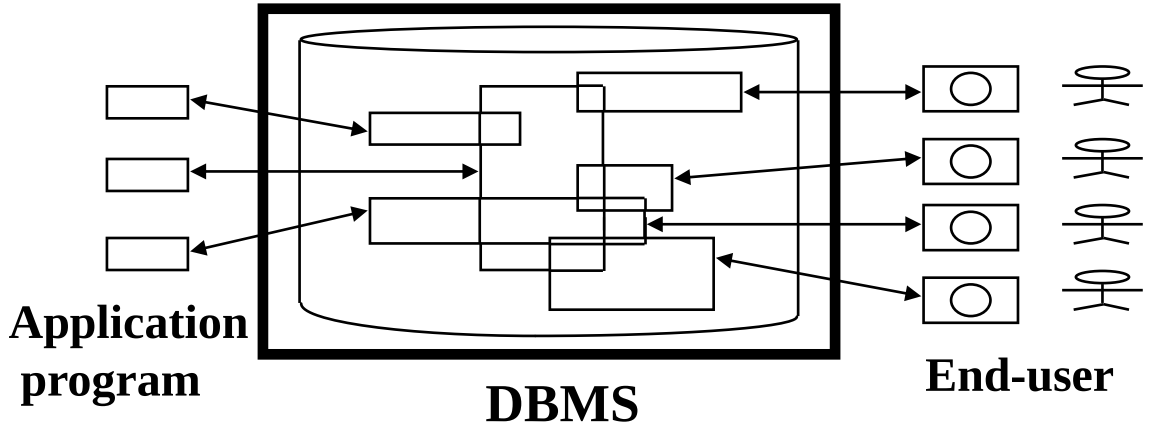
DBMS Indexing

Dr. Shaheen Khatoon

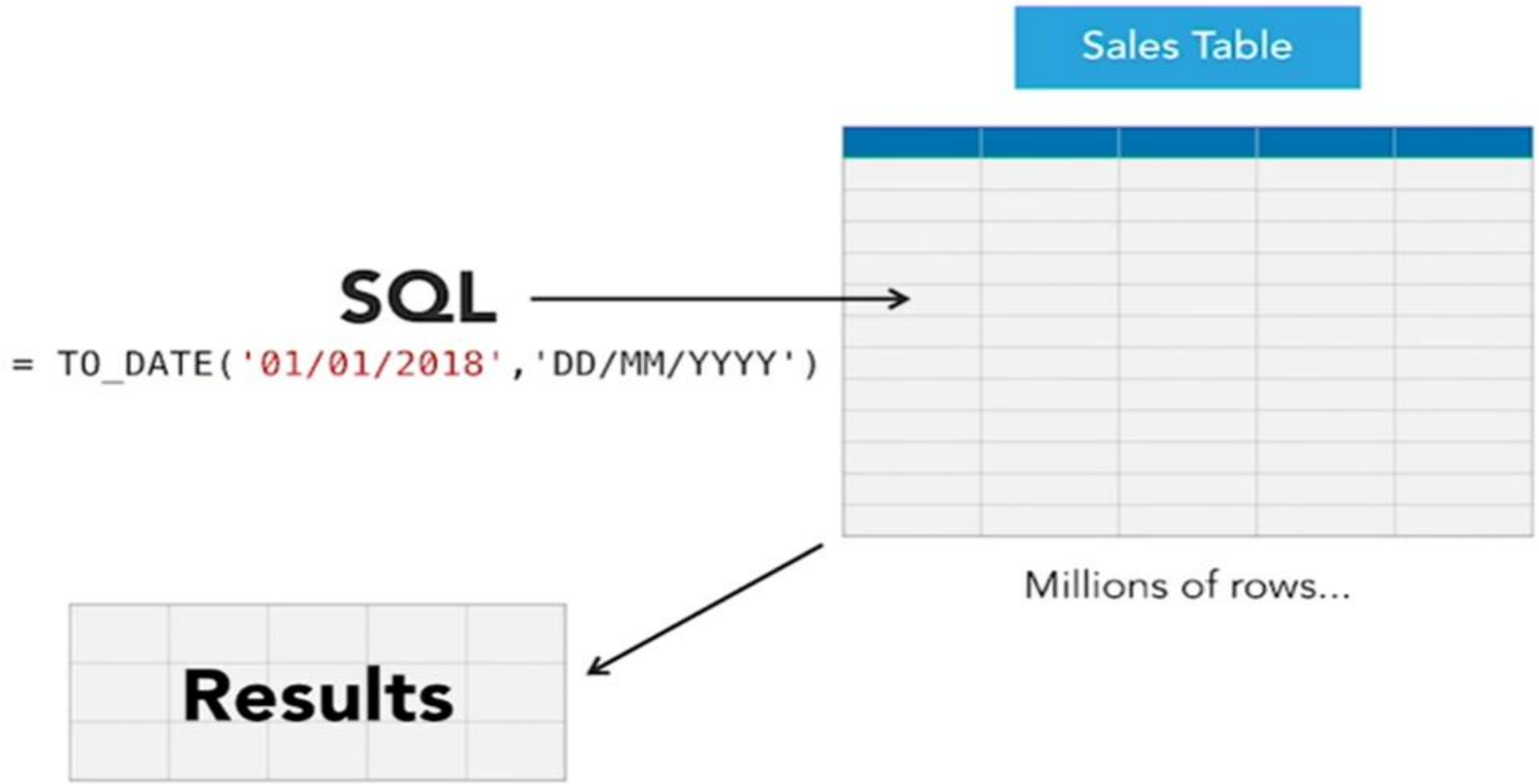


Indexes

- Primary mechanism to get improved performance on a database

- Schema objects created in the database in one or more column of a table
  - Persistent data structure stored in database

Without Index –Table Scan



- Without indexes, the entire table will be scanned to find the matching row (Linear search)
- The search space can be reduced by using indexes

Basic Search Demo

- Linear Search or sequential scan (unordered data file)
  - Average search time b/2, where b is number of blocks required to store a data file
- Binary search
  - If a file is ordered, Average search time $\log_2 b$

https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

Need For Indexing: Speed

Consider searching your hard disk using the Windows SEARCH command.

- Search goes into directory hierarchies.
- Takes about a minute, and there are only a few thousand files.

Assume a fast processor and (even more importantly) a fast hard disk.

- Assume file size to be 5 KB.
- Assume hard disk scan rate of a million files per second.
- Resulting in scan rate of 5 GB per second.

Largest search engine indexes more than 8 billion pages

- At above scan rate 1,600 seconds required to scan ALL pages (Sequential Search).
- This is just for one user!
- No one is going to wait for 26 minutes, not even 26 seconds.

Search Engine	Reported Size	Page Depth
