THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

DB Performance

CSCI 2541 Database Systems & Team Projects

Wood

Slides adapted from Prof. Bhagi Narahari, Rahul Simha, and Silberschatz, Korth, and Sudarshan

Disk Access Times

Average time to access a target sector approximated by :

Taccess = T_{avg} seek + T_{avg} rotation + T_{avg} transfer

Seek time (Tavg seek)

- Time to position heads over cylinder containing target sector.
- Typical Tavg seek = 9 ms

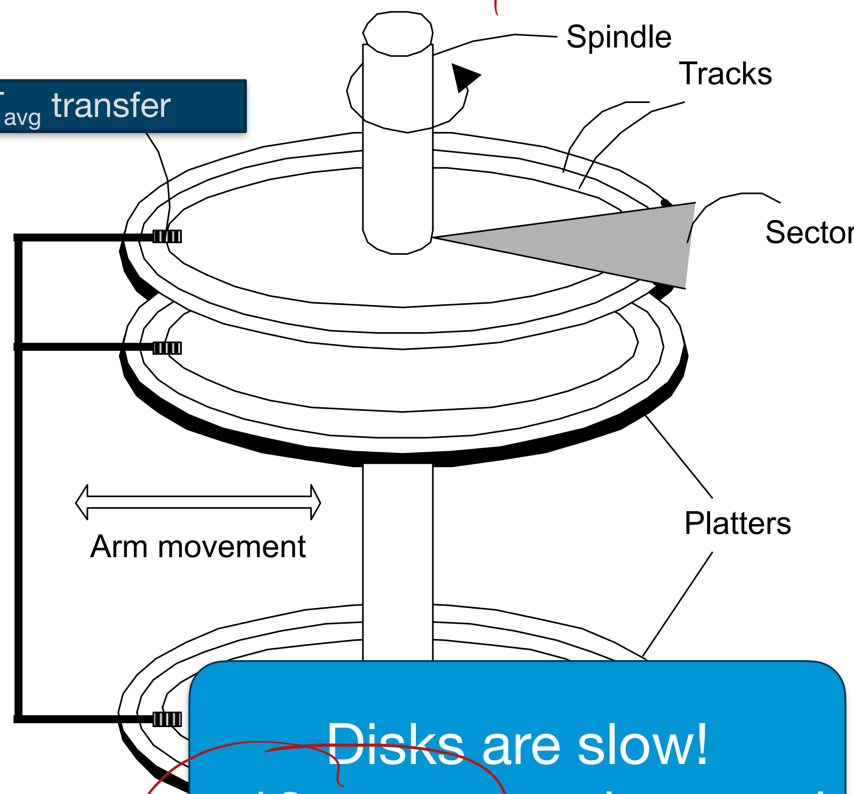
Rotational latency (Tavg rotation)

- Time waiting for first bit of target sector to pass under r/w head.
- Tavg rotation = 1/2 x 1/RPMs x 60 sec/ 1 min = 6 ms

Transfer time (Tavg transfer)

