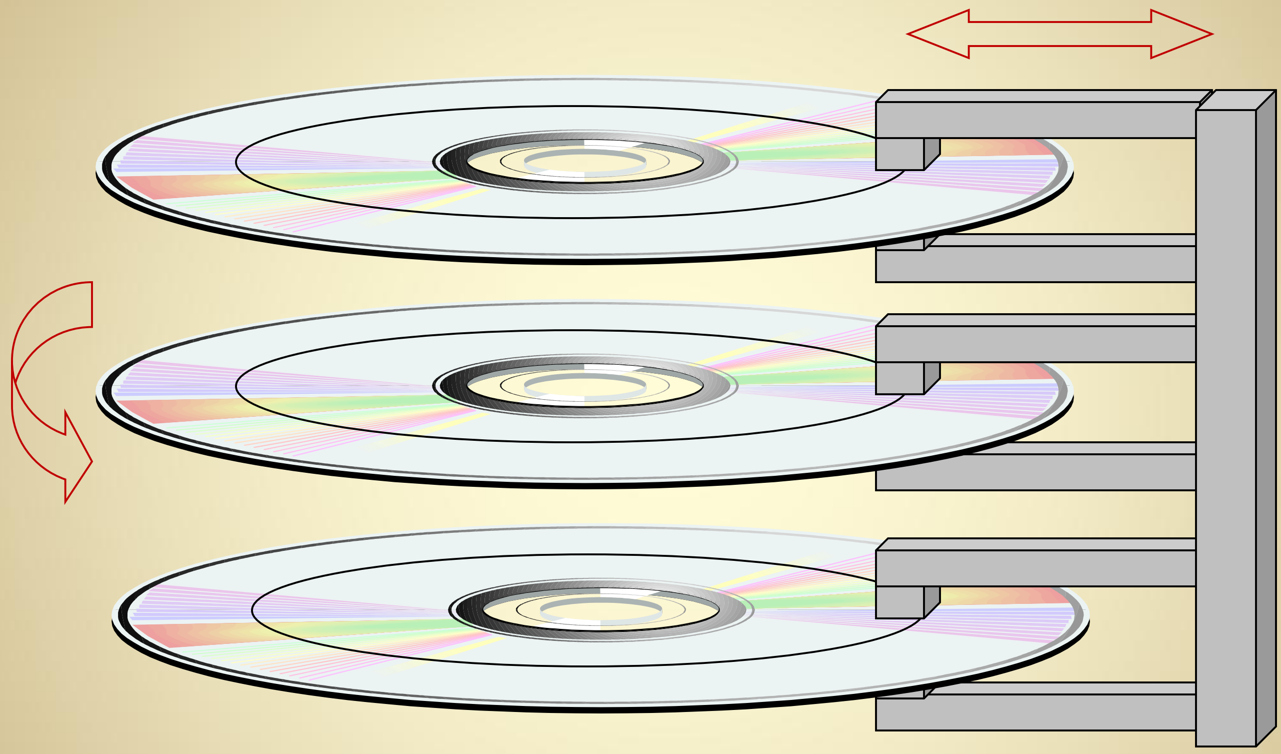
Advanced Database Management Systems

Lecture 14 – Sections 13.3-13.7
File Organization

Review: Hard Drives

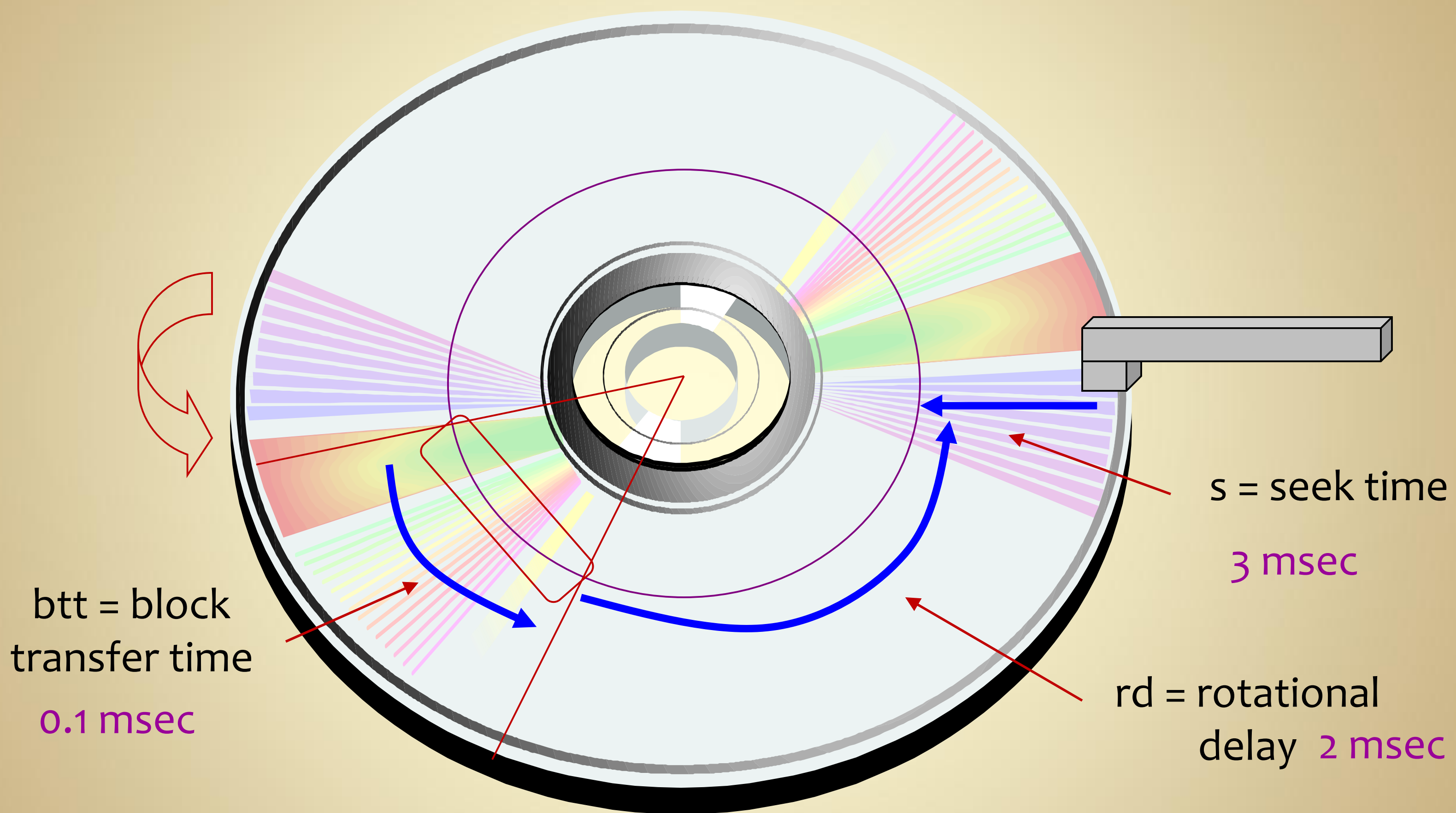


What are the primary factors in computing the time for data transfer?

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

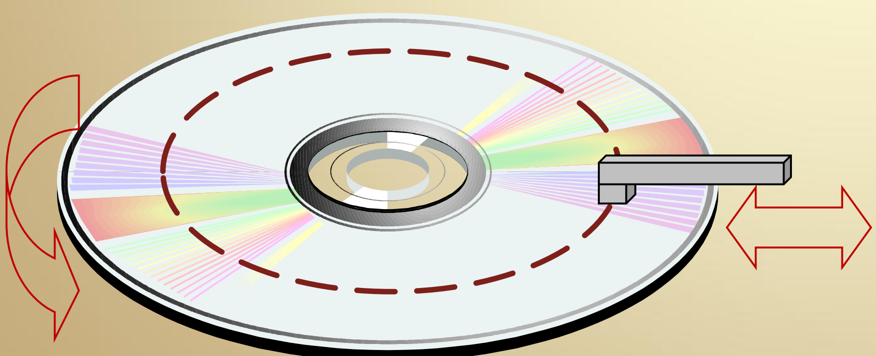
= s + rd + btt (3+2+0.1 = 5.1msec)

Transferring Multiple Blocks

Time to transfer n blocks of data:

randomly located blocks:

rbtt = n*(s + rd + btt)



