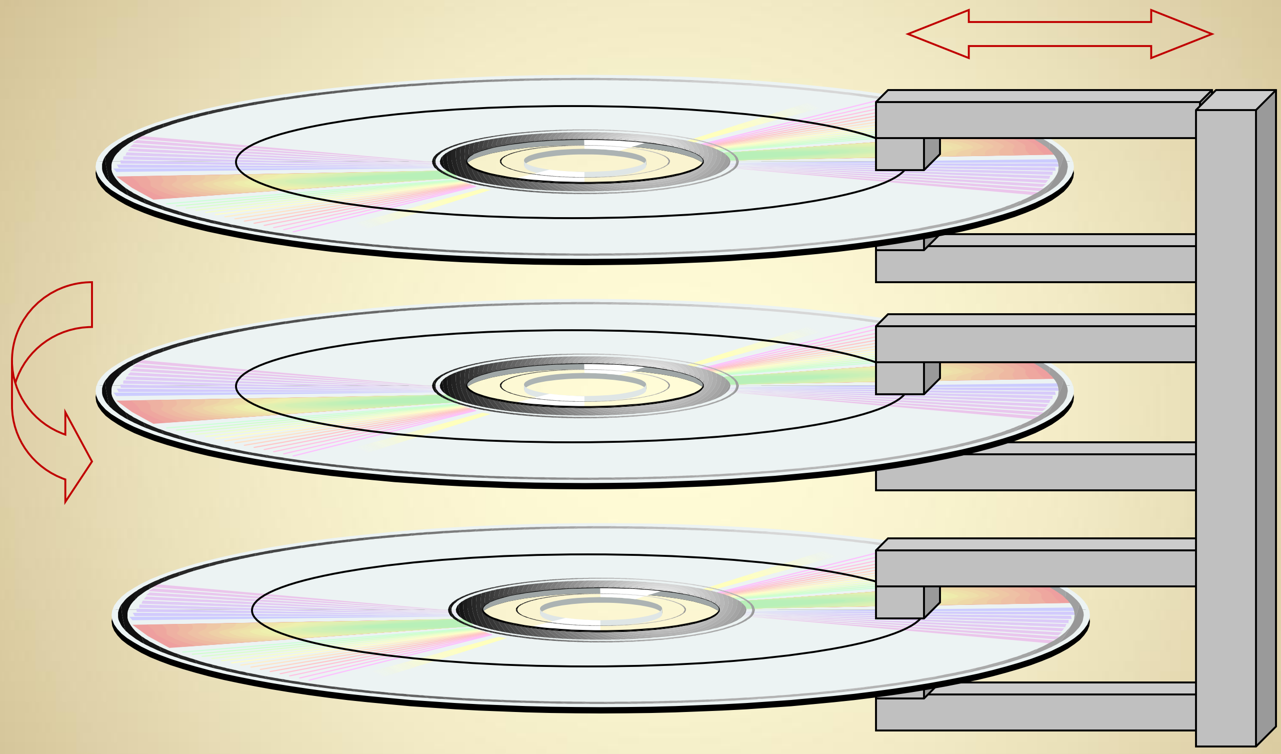
Advanced Database Management Systems

Lecture 15

Index Structures: Sections 14.1, 14.2

Review: Disks

Review: Hard Drives

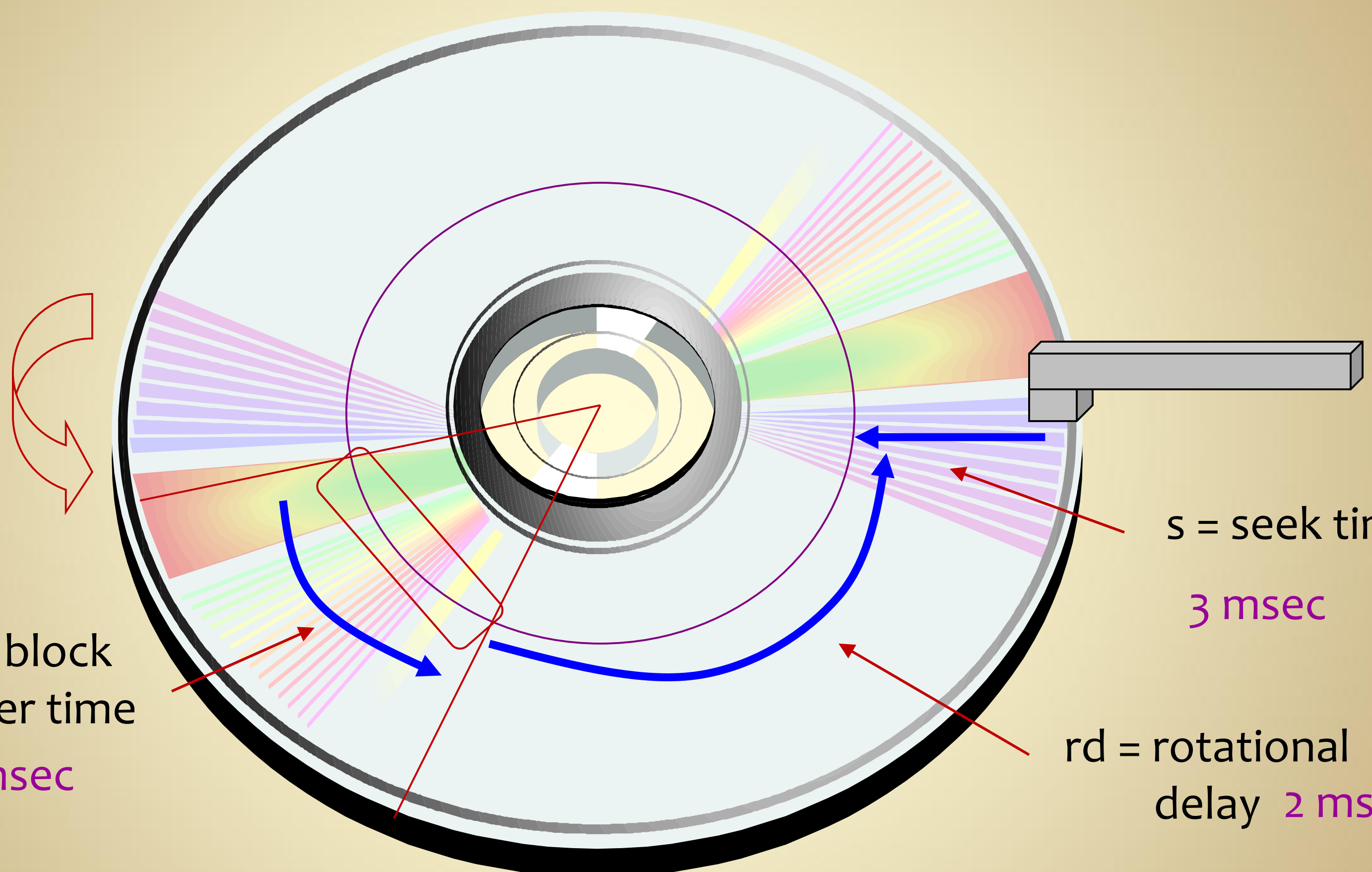


What are the primary factors in computing the time for data transfer?

Disk Parameters

parameter		typical value	source
s	seek time	3 msec	fixed
p	rotational velocity	10,000 rpm 167 rps	fixed
rd	rotational delay (latency)	2 msec	.5*(1/p) (average)
T	track size	50 Kbytes	fixed
B	block size	512-4096 bytes	formatted