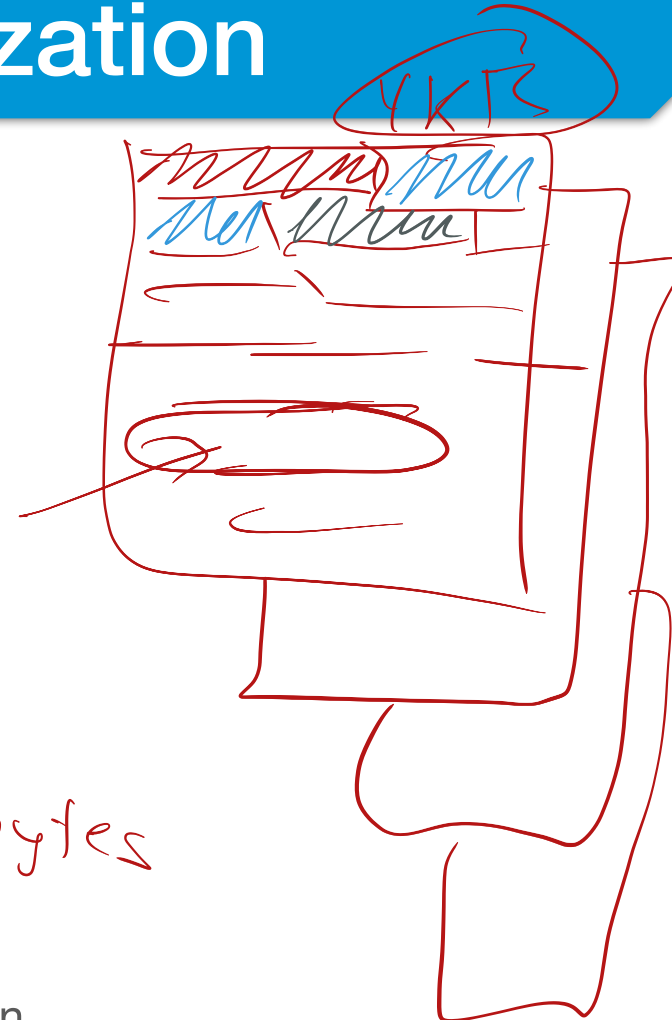
- Time to read the bits in the target sector.
- Tavg transfer = 1/RPM x 1/(avg # sectors/track) x 60 secs/1 min. = ~200 MB/sec

4KB



Disks are slow!
~10ms per random read

Recap: File Organization



Tables mapped as File

- Row is a Record
- Column is field (in record)

Data stored in secondary storage

- Disks – organized as number of disk blocks

Records mapped to disk blocks

Size of file in disk blocks/pages: N

- Number of records/tuples/rows: n
- Size disk block (i.e., page): b bytes
- Size of record (row): r bytes – 200 bytes
- Blocking factor p = b/r
- File size N = n/b pages

Efficiency/performance of a file organization

- Time for Search, Insert, Delete

Example

File of 1,000,000 records

record size 200 bytes

blocks are 4096 bytes

fill in the blanks

- n = 1,000,000
- r = 200
- b = 4096
- Blocking factor (records per block), p = b/r = 20 rec/block
- file size = N = n/p = 50,000 blocks

4096/200 ~ records/block

50,000 × 10ms = 500,000 milliseconds

500 seconds

Example

File of 1,000,000 records

record size 200 bytes

blocks are 4096 bytes

- $n = 1,000,000$
- $r = 200$
- $b = 4096$
- Blocking factor, $p = b/r = 4096/200 = 20$
- file size = $N = n/p = 1,000,000/20 = 50,000$ blocks

File Organizations

File organization determines how records are physically placed on disk

- heap file: no particular order
- sorted file
- indexed file
  - hash index
  - tree indices

Efficiency of file organization typically measured in terms of number of disk/SSD accesses to fetch data

Heap File

Unorganized "heap" of data

Each block has 200 records

1M records, 50K blocks

76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
33465	Gold	Physics	87000

… (1M records, 50K blocks)

```
SELECT * FROM profs
WHERE ID = 231531
```

Worst case query time?

Average query time?

Heap File

Unorganized “heap” of data

Each block has 200 records

1M records, 50K blocks

```
INSERT INTO profs
VALUES (…)
```

76766	Crick	Biology	72000	
10101	Srinivasan	Comp. Sci.	65000	
45565	Katz	Comp. Sci.	75000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	
12121	Wu	Finance	90000	
76543	Singh	Finance	80000	
32343	El Said	History	60000	
58583	Califieri	History	62000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
33465	Gold	Physics	87000	

… (1M records, 50K blocks)

Worst case query time?

Average query time?

Heap File Performance: Example

Successful lookup: average ½ N= 25,000

- worst case is N= n/p= 50,000 disk accesses
- At 10ms disk access time, this is 500 seconds ~ 8 minutes!

insertion = 2 disk accesses

- unless you need to check uniqueness!

deletion = ½(n/p)+1 = 25,001

- worst case = 50,001

Heap file summary: not great

Attempt 1: better organize the records on disk

Heap file will not cut it!

Need to organize physical records on the file in some “smart” manner

- Sorted file
- Hash file

Sorted File

Sort by ID

Each block has 200 records

1M records, 50K blocks

```
SELECT * FROM profs
WHERE ID = 231531
```

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

… (1M records, 50K blocks)

Worst case query time?