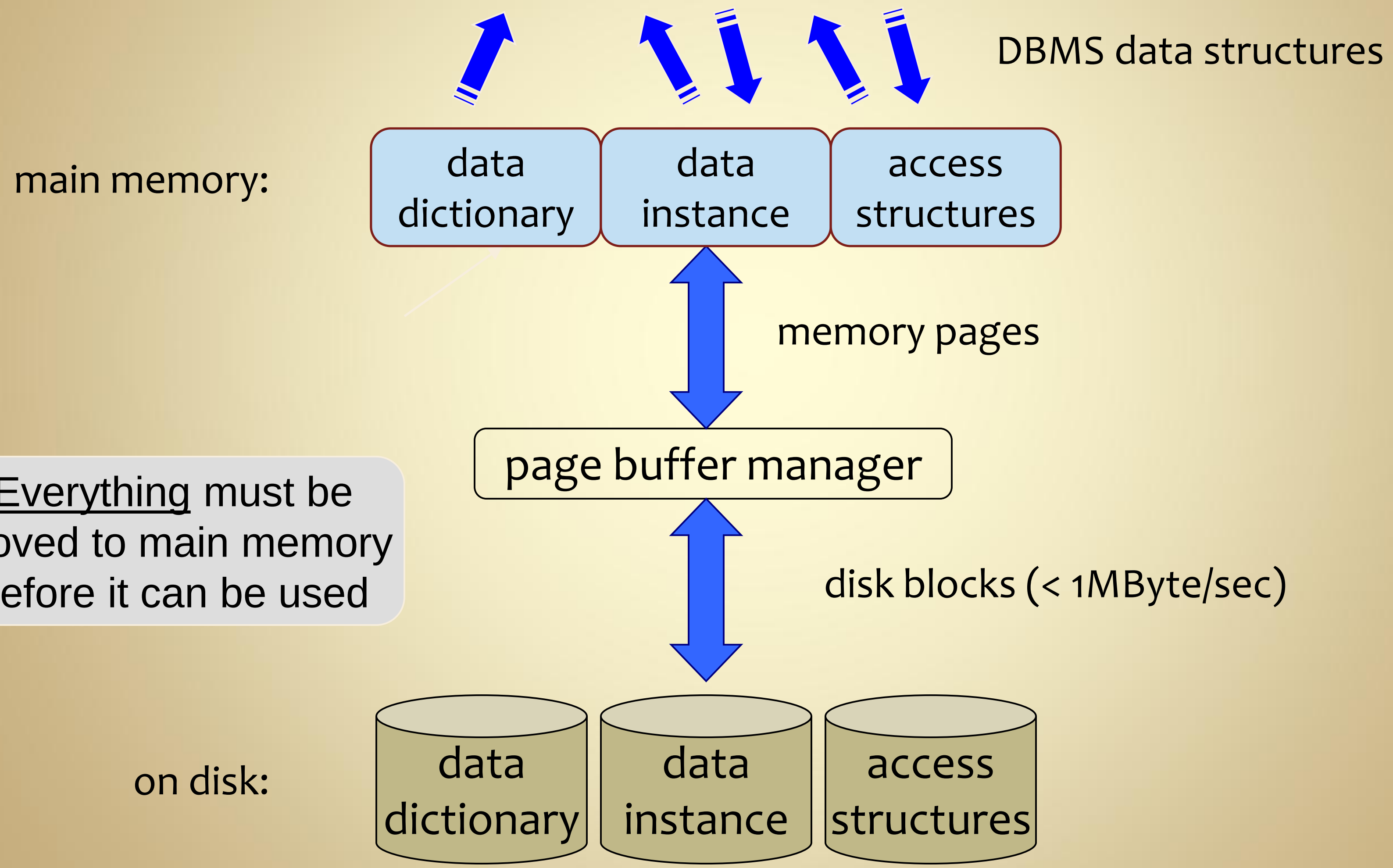
consecutively located blocks:

cbtt = s + rd + n*btt

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

Page Buffer Management



Everything must be moved to main memory before it can be used

Blocks and Pages

- A *page* is a unit of data transfer
  from the DBMS point of view

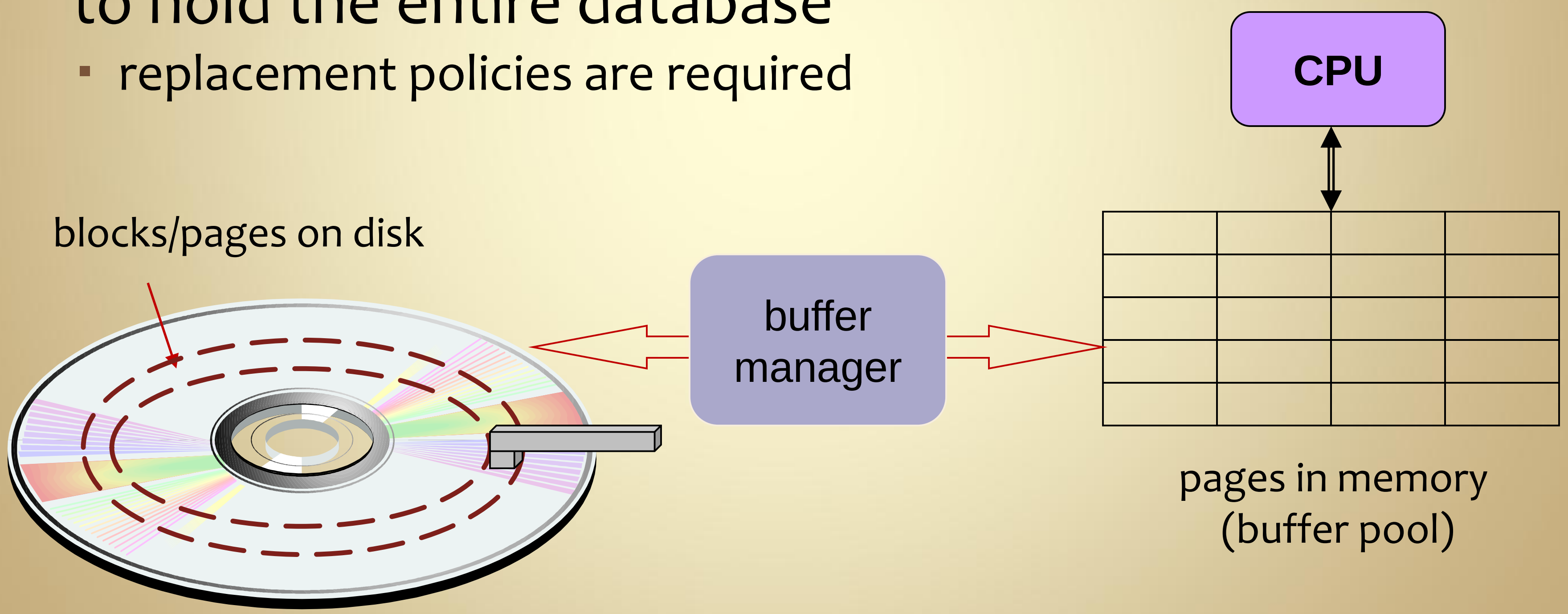
- Disk blocks are the smallest practical page size

- Larger pages are also possible:
  - all data on one track
  - all data on one cylinder
  - same block, same cylinder on all surfaces
- typical page size: 1-10 Kbytes

- typical page transfer time: 1-10 msec

Page Management

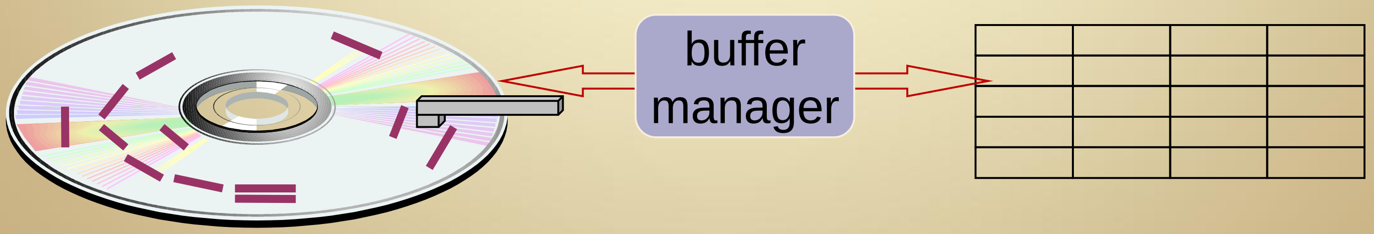
- pages are held in main memory in a *buffer pool*
- the buffer pool is typically not large enough to hold the entire database
  - replacement policies are required