G	interblock gap size	128 bytes	formatted
tr	transfer rate	800Kbytes/sec	T*p
btt	block transfer time	1 msec	B/tr
btr	bulk transfer rate (consecutive blocks)	700Kbytes/sec	(B/(B+G))*tr

Random Block Transfer Time



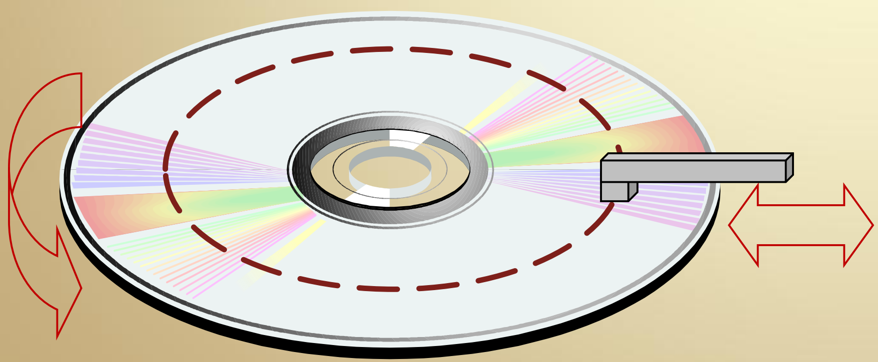
rbtt = time to locate and transfer one random block
= s + rd + btt (3+2+0.1 = 5.1msec)

Transferring Multiple Blocks

Time to transfer n blocks of data:

randomly located blocks:

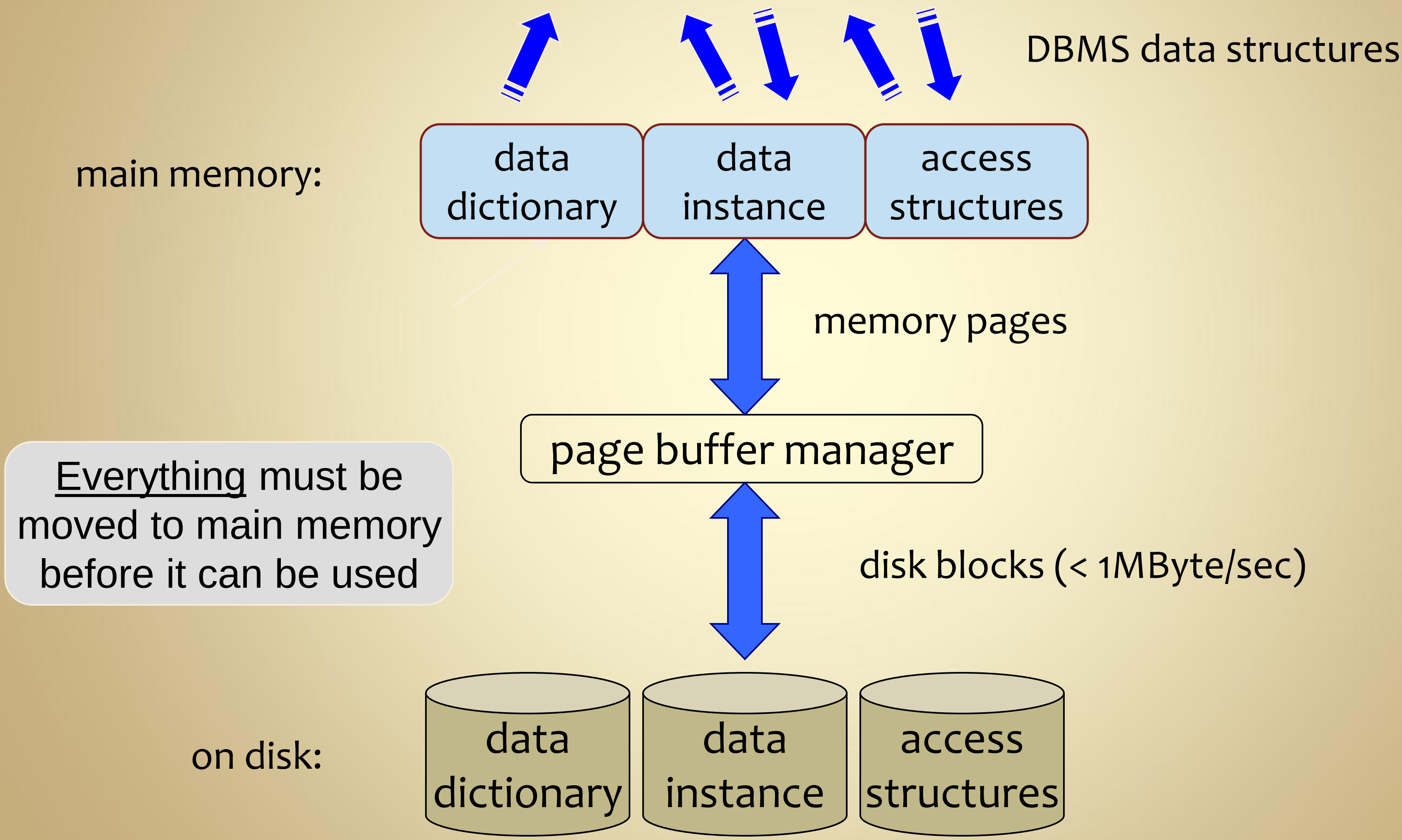
rbtt = n*(s + rd + btt)



