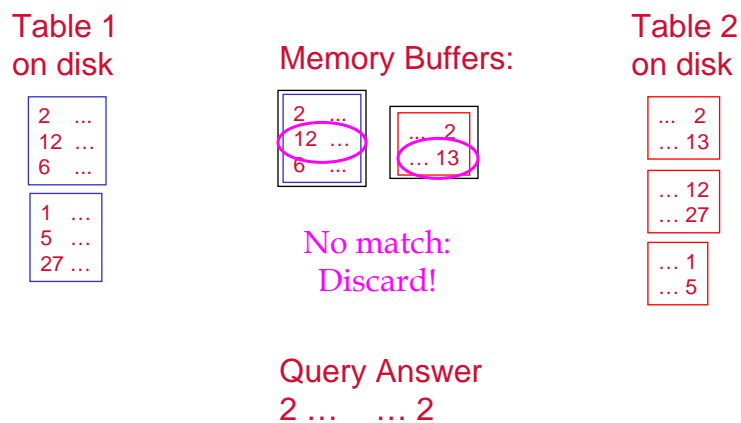
### Simple Nested Loops Join (please make it stop!)



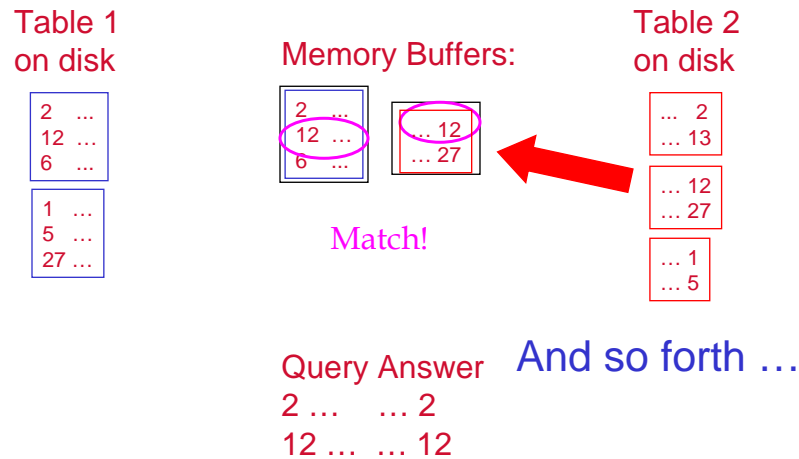
### Simple Nested Loops Join (99 bottles of beer on the wall...)



### Simple Nested Loops Join (this space for rent)



### Simple Nested Loops Join (if this is the best way, I'm outta here)



### Page-oriented Nested Loops Join

```

for each page of tuples r in R do
  for each page of tuples s in S do
    (match all combinations in memory)
    if ri == sj then add <r, s> to result
    
```

For each *page* of R, get each *page* of S, write out matching pairs of tuples <r, s>.

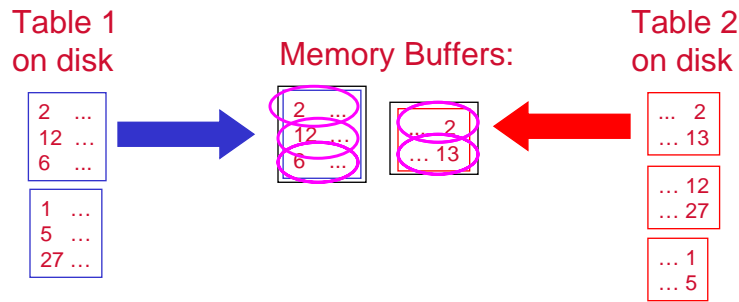
Cost:  $M + M*N = 1000 + 1000*500 = 501,000$  (R outer)

Cost:  $N + N*M = 500 + 500*1000 = 500,500$  (S outer)

Therefore – typically use smaller relation as outer relation.