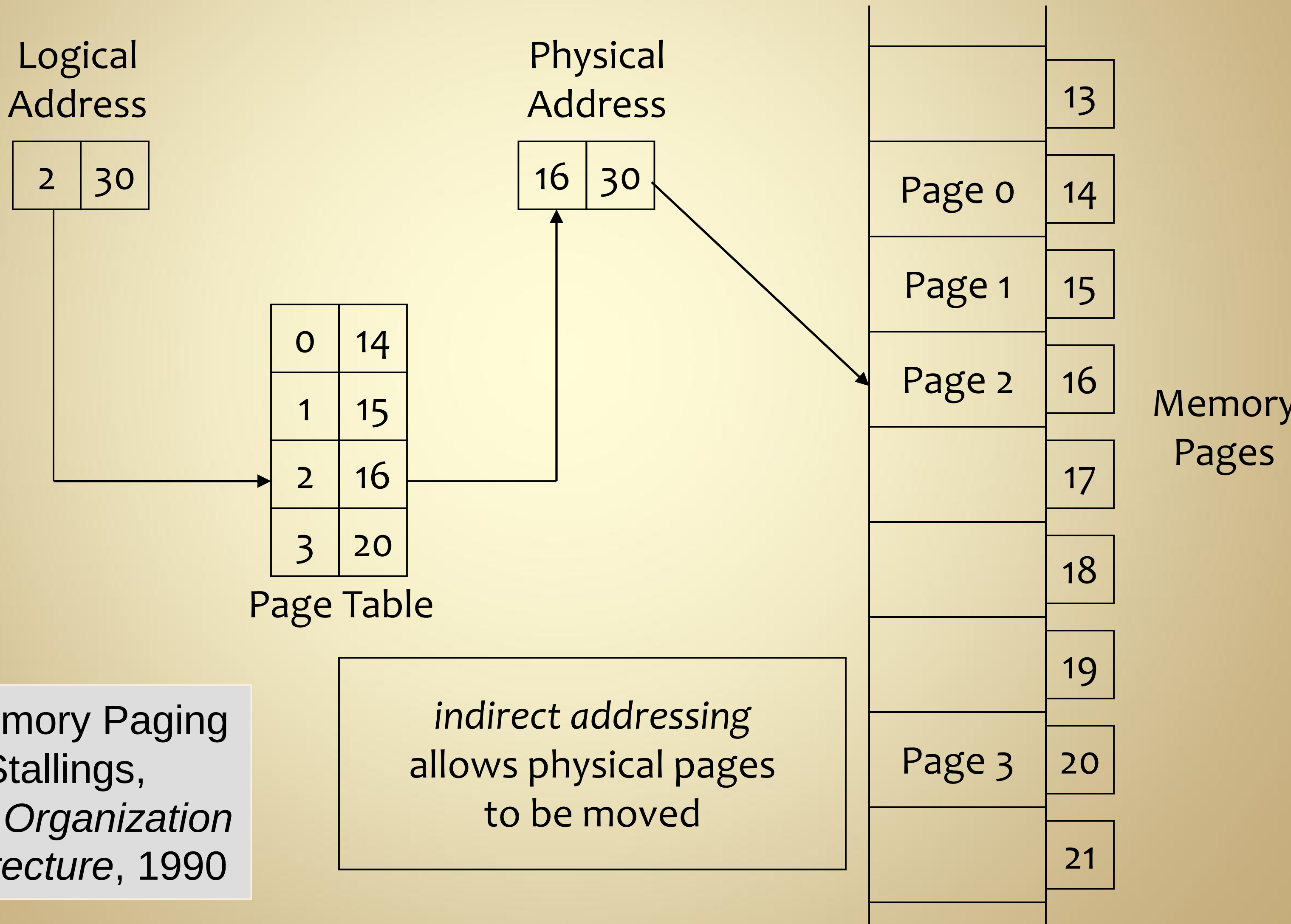
Page Buffer Manager

- The buffer manager must know
  - which pages are in use (pinned pages)
  - which pages have modified data (dirty pages)

- replacement policies:
  - FIFO: oldest non-pinned page is replaced
  - LRU: page that has been unpinned the longest is replaced
  - semantic strategies: if page m is replaces, replace page n next, because pages m and n are related