consecutively located blocks:

cbtt = s + rd + n*btt

Page Buffer Management



Fundamental Results

- **Organize data in blocks**
  - a block is the basic unit of transfer
- **Layout data to maximize possibility of consecutive block retrieval**
  - avoid seek time and latency
- This will impact
  - record layout in files
  - access structure (indexes, trees) organization

Review: Files

Blocks, Records and Files

- A ***record*** is one unit of structured data
  - **tuple** in a relation
  - **node** in a tree or index
  - object or other structure
- Records may be *fixed length* or *variable length*
- A ***file*** is a set of records stored as a unit on disk
  - a file is stored on some set of disk blocks

File Organization Options

- Contiguous vs. non-contiguous
  - Is the file stored in contiguous blocks, or scattered around the disk
- Ordered vs. non-ordered
  - Are the records ordered on some ordering field?
- Spanning vs. non-spanning
  - Can a record be split across blocks, or is every record contained within one block?

Blocking Factors

- *Blocking factor* determines how many records can fit in a block
  - bfr = $\lfloor B / R \rfloor$ where B = block size and R = record size
- *Number of blocks* required is determined by the blocking factor and the number of records
  - b = $\lceil r / bfr \rceil$ where r = number of records

File Structure

- Some navigational structure is needed to identify/locate all blocks that comprise a file
  - Similar to dynamic memory management

- Possible structures:
  - ***contiguous***: Once you find the first block, keep reading blocks in sequence until the end
  - ***linked***: Each block has a pointer to the next block
  - ***indexed***: an access structure (index or tree) holds block pointers to all blocks in the file

Unordered vs. Ordered Files

- Unordered
  - insert: new records placed at the end of file
    - O(1)
  - find/delete: linear search
    - O(n)
- Ordered
  - insert: requires reorganization
    - O(n)
  - find/delete on ordering field: binary search
    - O(log n)
  - find/delete on other field: linear search
    - O(n)

Ordered File Example

	NAME	SSN	BIRTHDATE	JOB	SALARY	SEX
block 1	Aaron, Ed					
	Abbott, Diane					
	⋮					
	Acosta, Marc					
block 2	Adams, John					
	Adams, Robin					
	⋮					
	Akers, Jan					
block 3	Alexander, Ed					
	Alfred, Bob					
	⋮					
	Allen, Sam					
block 4	Allen, Troy					
	Anders, Keith					
	⋮					
	Anderson, Rob					
block 5	Anderson, Zach					
	Angeli, Joe					
	⋮					
	Archer, Sue					
block 6	Arnold, Mack					
	Arnold, Steven					
	⋮					
	Atkins, Timothy					
⋮						
block n –1	Wong, James					
	Wood, Donald					
	⋮					
	Woods, Manny					
block n	Wright, Pam					
	Wyatt, Charles					
	⋮					
	Zimmer, Byron					

Disk Access Costs

rbtt = random block transfer time = s + rd + btt (bad)
cbtt = consecutive block transfer time = btt (good)

	insert	find/delete
unordered non-contiguous	O(1)*rbtt	O(n)*rbtt
unordered contiguous	O(1)*rbtt	O(n)*cbtt
ordered non-contiguous	O(n)*cbtt	O(log n)*rbtt

assumes we don't reorganize file, simply mark the record as deleted

Access Stuctures

Access Structures

- Access structures are auxiliary data structures constructed to assist in locating records in a file

- Benefit of access structures
  - more efficient data access

- Cost of access structures
  - storage space:
    access structures must persist along with the data
  - update time:
    access structures must be modified when file is modified

Index Blocking

- Access structures also have to be blocked
  - records are index nodes

- *Blocking factor*: nodes per block
  - $bfr_i = \lfloor B/R_i \rfloor$ where B = block size and R_i = node size