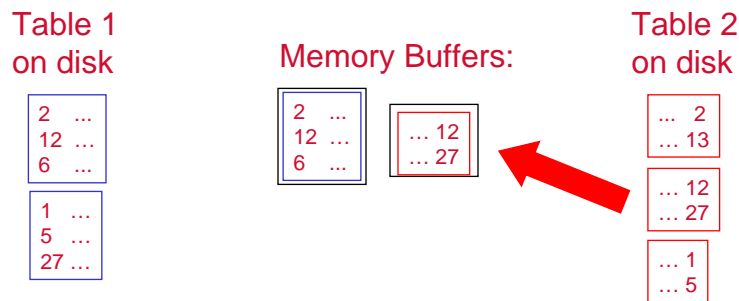
500,000 I/Os  $\approx$  1.4 hours

## Page-Oriented Nested Loops Join



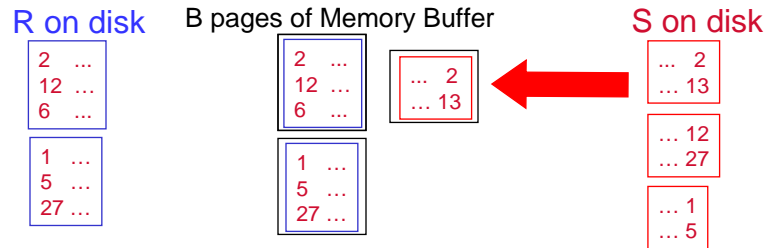
Once we've got these two pages in memory, check every combination from one page to the other page!

## Page-Oriented Nested Loops Join



Do the same thing...compare all combinations in memory - between these two pages!

## The best loops-based join algorithm: Block Nested-Loops Join



- Algorithm:
    - One page is assigned to be the output buffer
    - One page assigned to input from S, B-2 pages assigned to input from R
- ```

Until all of R has been read {
  Read in B-2 pages of R
  For each page in S {
    Read in the single S page
    Check pairs of tuples in memory and output if they match } }
    
```

Cost:  $M + (M/(B-2)) * N$ .

For  $B=35$ , cost is  $1000 + 1000 * 500 / 33 = 16,000$  I/Os  $\approx 3$  minutes

CS386 Introduction to Databases, © Lois Delcambre, David Maier 1998-2006  
Some slides adapted from R. Ramakrishnan, with permission

Slide 53  
Lecture 7

## 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 search key  $s_j$  then can use the index on the inner table - get matching tuples!

Cost:  $M + (M * p_R) * \text{cost of finding matching S tuples}$   
 $= 500 + (500 * 80 * 4) = 160,500 \approx 1/2$  hour (Reserves as inner)  
 $= 1000 + (1000 * 100 * 3) = 301,000 \approx 1$  hour (Sailor as inner)

These could be smaller – if top levels of B+ tree are in memory

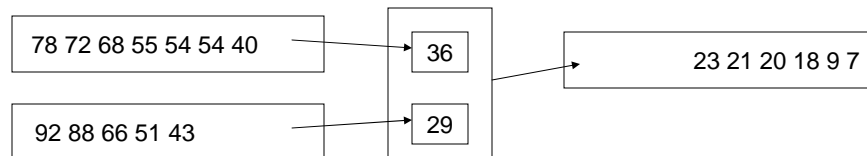
For each R tuple, cost of probing S index is about 2-4 for B+ tree.

CS386 Introduction to Databases, © Lois Delcambre, David Maier 1998-2006  
Some slides adapted from R. Ramakrishnan, with permission

Slide 54  
Lecture 7

## External Sorting

- Various relational operator algorithms require sorting a table
- Issue: table won't fit in memory
- Approach: Use merge-sort where sorted runs can be read sequentially into memory



CS386 Introduction to Databases, © Lois Delcambre, David Maier 1998-2006  
Some slides adapted from R. Ramakrishnan, with permission

Slide 55  
Lecture 7

## External Sorting – Really

- Read and write runs a page at a time
- Do an n-way merge rather than a 2-way merge
- Each pass does  $2 \cdot M$  I/Os (where  $M$  is number of pages in the table)
- Number of passes depends on how many pages of memory are devoted to sorting  
Can sort 100 million pages in 4 passes with 129 pages of memory space
- Can sort  $M$  pages using  $B$  memory pages in **2 passes** if  $\sqrt{M} \leq B$  (this condition is satisfied often)

CS386 Introduction to Databases, © Lois Delcambre, David Maier 1998-2006  
Some slides adapted from R. Ramakrishnan, with permission

Slide 56  
Lecture 7

## Sort-Merge Join

1. Sort R on join attribute
2. Sort S on join attribute
3. Merge R and S
  - Advance scan of R until current R-tuple  $\geq$  current S tuple, then advance scan of S until current S-tuple  $\geq$  current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value in  $R_i$  (*current R group*) and all S tuples with same value in  $S_j$  (*current S group*) match; output  $\langle r, s \rangle$  for all pairs of such tuples.
  - Then resume scanning R and S.

## Sort-Merge Join, cont.

R is scanned once; each S group is scanned once per matching R tuple. Depends on the size of the group! If the matching group is small - matching is in memory.