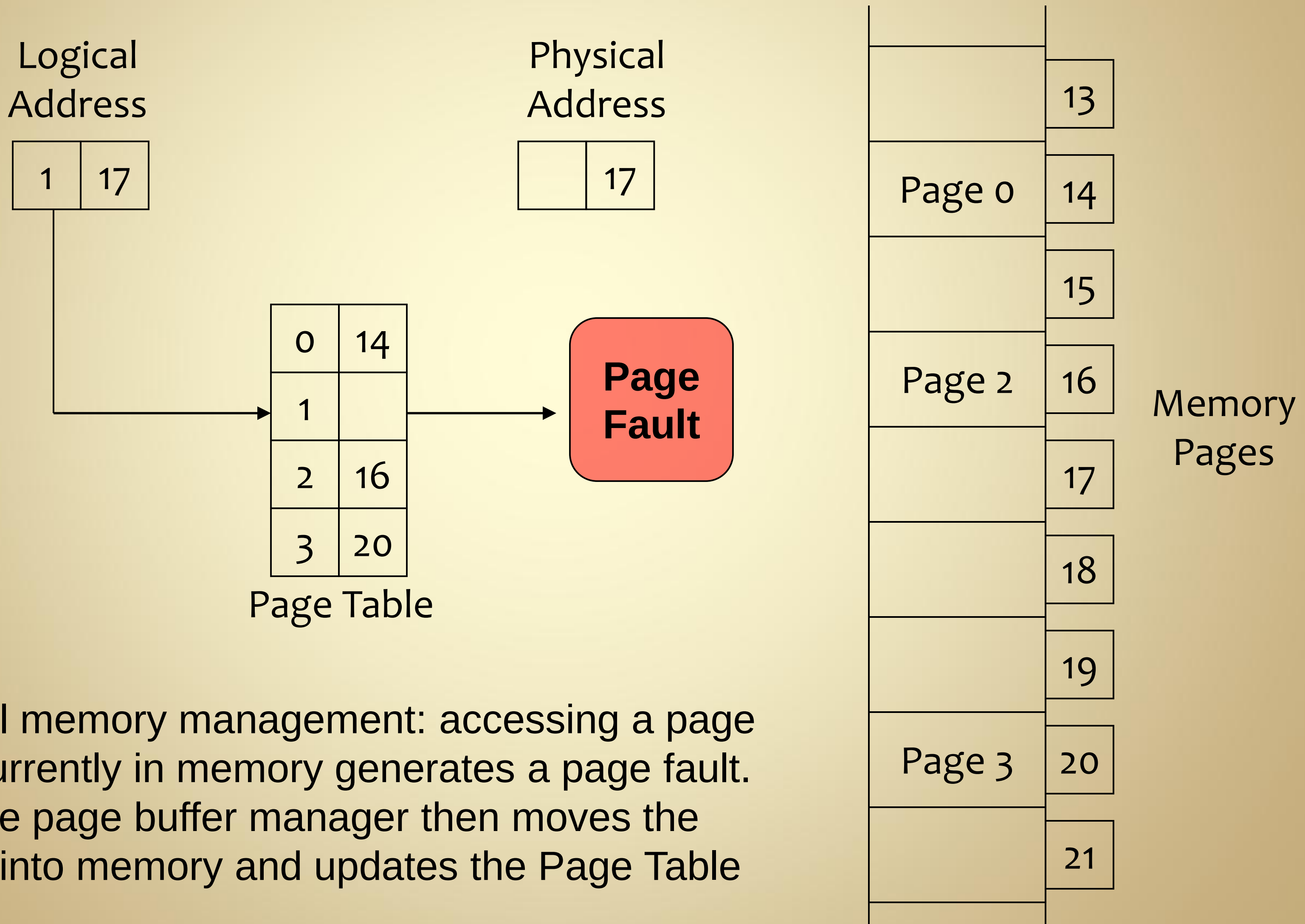
Memory Page Management



indirect addressing allows physical pages to be moved

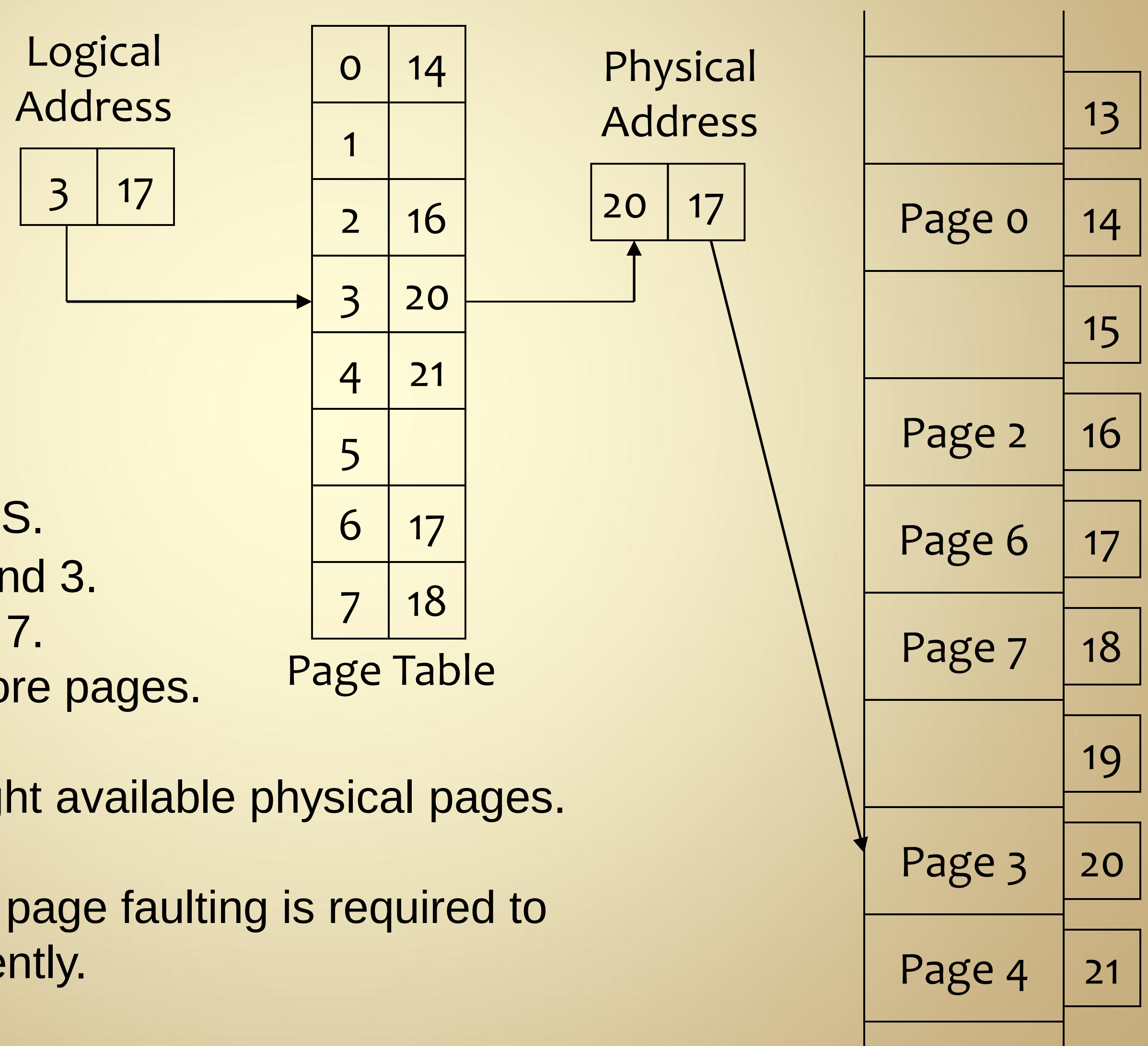
Virtual Memory Paging from Stallings, *Computer Organization and Architecture*, 1990

Virtual Memory Management



Virtual memory management: accessing a page not currently in memory generates a page fault. The page buffer manager then moves the page into memory and updates the Page Table

DBMS Page Buffer Mngt



Example: compute R ⋈ S.
R occupies pages 1, 2 and 3.
S occupies pages 6 and 7.
Result will require six more pages.

Suppose there's only eight available physical pages.

Something smarter than page faulting is required to implement the join efficiently.

DBMS vs. OS Paging

- A DBMS understands the algorithms being applied to the data
  - It can utilize replacement policies appropriate for the current data and operations.
  - It can influence data access patterns.
- An OS does not understand the algorithms being applied to the data structures
  - It must use generic replacement policies.
  - It can only see the data access patterns as they happen.

Exercise

- Conceptual Schema:

  Employee(<u>EmpID</u>, Lname, Fname, Salary, Title)
  - sizeof(EmpID) = 8 bytes
  - sizeof(Lname) = 20 bytes
  - sizeof(Fname) = 20 bytes
  - sizeof(Salary) = 8bytes
  - sizeof(Title) = 20 bytes