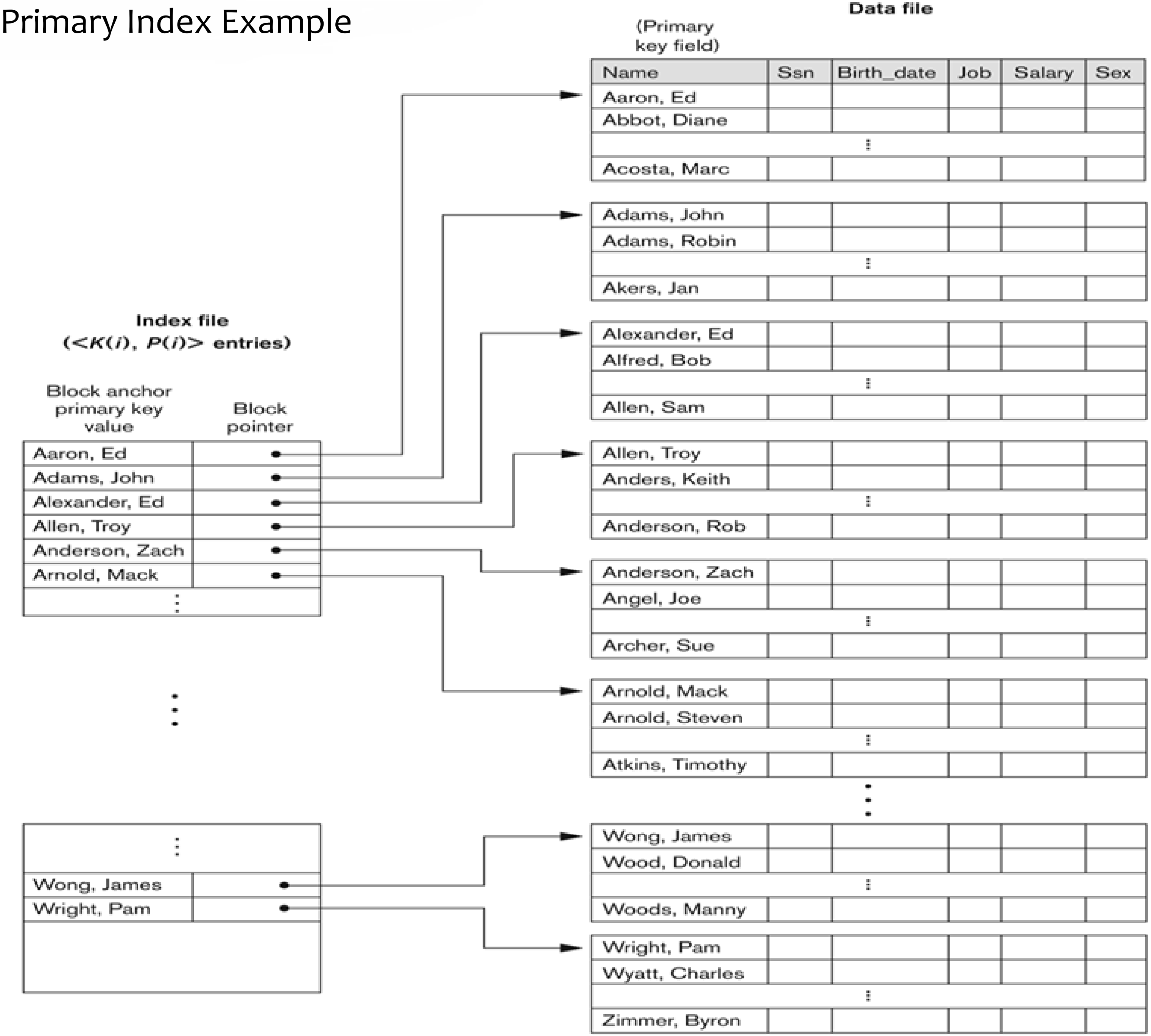
- Blocks required to store index
  - $b_i = \lceil r_i / bfr_i \rceil$ where r_i = number of nodes

Single-Level Indexes

Primary Index

- Primary index:
  - fixed length records, non-spanning
  - each record has two fields: < key, block pointer >
  - index file is ordered on key
  - one index record for each block in data file
  - key must be the ordering field of the data file
  - only hold key values for first record (anchor) of each data file block

Primary Index Example



Primary Indexes

- Advantage of a Primary Index
  - much smaller than data file
  - index records (usually) smaller than data records
  - number of index records smaller than number of data records
- Lookup algorithm:
  - binary search on index, then read data block
  - cost with index: $\log_2(r_i) + 1$, r_i = # index blocks
  - cost without index: $\log_2(r)$, r = # data blocks

Primary Index Example

Data File

data records	r = 200,000 records
record size	R = 90 bytes
block size	B = 1024 bytes
key size	V = 9 bytes
block pointer size	P = 6 bytes

data file blocking factor	bfr = $\lfloor B/R \rfloor$ = 11 records/block
data file blocks	b = $\lceil r / bfr \rceil$ = 18,182 blocks

binary search cost without index (worse case):
$\log_2(b) = \log_2(18{,}182) = 15$

Primary Index Example

Index File

index records	$r_i = b = 18{,}182$ records
block size	$B = 1024$ bytes
key size	$V = 9$ bytes
block pointer size	$P = 6$ bytes
index record size	$R_i = V + P = 15$ bytes

index file blocking factor	$bfr_i = \lfloor B / R_i \rfloor = 68$ records / block
index file blocks	$b_i = \lceil r_i / bfr_i \rceil = 268$ blocks

binary search cost with index (worse case):