Google	8.1 billion	101K
MSN	5.0 billion	150K
Yahoo	4.2 billion (estimate)	500K
Ask Jeeves	2.5 billion	101K+

Hence, a sequential scan is simply not feasible.

Need For Indexing: Query Complexity

- How many customers do I have in Nanjing?
- How many customers in Nanjing made calls during April?
- How many customers in Nanjing made calls to Xiamen during April?
- How many customers in Nanjing made calls to Xiamen during April using a particular calling package?

Need For Indexing: I/O Bottleneck

- Throwing hardware just speeds up the CPU intensive tasks.

- The problem is of I/O, which does not scales up easily.

- Putting the entire table in RAM is very very expensive.

- Therefore, index!

Indexing Concept

- Purely physical concept, nothing to do with logical model.

- Invisible to the end user (programmer), optimizer chooses it, effects only the speed, not the answer.

- Persistent data structure, stored in database

Indexing Goal

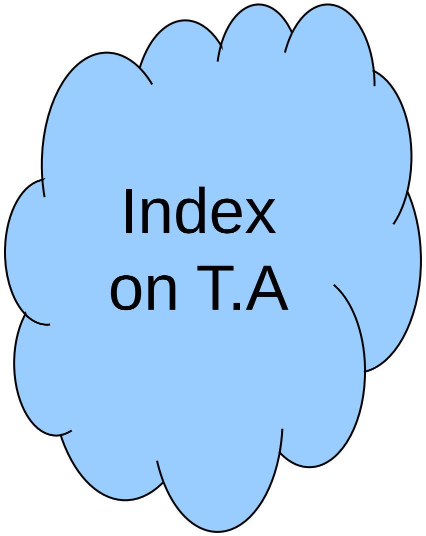
Look at as few blocks as possible to find the matching record(s)

- The point of using an index is to **increase the speed** and **efficiency** of searches of the database.
- Without some sort of index, a user's query must **sequentially** scan the database, finding the records matching the parameters in the WHERE clause.

Functionality

T.A= ‘cow’

T.A=‘cat’



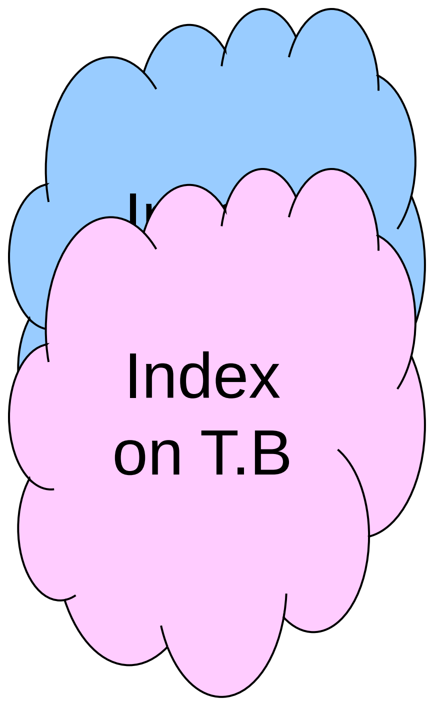
T

	A	B	C
1	cat	2	…
2	dog	5	…
3	cow	1	…
4	dog	9	…
5	cat	2	…
6	cat	8	…
7	cow	6	…
	…	…	…