- Instance: 100 records
- Disk block size: 1024 bytes

- How can we organize this data in a file on disk?
- How many blocks will the file require?

Exercise

- Conceptual Schema:

  WorksOn(<u>EmpID</u>, <u>ProjectID</u>, Hours, Role, <u>Week</u>)
  - sizeof(EmpID) = 8 bytes
  - sizeof(ProjectID) = 4 bytes
  - sizeof(Hours) = 8 bytes
  - sizeof(Role) = 30 bytes
  - sizeof(Week) = 6 bytes
- Instance: 10,000 records
- Disk block size: 1024 bytes

- How many blocks?
- Should we sort/order the file? What order?

Blocking Factors

- Data must be organized in *blocks*
  - a block (or page) is the unit of transfer between disk and memory
- A *record* is a data unit
  - tuple, object or portion of an access structure
- *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

Blocking and Sorting

- Let R be a relation with key attribute(s) K
- Options for storing R on disk:
  - unsorted, random blocks
  - sorted by key, random blocks
    (impractical, requires "next-block" pointers,
    yielding worst case performance)
  - unsorted, consecutive blocks
  - sorted by key, consecutive blocks

Disk Access Costs

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

	insert	delete	select (on key)
unsorted non-consecutive	O(1)*rbtt	O(n)*rbtt	O(n)*rbtt
unsorted consecutive	O(1)*rbtt	O(n)*cbtt	O(n)*cbtt
sorted consecutive	O(n)*cbtt	O(log n)*rbtt	O(log n)*rbtt