$\log_2(b) + 1 = \log_2(268) + 1 = 10$

Index Classifications

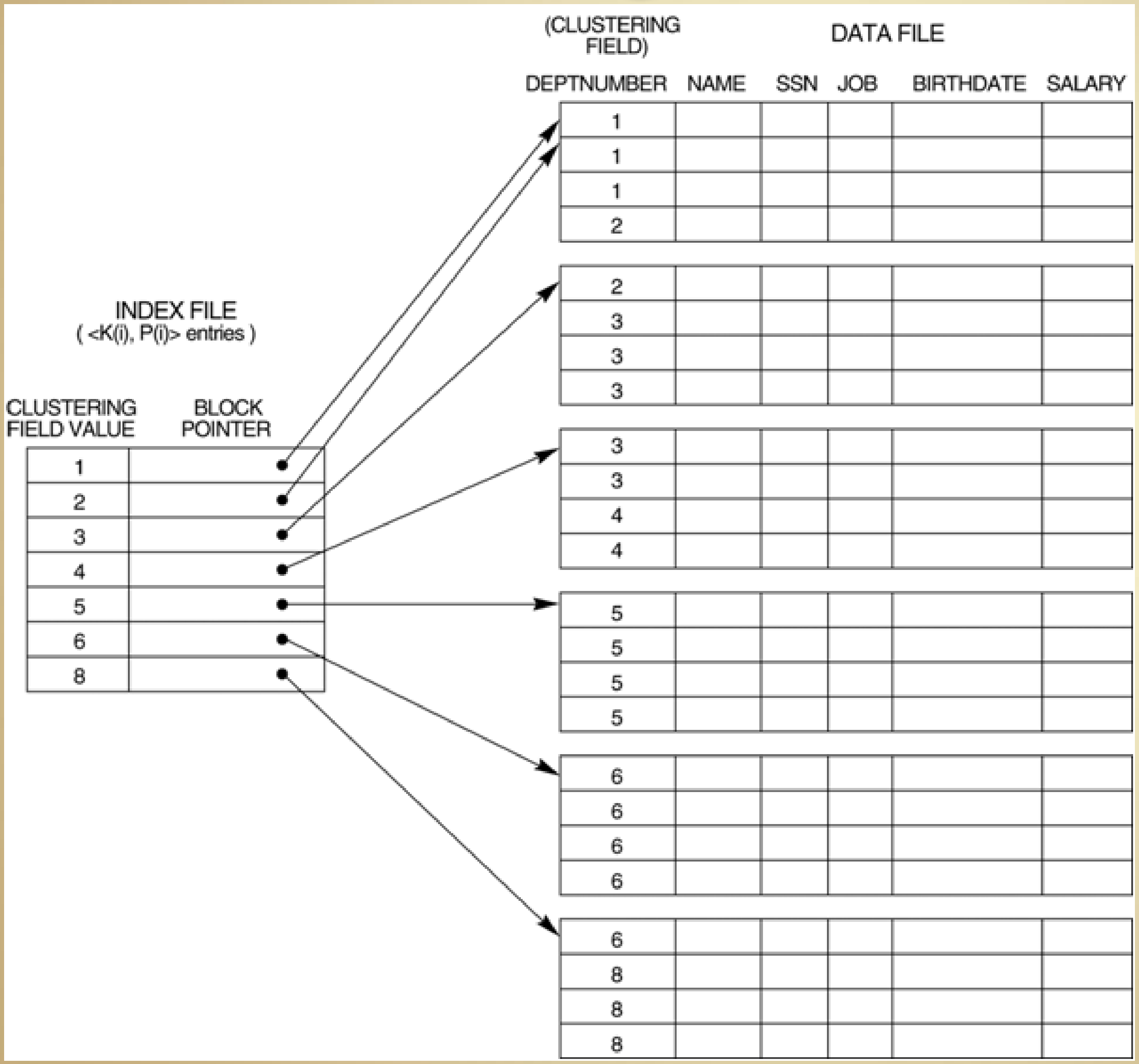
- **dense index:**
  index entry for every data record
- **sparse index:**
  index entries for subset of data records
  (block anchors)

- **primary index:** key, ordering field
  - sparse
- **clustering index:** non-key, ordering field
  - sparse
- **secondary index:** non-ordering field
  - dense or sparce

Clustering Index

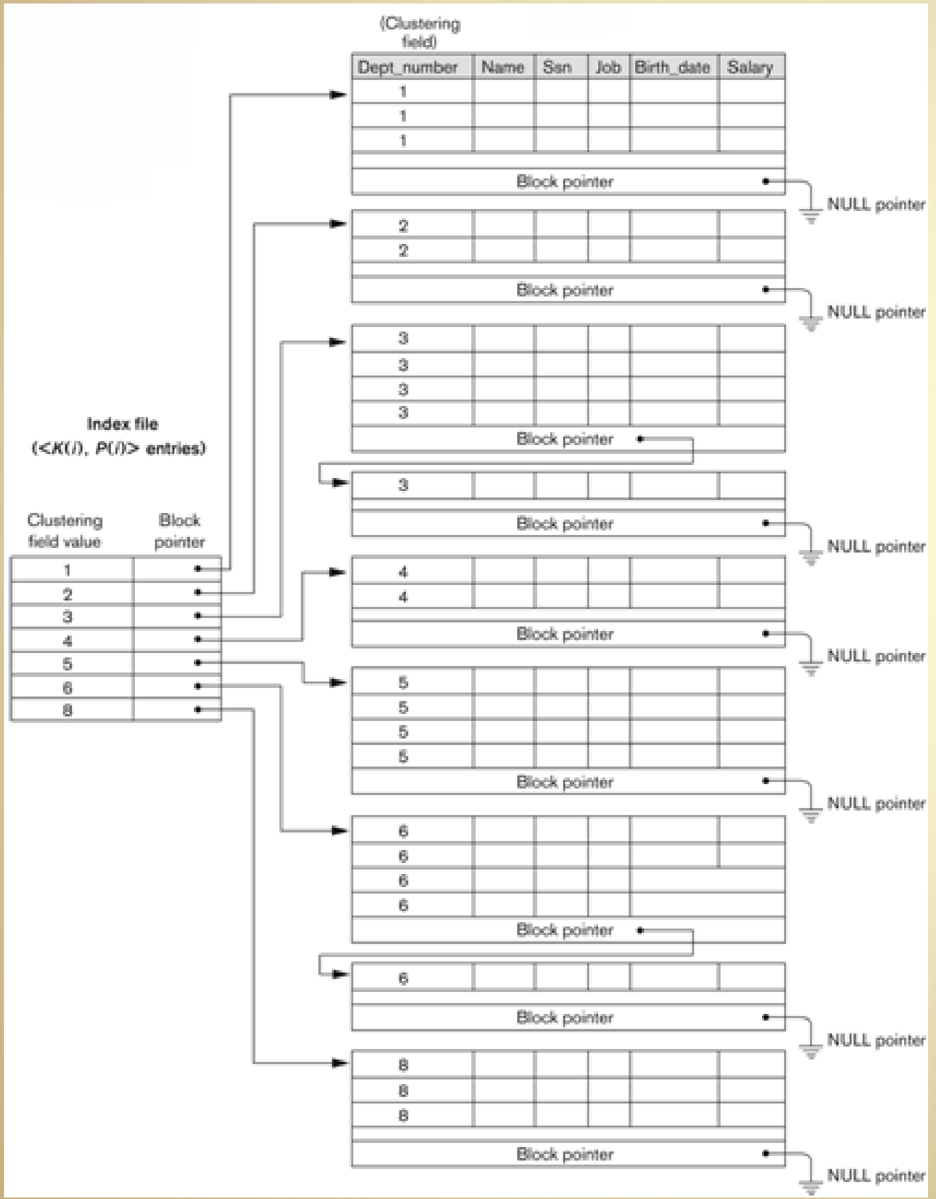
- Data file is ordered on a non-key (non-unique) field
- One index entry *for each distinct value* of the field
  - index entry points to the first data block that contains records with that field value.