Functionality

T.B<6

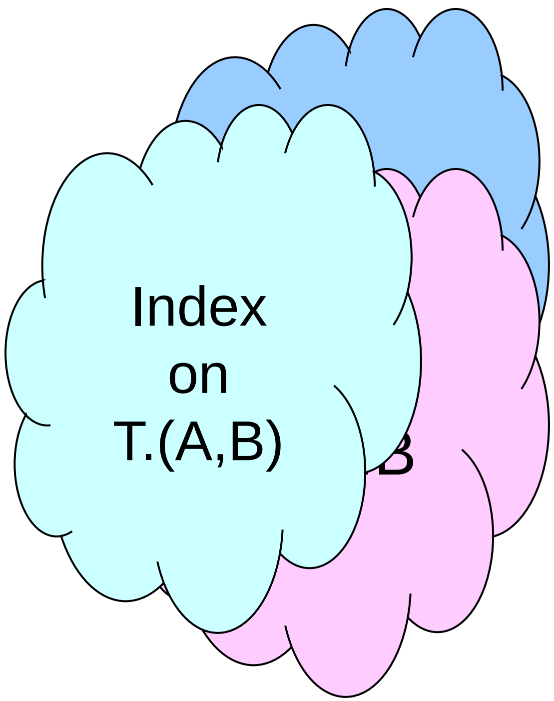
4<T.B<=8



T

	A	B	C
1	cat	2	...
2	dog	5	...
3	cow	1	...
4	dog	9	...
5	cat	2	...
6	cat	8	...
7	cow	6	...
	...	...	...

Functionality



T

	A	B	C
1	cat	2	…
2	dog	5	…
3	cow	1	…
4	dog	9	…
5	cat	2	…
6	cat	8	…
7	cow	6	…
	…	…	…

T.A= ‘cat’
AND T.B>5

SQL Syntax

```
Create Index IndexName on T(A)

Create Index IndexName on T(A1,A2,…,An)

Create Unique Index IndexName on T(A)

Drop Index IndexName
```

Utility

- Index = difference between full table scans and immediate location of tuples
  - Orders of magnitude performance difference
- Underlying data structures
  - Balanced trees (B trees, B+ trees)
  - Hash tables

Conventional indexing Techniques

- Dense
- Sparse

- Primary Index vs. Secondary Indexes
- Multi-level (or B-Tree)