Average query time?

Sorted File

Sort by ID

Each block has 200 records

1M records, 50K blocks

```
INSERT INTO profs
VALUES (…)
```

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

… (1M records, 50K blocks)

50K

Worst case query time?

Average query time?

Other approaches...

Sorted File... how long ?

- Search time: Log (Number of disk blocks)
- Log (50,000) blocks = 16 IF the blocks are contiguous on the disk
  - Big/unrealistic assumption that records are stored in consecutive blocks on disk
- Insertion: Could be terrible (N) if we need to rewrite everything in order (in practice we will avoid this)

Even if we don't care about insertion cost, is a sorted file a perfect solution?

Attempt 2: separate lookup from file structure

The structure of the file on disk can't be perfect for all query types! We need to try something else...

Many queries reference small portion records

- DBMS should be able to locate these without having to search all records

Create another type of record (pointer?!) which contains subset of the information in the record

- Analogy: Index in a book or Card Catalog in a library

Index Basics

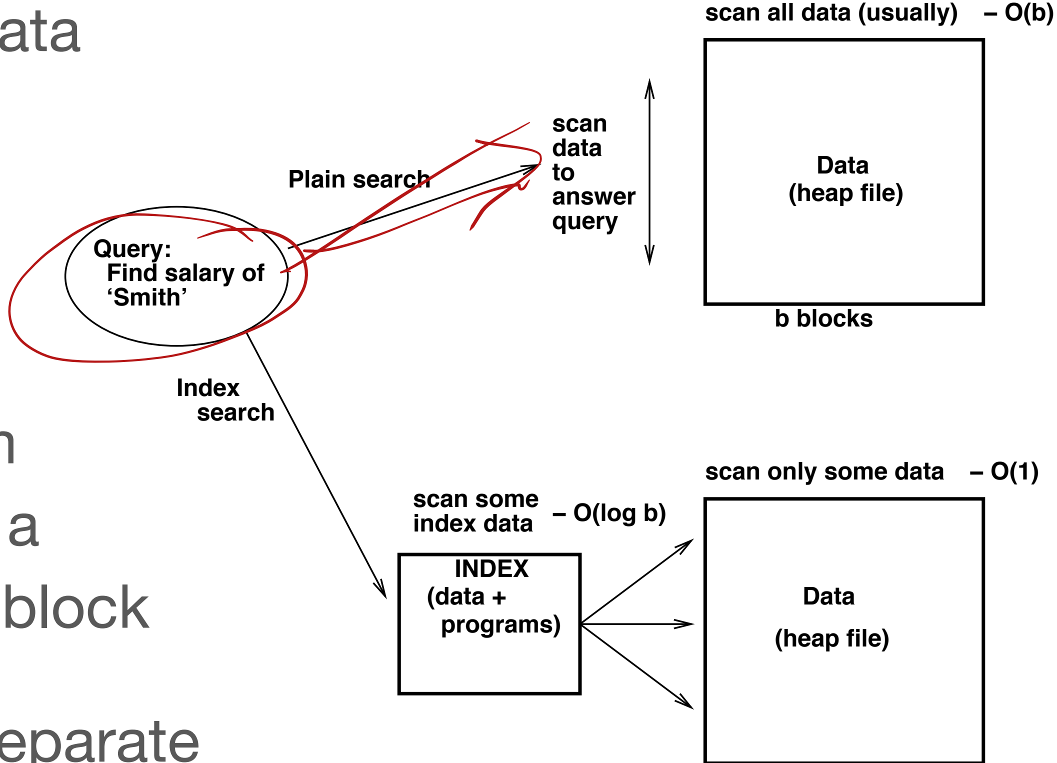
An **index** allows us to more quickly find a piece of data