Clustering Index Example



assumes contiguous storage of data file

Clustering Index Example



non-contiguous storage of data file

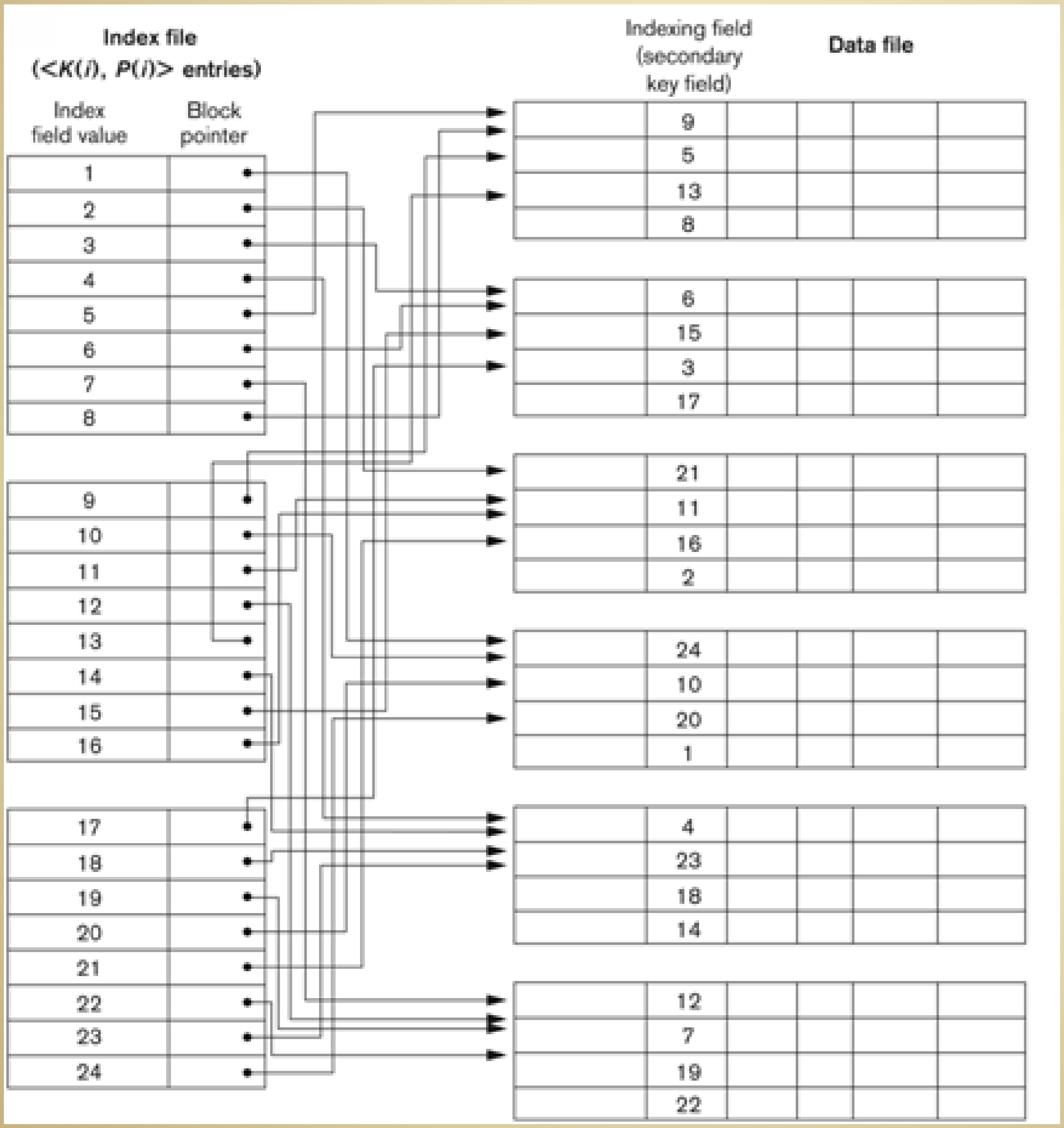
blocks are linked by block pointers

all records in same data block have same clustering field value

Secondary Indexes

- Index on a non-ordering field
  - must be a dense index: one entry for every data record
  - indexing field may be key or non-key
- There may be multiple secondary indexes for the same data file
- Advantage of a secondary index:
  - provides a structure in which index field is ordered, thus allowing binary search