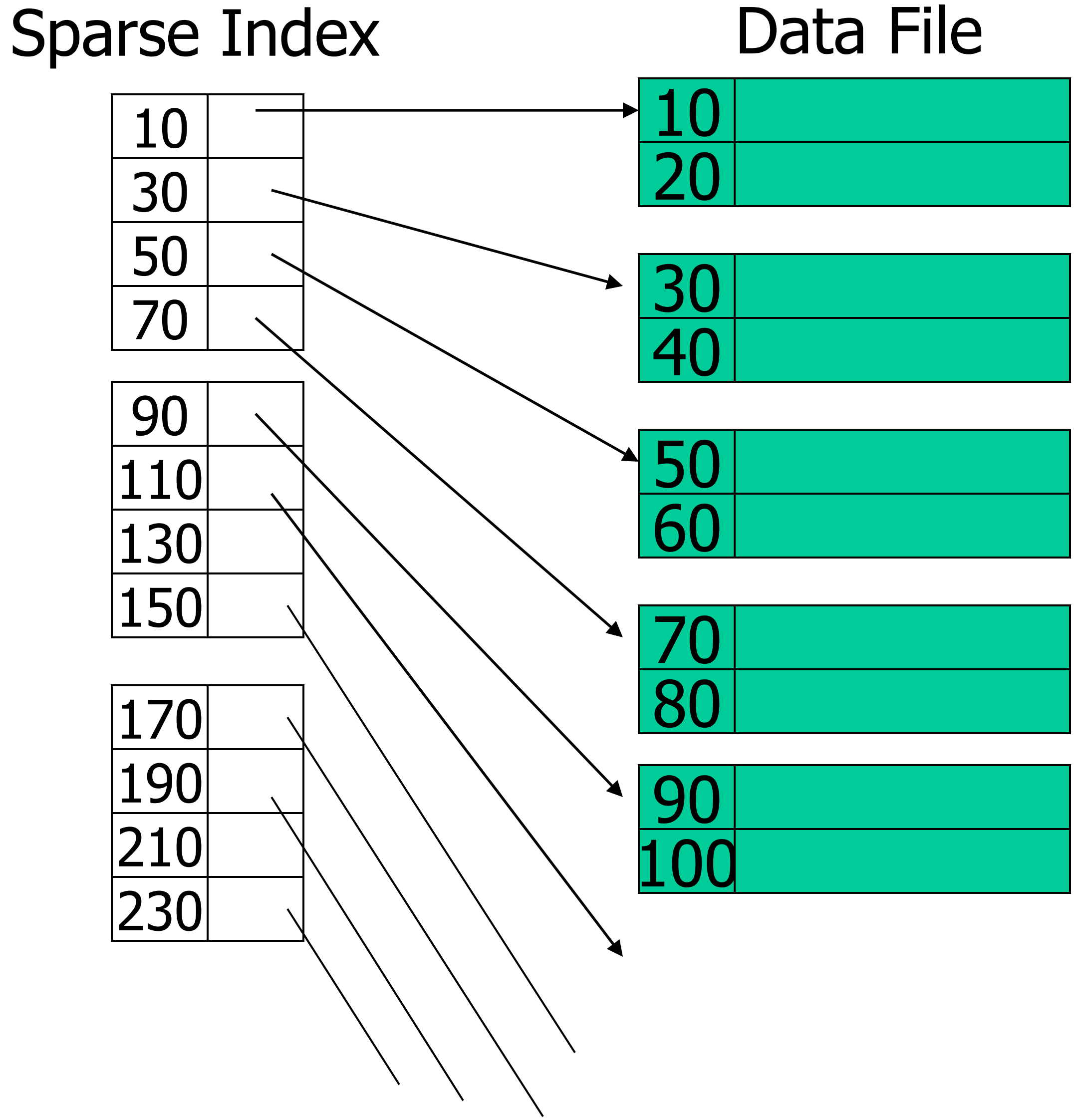
Sparse Index

Index is always ordered

Two entries: Index field and block pointer

Sparse index keeps only one key per data block

Some keys in the data file will not have an entry in the index file



Sparse Index: Adv & Dis Adv

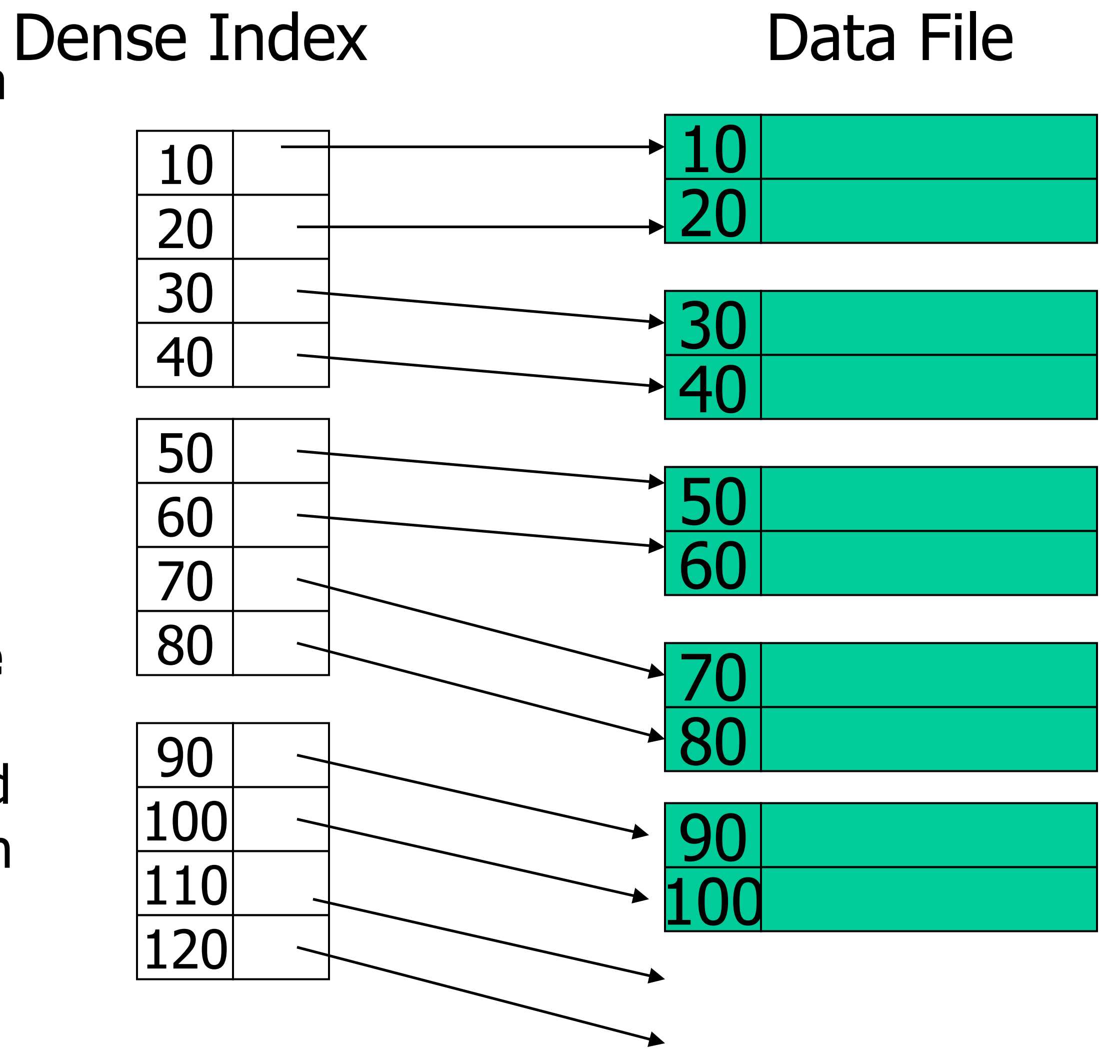
- Advantage:
  - A sparse index uses **less space** at the expense of somewhat more time to find a record given its key
  - **Support multi-level indexing structure**
- Disadvantage:
  - Locating a record given a key has **different performance** for **different key values**