Search Key

- Value to be searched for

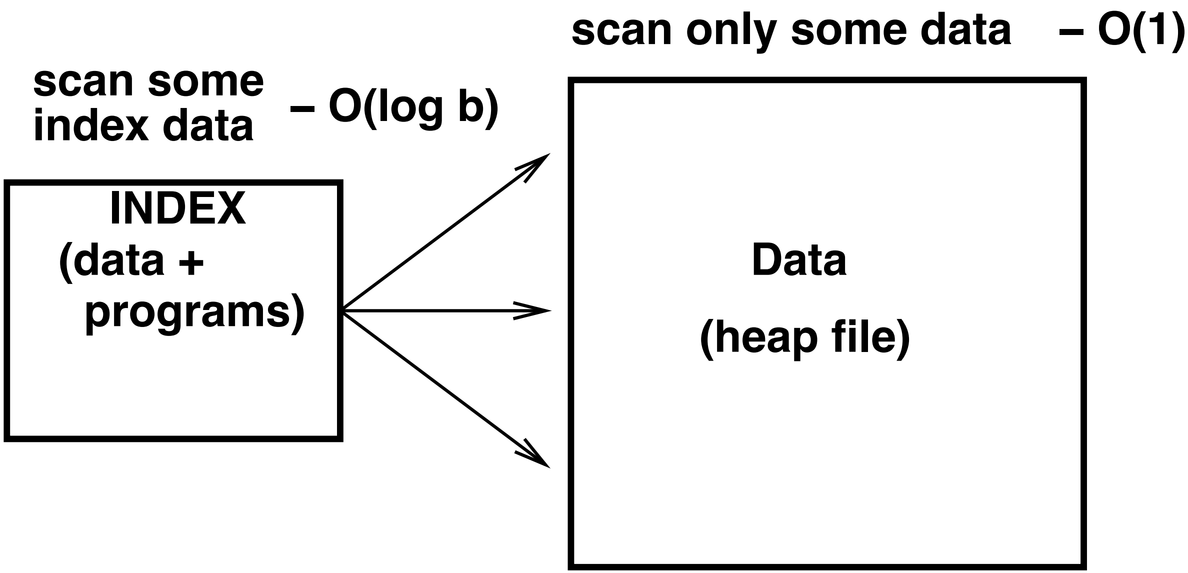
Index maps from a Search Key to a record in a data block

Index will be a separate file on disk - need to keep it up to date!



Index Benefits

Even if we have to scan the entire index, why will this be better than scanning the entire data file?



Dense Index

A **dense index** contains an entry for every data record

Index field specifies what attribute the index lets you search

- A **primary index** is an index on a field that is the primary key of the data file (file might be sorted on the primary key!)
- A **secondary index** is not on a primary key

index

10101	
12121	
15151	
22222	
32343	
33456	
45565	
58583	
76543	
76766	
83821	
98345	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Non-Dense Index?

A **dense index** contains an entry for every data record

Do we really need an index entry for every record?? Why not?

10101	→
12121	→
15151	→
22222	→
32343	→
33456	→
45565	→
58583	→
76543	→
76766	→
83821	→
98345	→

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Non-Dense Index?

A **dense index** contains an entry for every data record

Do we really need an index entry for every record?? Why not?

10101	
32343	
76766	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

If records are sorted, we can use a **sparse index** to jump to the right range, and then do binary search