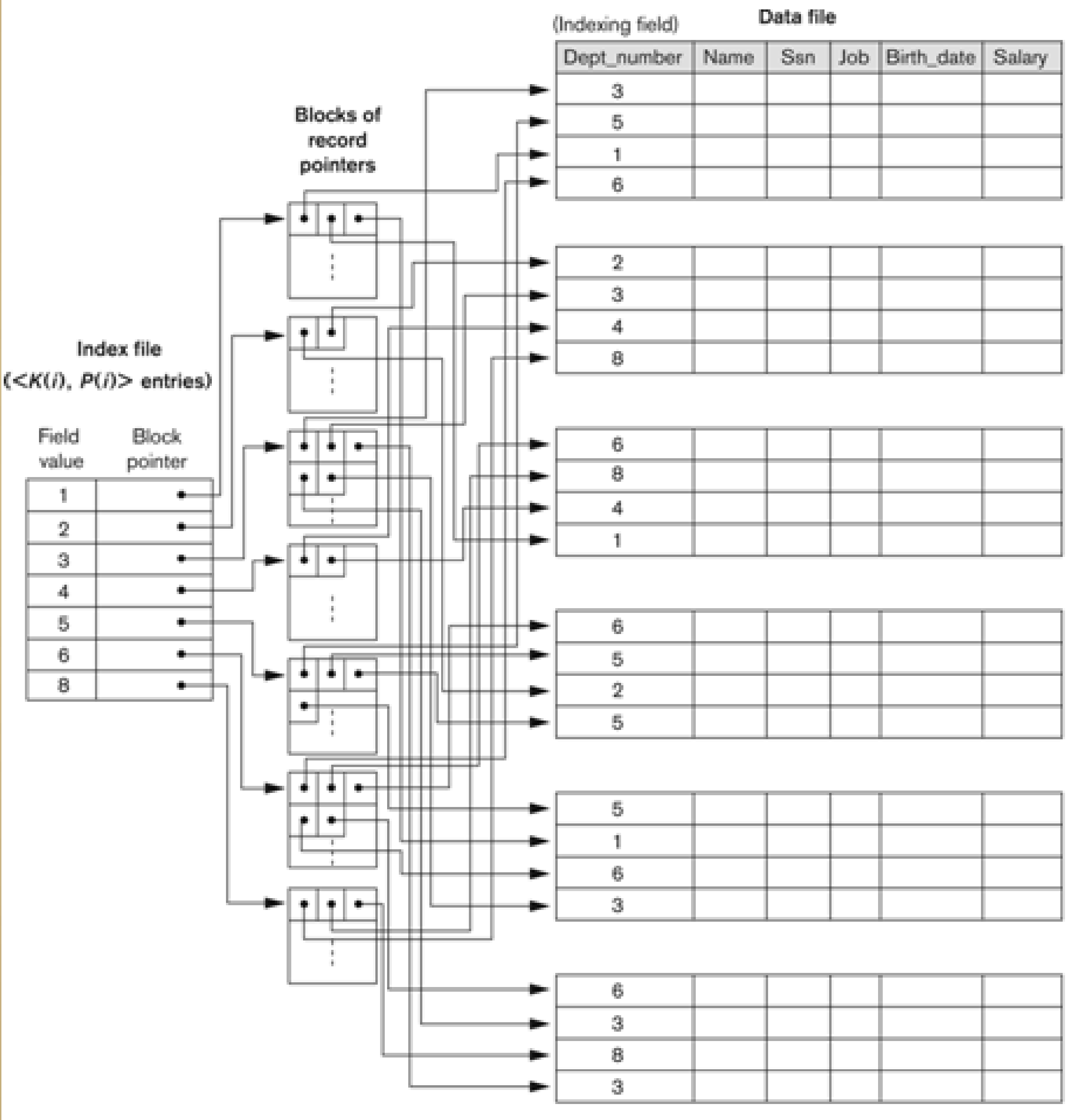
Secondary Index Example



key, non-ordering index field

binary search on index gives data block of target record

Secondary Index Example



non-key,
non-ordering
index field

may not be effective when a small number of index values are evenly distributed among data blocks

Properties of Index Types

TABLE 14.2 PROPERTIES OF INDEX TYPES

TYPE OF INDEX	NUMBER OF (FIRST-LEVEL) INDEX ENTRIES	DENSE OR NONDENSE	BLOCK ANCHORING ON THE DATA FILE
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no[a]
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records[b] or Number of distinct index field values[c]	Dense or Nondense	No

[a]Yes if every distinct value of the ordering field starts a new block; no otherwise.

[b]For option 1.

[c]For options 2 and 3.