**Best case: cost is:** Cost to sort R + Cost to sort S +  $(M+N)$  assuming all matches fit in memory

Worst case: R and S all have the same join value – thus the matching group is the entire relation, for R and for S.  
Cost is: Cost to sort R + Cost to sort S +  $(M*N)$

## 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: (cost to sort R) + (cost to sort S) + (M+N) (in memory matches)

With 35 buffers, Reserves and Sailors can each be sorted in 2 passes

Cost is:  $4 * 1000 + 4 * 500 + 1000 + 500 = 7500$

(we multiply by 4 because there are 2 passes, and we read and write each page, each pass)

## Cost of Sort-Merge

(cost to sort R)+(cost to sort S)+(cost to merge)

Cost to sort M pages in 2 passes =  $4 * M$ . Why?

Cost to merge is typically  $M+N$ . Why?

If both R and S can be sorted in 2 passes, then

Cost is:  $4M+4N+(M+N) = 5*(M+N)$

There is an optimization (page 462 in our text) that improves this to  $3*(M+N)$

Thus the cost of joining Sailors and Reservations, assuming there are enough buffers to sort each table in two passes, is

$5*(M+N) = 7500$  Pages

## Hash Join

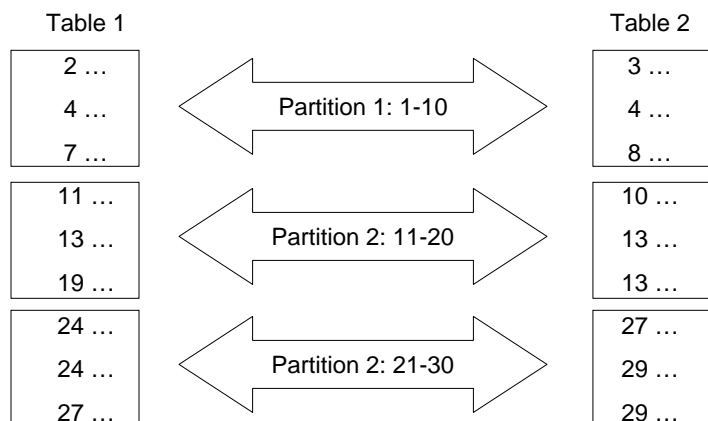
Simple case – S fits in main memory

- Build an in-memory hash index for S
- Proceed as for index nested-loops join

Harder case – neither R nor S fits in memory

- Divide them both in the same way (1 pass) so that each partition of S fits in memory
- Do the simple case on each pair of matching partitions

## Partitioning



## Use Hash Function Instead of Ranges

- No guarantee we can find ranges of values that will divide Table 2 into roughly equal-sized partitions
- Apply hash function  $h$  to join value
  - Partition 1:  $h(\text{val}) = 1$
  - Partition 2:  $h(\text{val}) = 2$
  - Partition 3:  $h(\text{val}) = 3$

## Hash Join Cost

- Cost to partition R:  $2M$
- Cost to partition S:  $2N$
- Can do this in one pass if  $\sqrt{M} \leq B$
- Cost to join partitions:  $M+N$
- Total:  $3*(M+N)$ , same as sort-merge with the optimization.

## Comparison of Approximate Costs of Joining R and S, assuming 100 I/Os/second

| Algorithm           | I/Os       | Time      |
|---------------------|------------|-----------|
| Simple Nested Loops | 50,000,000 | 6 days    |
| Page Nested Loops   | 500,000    | 1.4 hours |
| Block Nested Loops* | 16000      | 3 minutes |
| Index Nested Loops  | 160,500    | ½ hour    |
| Sort-Merge**        | 4,500      | 1 minute  |
| Hash join**         | 4,500      | 1 minute  |

\*Assuming 35 buffer pages

\*\*Assuming appropriate files, M, satisfy  $\sqrt{M} < \text{pages of buffer}$

## So Why Bother with Nested-Loops Join?

It's the most general join

Works with  $<$ ,  $<=$ ,  $<>$ , etc.

## Summary – Algorithms for Relational Algebra Operators

- A virtue of relational DBMSs: *queries are composed of a few basic operators*; the implementation of these operators can be carefully tuned (and it is important to do this!).
- Many alternative implementation techniques for each operator; no universally superior technique for most operators.
- Must consider available alternatives for each operation in a query and choose best one based on system statistics, etc. This is part of the broader task of optimizing a query composed of several ops.