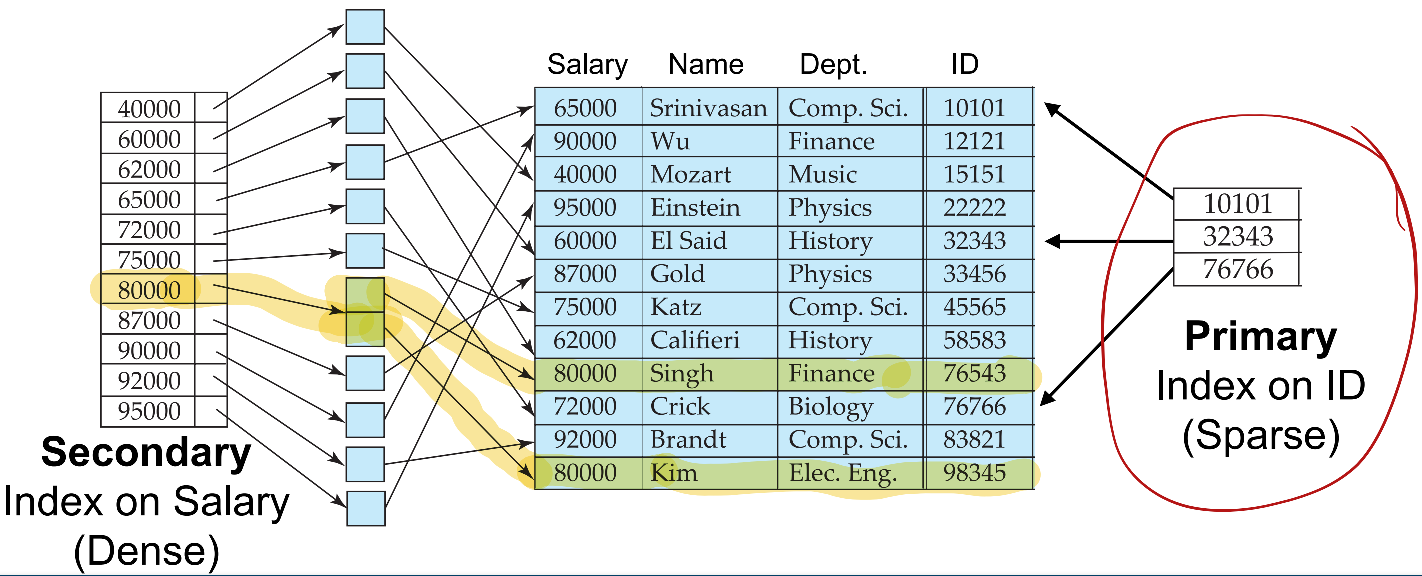
Multiple Indexes

We can have multiple indexes to allow us to find different search keys

- All index files will map to records in the same data file

Secondary index may go to non-unique key! ("**Clustering index**")

- Each index will map to a bucket with pointers to one or more records



Salary	Name	Dept.	ID
65000	Srinivasan	Comp. Sci.	10101
90000	Wu	Finance	12121
40000	Mozart	Music	15151
95000	Einstein	Physics	22222
60000	El Said	History	32343
87000	Gold	Physics	33456
75000	Katz	Comp. Sci.	45565
62000	Califieri	History	58583
80000	Singh	Finance	76543
72000	Crick	Biology	76766
92000	Brandt	Comp. Sci.	83821
80000	Kim	Elec. Eng.	98345

Index Evaluation Metrics

Index methods can be evaluated for functionality, efficiency, and performance.

The **functionality** of an index can be measured by the types of queries it supports. Two query types are common:

- exact match on search key
- query on a range of search key values

The **performance** of an index can be measured by the time required to execute queries and update the index.

- Access time, update, insert, delete time

The **efficiency** of an index is measured by the amount of space required to maintain the index structure.

Index Performance

Our DB: 1M records, 50K disk blocks

Heap file: 50K disk accesses worst case

Sorted File: Log (50,000) blocks = 16

Indexed Sorted File?

- Suppose 10 byte key + 10 byte record pointer = 20 bytes
- 4KB page -> 200 index records per page

How big will index be? How many accesses?

Dense index:

-
-

Sparse index:

-
-

Index Performance

Our DB: 1M records, 50K disk blocks

Heap file: 50K disk accesses worst case

Sorted File: Log (50,000) blocks = 16

Indexed Sorted File?

- Suppose 10 byte key + 10 byte record pointer = 20 bytes
- 4KB page -> 200 index records per page

Dense index:

- 1M records / 200 = 5,000 index ~~pages~~ blocks
- Log(5000) = 12 + 1 = 13 disk accesses