Dense Index

Every key in the data file is represented in the index file

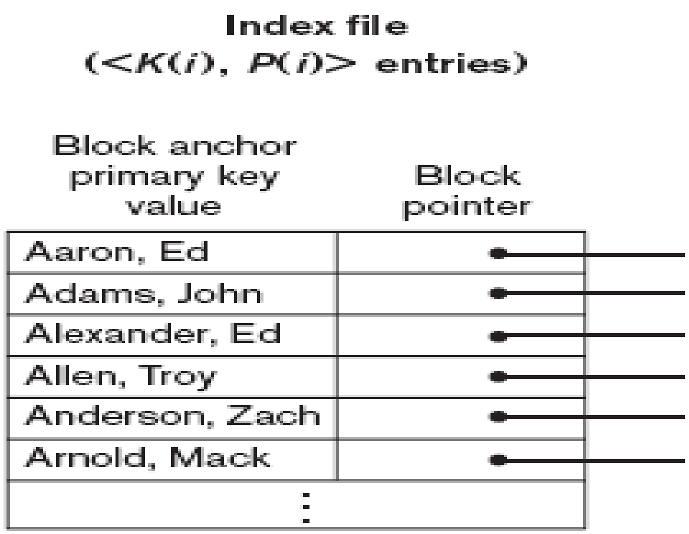
very **efficient** in locating a record given a key

if too big and **doesn't fit into the memory**, will be expensive when used to find a record given its key