Multi-level Indexes

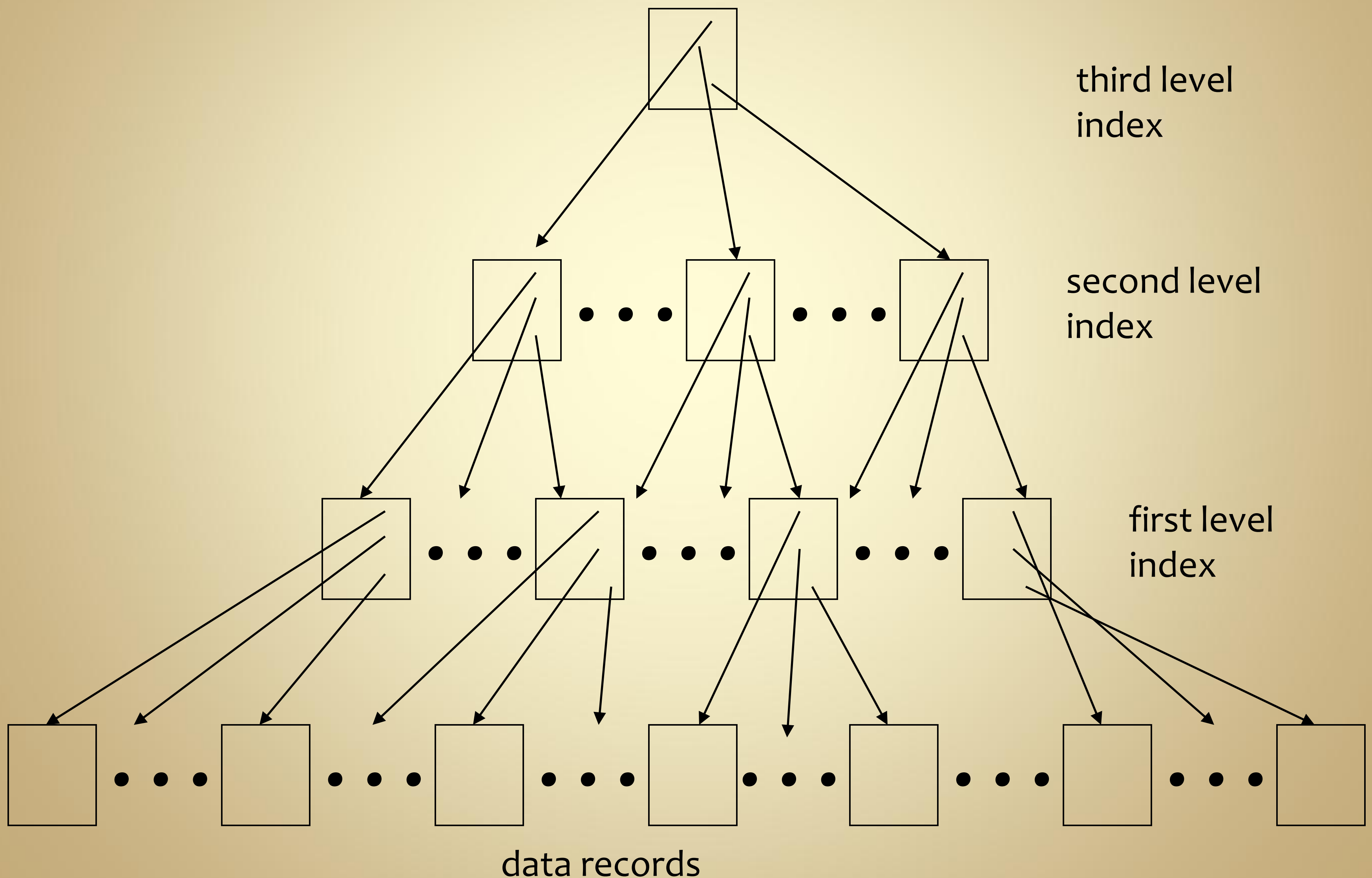
Multi-level Primary Indexes

- multi-level index:
  a tree built by indexing the indexes

- tree arity (fan-out) is the index blocking factor
- tree height is $\log_{fo}(b)$

- Example: b = 20,000 data blocks, bfr_i = fo = 60
  - first level index: $b_1 = \lceil 20000/60 \rceil$ = 334 blocks
  - second level index: $b_2 = \lceil 334/60 \rceil$ = 6 blocks
  - third level index: $b_3 = \lceil 6/60 \rceil$ = 1 block
  - lookup cost:
    one index block read per level + one data block read
    = 4 (random) block reads

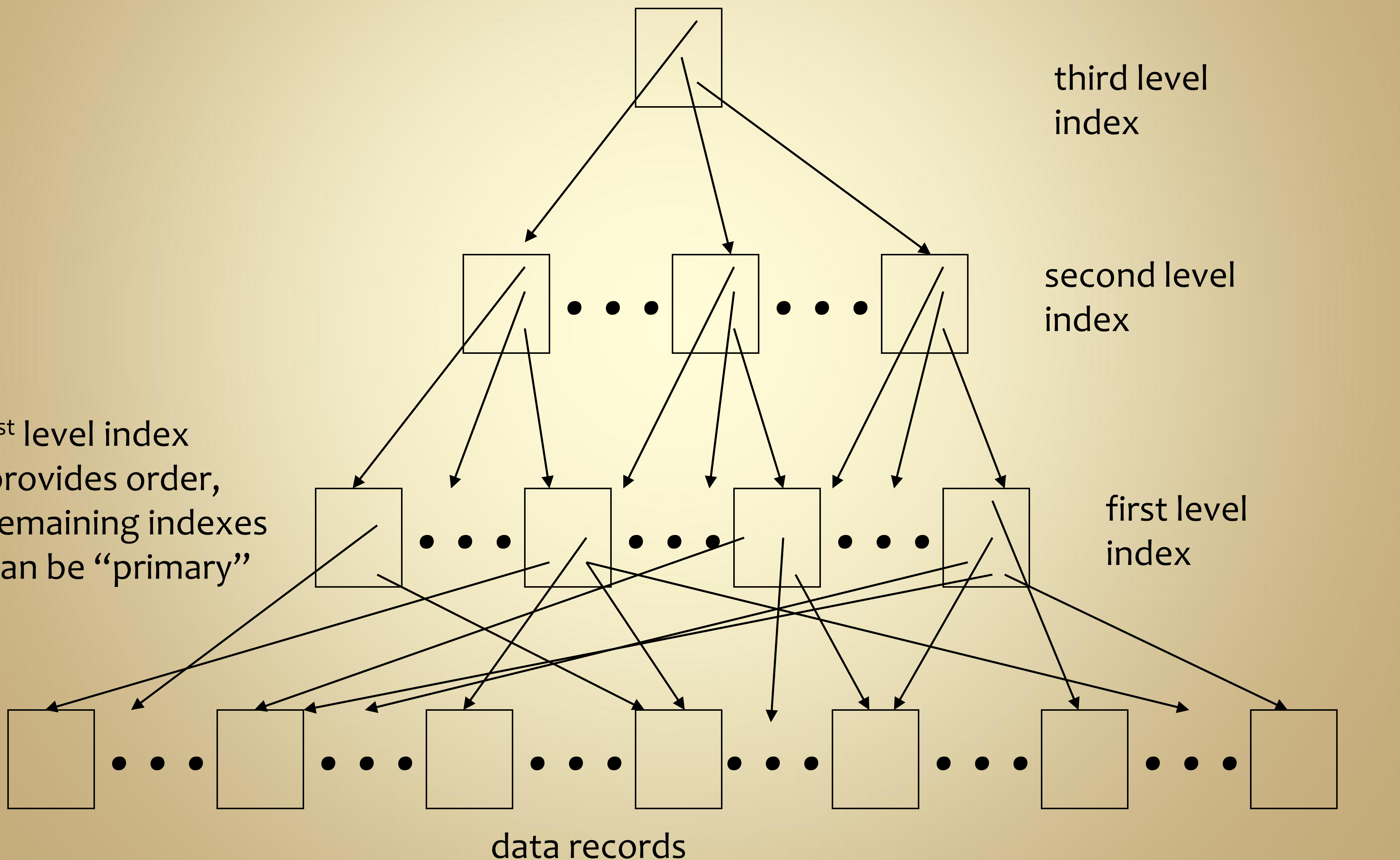
Multi-level Primary Index



Multilevel Secondary Indexes

- Since the first level index is ordered, it can be indexed in the same manner as primary key indexes

Multi-level Secondary Index



1st level index provides order, remaining indexes can be "primary"

Static vs. Dynamic Indexes

- Indexes seen so far are static
  - built from a fixed data file
- Updates to data file require rebuilding the index
  - could be very expensive

- Solution: Dynamic Indexes
  - B-trees and B^+-trees can be adjusted as the data file changes