Sparse index: 1 index record per disk block

- 50K / 200 = 250 index ~~pages~~ block
- Log(250) = 8 + 1 = 9 disk accesses

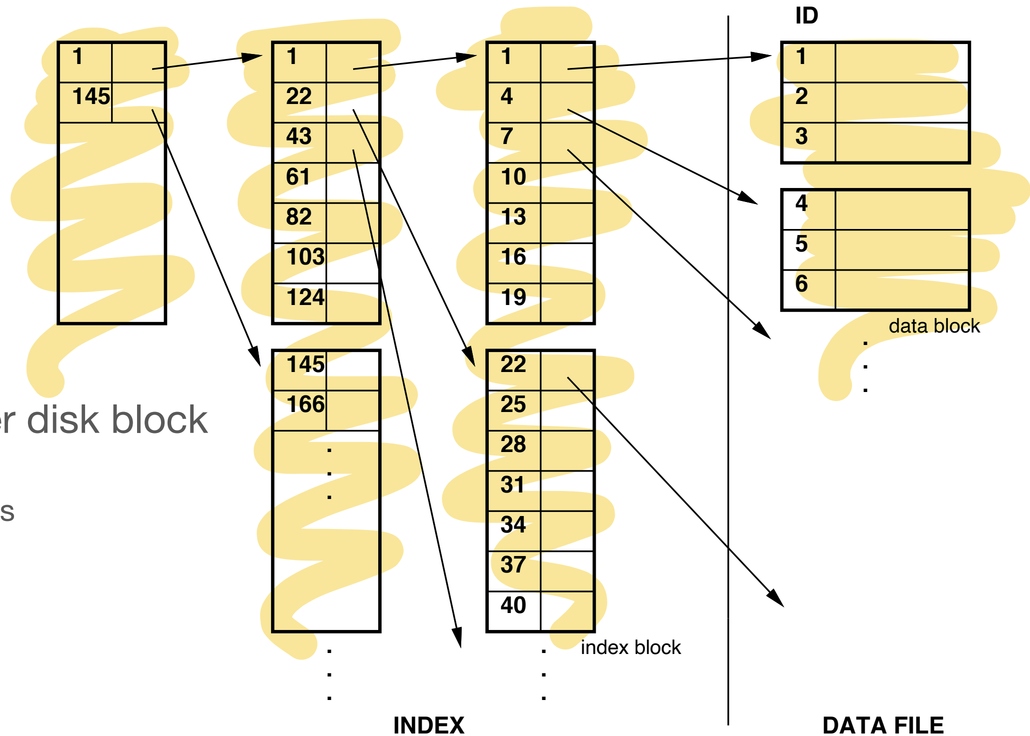
Large Index

What do we do if index gets too large?

Multi-layer Indexes

We can create an index for our index!

Each index layer speeds up search but consumes more space



Sparse index: 1 index record per disk block

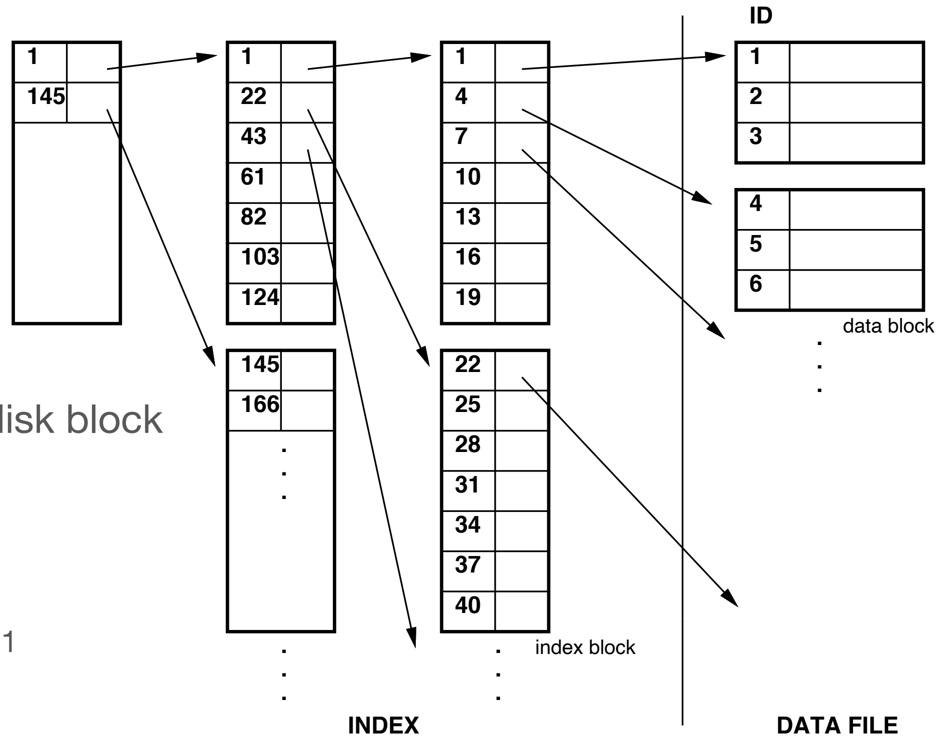
- 50K / 200 = 250 index pages
- Log(250) = 8 + 1 = 9 disk accesses

2 Layer Index ???

Multi-layer Indexes

We can create an index for our index!