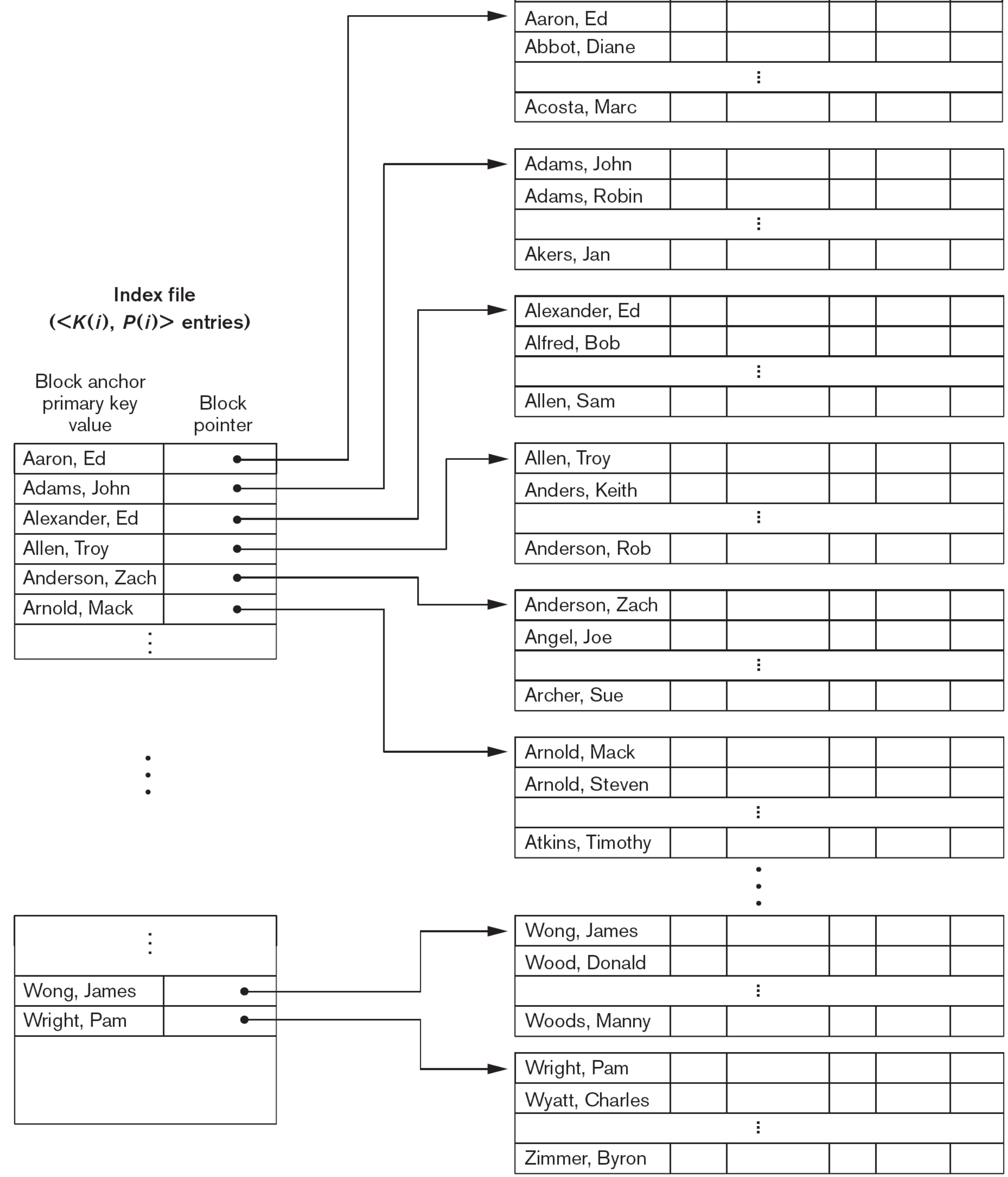
Single Level Index

- A **single-level** index is an auxiliary file that makes it more efficient to search for a record in the data file.
- The index is usually specified on one field of the file (although it could be specified on several fields)
- One form of an index is a file of entries <**field value, pointer to block>**, which is ordered by field value



Primary Index ~~on the Ordering~~ Key Field

Figure 18.1
Primary index on the ordering key field of the file shown in Figure 17.7.



Number of entries in index file = number of block require for primary data file

Example: Record search without index

- Example 1: Given the following data file EMPLOYEE(NAME, SSN, ADDRESS, JOB, SAL, ...) Suppose that:
  - record size R=100 bytes block size B=1024 bytes r=30000 records
- Then, we get:
  - blocking factor bfr=floor(B/R)= floor(1024/100)=10 records/block
  - number of file blocks needed for data file b= ceiling (r/Bfr)= (30000/10)= 3000 blocks
  - Average linear search cost:(b/2)= 3000/2= 1500 block accesses
  - If the file records are ordered, the binary search cost would be:
    - $\log_2 b = \log_2 3000 =$ 12 block accesses

Example: Record search with Index

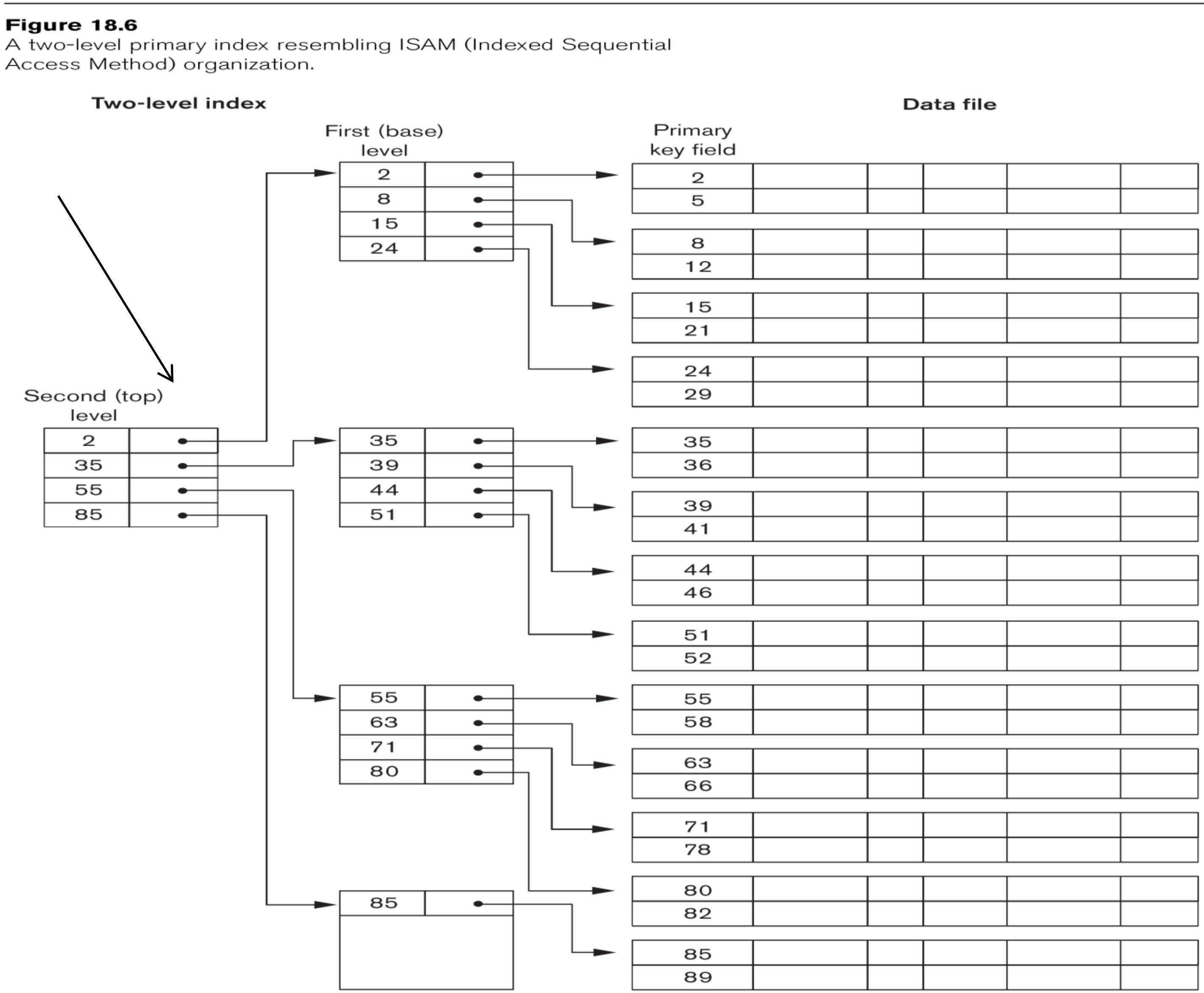
- For an index on the SSN field, assume the field size V_{SSN}=9 bytes, assume the record pointer size P_R=6 bytes. Then:
  - index entry size $R_I=(V_{SSN}+P_R)=(9+6)=15$ bytes
  - index blocking factor $Bfr_I= B/R_I= 1024/15= 68$ entries/block
  - number of block to store index $b= (r/ Bfr_I)= (3000/68)= 45$ blocks
  - binary search needs $\log_2 b_I= \log_2 45= 6$ block accesses
  - we need one additional block access to the data file so 6+1=7
  - An improvement on binary search of data file which require 12 blocks

Multi-Level Indexes

- Binary search in a single level index require a search time of order of $\mathbf{log_2 b}$ number of block access
  - Here *b* is the number of blocks
  - If **bfr** (blocking factor) of index file is greater than 2, number of block access can be even **reduce further**
- Multiple level indexes are meant for such reduction
  - Contain several level of index files
  - Each index at given level connect to maximum number of **FO** (fan out) number of block at next level
  - Block access reduce from $\mathbf{log_2 b}$ **to** $\mathbf{log_{fo} b}$ on an average for FO>2

A Two-Level Primary Index

Value >=2<35
is found index
entry 2



Multi level index

- Far ordered index file discussed so far.
  - A **binary search** is applied to the index to locate pointers to a disk block.
  - It requires **($\log_2 b_i$) block accesses** for an index with b_i blocks
    - because each step of the algorithm **reduces** the index file to search by a factor of **2**.
- **Multilevel index** is to reduce the part of the index that to search by $\boldsymbol{bfr_i}$, the blocking factor for the index, which is larger than 2.
  - Search space reduced much faster.
  - The value *bfri* is called the **fan-out (*fo*)** of the multilevel index. Where it divide the *record search space* into *n*-ways (where *n* = the ***fo*** at each search step using the multilevel index.
- Searching a multilevel index requires approximately **($\log_{fo} b_i$) block accesses,** smaller number than for a binary search if the fan-out is larger than 2.