Each index layer speeds up search but consumes more space



Sparse index: 1 index record per disk block

- 50K / 200 = 250 index pages
- Log(250) = 8 + 1 = 9 disk accesses

2 Layer Index:

- 50K / 200 = 250 index pages in layer 1
- 250/200 = 2 pages in layer 2
- Log(2) + 1 + 1 = 4 disk accesses

Alternatives

Indexes and sorted files work pretty well, but don't handle updates well

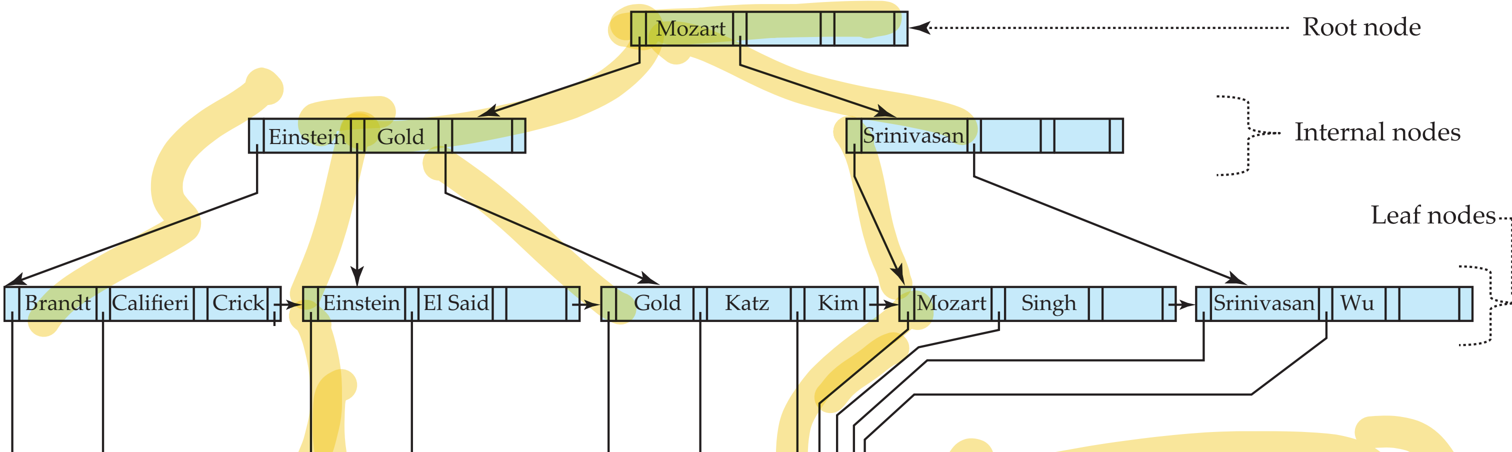
- Performance degrades as files get larger
- May need to reorganize data file and index file

B^+-Trees are data structures customized for database storage and indexing

- Allow efficient searching, including range queries
- Automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
- Reorganization of entire file is never required to maintain performance.

B+-Tree

Efficient, dense, multi-level index



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Takes the concepts we covered today and builds them into a data structure

Indexes in practice

DBMS will allow **you** to create an index on the fields you expect will have the most searches

```
CREATE INDEX idx_lastname
ON Persons (LastName);
```

Now all `WHERE Persons.LastName = "..."` queries will be faster!

- But all updates to Persons will be (slightly) slower

Your project DBs will all fit in memory, so no significant benefit from using indexes...

Summary

Yet one more amazing thing that the DBMS can do for you!

Way better than needing to write your own code to optimize a query or worry about how to layout data on disk yourself!