Multi level index structure

- **First** (base) level is usually ***primary index*** that is maintained in a sorted file
- *Second* (top) level is a primary index at first level index file
- We can repeat the process, creating **a third, fourth, ..., top level** until all entries of the *top level* fit in one disk block
- Each level *reduce the number of entries of its next level* by factor of *fo*
- *If* $bfr_i = fo$, entries needed at each level:
  - First, level needs $\lceil (r_1/fo) \rceil$ blocks, which is therefore the number of entries r_2 needed at the second level of the index:
    - $r_2 = \lceil (r_1/fo) \rceil$ number of block
    - $r_3 = \lceil (r_2/fo) \rceil$ …..
- We can repeat the preceding process until all the entries of some index level *t* fit in a **single block**. This block at the *t*th level is called the **top** index level**.**
- **Single disk block** is retrieved **at each level**, t disk blocks are accessed for an index search, where t is the number of index levels.

Example of multi-level index

- Suppose we have a data file with r=300,000 records and each record length (R) 100 byte and block size (B) 4096 bytes
- We first need to identify how many disk block is required to store the data file by calculating blocking factor for the data file
- The blocking factor for the file would be bfr = $\lfloor(B/R)\rfloor = \lfloor(4{,}096/100)\rfloor = 40$ records per block.
- The number of blocks needed for the file is b = $\lceil(r/bfr)\rceil = \lceil(300{,}000/40)\rceil = 7{,}500$ blocks.
- If data file is unorder a linear search on the data file would need approximately b/2=3750 block access
- If data file is ordered on some column, A binary search on the data file would need approximately $\lceil \log 2\ b\rceil = \lceil(\log 2\ 7{,}500)\rceil = 13$ block accesses

Example of multi-level index (cont…)

- Now suppose that the ordering key field of the file is V = 9 bytes long, a block pointer is P = 6 bytes long, and we have constructed a primary index for the file.
- The size of each index entry is Ri = (9 + 6) = 15 bytes, so the blocking factor for the index is bfri = $\lfloor(B/Ri)\rfloor = \lfloor(4{,}096/15)\rfloor$ = 273 entries per block.
- The total number of index entries ri is equal to the number of blocks in the data file, which is 7,500. The number of index blocks is bi = $\lceil(ri/bfri)\rceil = \lceil(7{,}500/273)\rceil$ = 28 blocks.
- To perform a binary search on the index file would need $\lceil(\log 2\ bi)\rceil = \lceil(\log 2\ 28)\rceil$ = 5 block accesses.
- To search for a record using the index, we need one additional block access to the data file for a total of 5 + 1 = 6 block accesses—an improvement over binary search on the data file, which required 13 disk block accesses.

Example of multi-level index (cont…)

- If we want to build a second level index on 28 blocks of first level index, we need
- First level index entries =7500 and number of blocks (b1) =7500/273 = 28 block
- Second level number of index entries = 28 and number of block(b2)= 28/273= less than 1 block
- So in this case we can have maximum 2 level index with number of index blocks = b1 + b2=28 +1 = 29 blocks to store 2 level index
- To access a record by searching the multilevel index, we must access one block at each level plus one block from the data file, so we need 2 + 1 **= 3 block accesses.**
- Compare this to single-level index with 6 block access with the single level index

Homework

- Consider a disk with block size B=512 bytes, and a record pointer is PR =7 bytes long. A file has r=30,000 EMPLOYEE records of fixed-length. Each record has the following fields: NAME (30 bytes), SSN (9bytes), DEPARTMENTCODE (9 bytes), ADDRESS (40 bytes), PHONE (9 bytes),BIRTHDATE (8 bytes), SEX (1 byte), JOBCODE (4 bytes), SALARY (4 bytes, realnumber). An additional byte is used as a deletion marker.

- Calculate:

1. What would be number of blocks required to store this data file assuming an unspanned file organization?
2. On average how many blocks are required to search for a record if the data file is ordered on SSN as key field? Compare with the search performance of unordered file organization.
3. Suppose the file is ordered by the key field SSN and we have primary index on SSN. On average, how many blocks are required to access to search a record with this primary index on place?

Homework (cont…)

3. Suppose we want to construct multi-level index. Calculate:

   a) How many index entries and number of bloack required at each levels needed to make it multilevel index?

   b) How many blocks required to store the multi-level index, and how many blocks need to search a record with multi-level index?

Multi level index structure

- **Search,** A multilevel index reduces the number of blocks accessed when searching for a record, given its indexing field value.

- **Insertions and deletions**, cause severe problem because all index levels are ***physically ordered files***.
  - The updating overhead of multi-level index is high because we have to reorganize all index files every time there is a delete or insert operation in the database.

- A multilevel index called a **dynamic multilevel index** that leaves some space in each of its blocks for inserting new entries and uses appropriate insertion/deletion algorithms for creating and deleting new index blocks when the data file grows and shrinks.

- It is implemented by using data structures called **B-trees and B+-trees,**

Summary

- Indexes
- Single Level Indexes
- Multilevel Indexes