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

binary search

deletion cost is same as selection:
we have to find the record
that we want to delete

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 or records stored as a unit on disk
  - a file is stored on some set of disk blocks
- *spanning* file organization:
  records can be split among blocks
- *non-spanning* file organization:
  whole records must be stored in a single block

Blocking

- ***Blocking***: storing a number of records in one block on the disk.
- ***Blocking factor*** (**bfr**) refers to the number of records per block.
- There may be empty space in a block if an integral number of records does not fill a block (non-spanning)

File Organization

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

Unordered Files

- Also called a **heap** or a **pile** file.

- New records are inserted at the end of the file.
  - Very efficient

- A **linear search** through the file records is necessary to search for a record.
  - Reading ½ the file blocks on the average → expensive: O(n)

Ordered Files

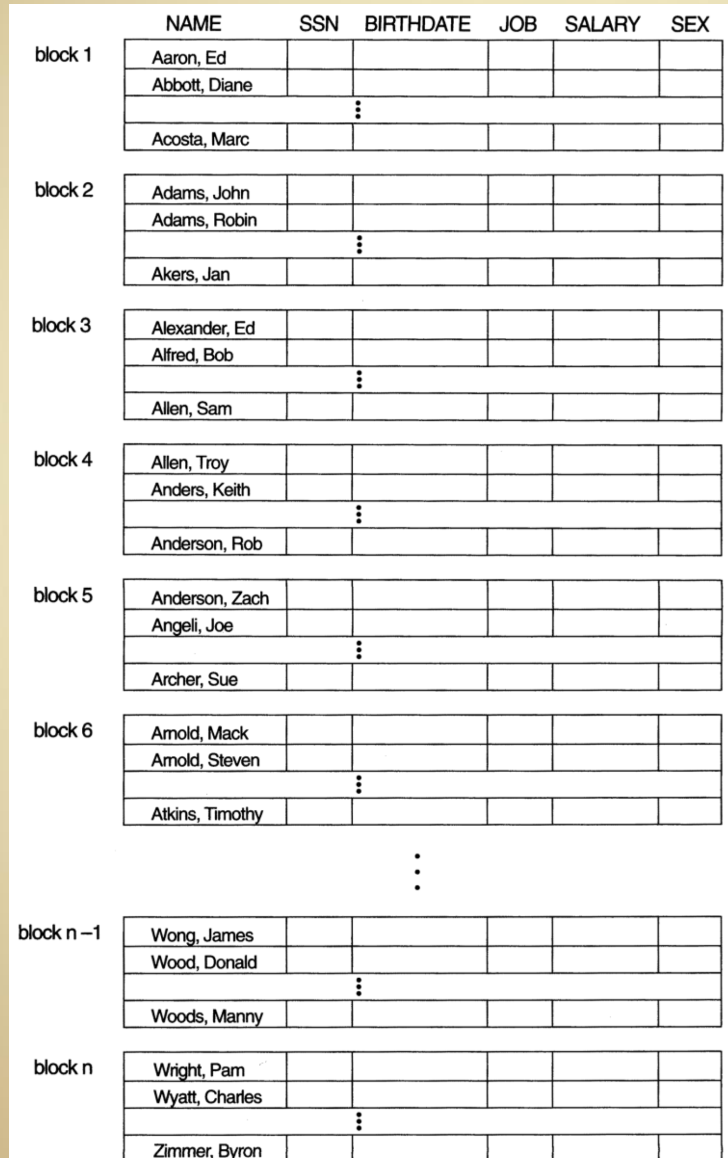
- Also called a **sequential** file.
- File records are kept sorted
  by the values of some *ordering field*.
- Records must be inserted in the correct order.
  - Insertion is expensive: n/2 reads & writes (on average)
- **Binary search** can be used
  to search for a record on its *ordering field* value.
  - O(log n) reads

Faster ins/del on Ordered Files

- Normally, insertion requires moving a lot of records
  - O(n) block read/writes
- Improvement:
  - Keep a separate unordered *overflow* file for new records to improve insertion efficiency
  - Periodically merge overflow file with the main ordered file.

- Deletion also requires moving a lot of records
- Improvement:
  - Simply mark deleted records (and ignore them on subsequent access)
  - Periodically scan the file and removed the marked records
  - Requires a “deleted flag” in each record

Ordered File Example



Blocks and Pages

- A *page* is a unit of data transfer from the DBMS point of view

- Disk blocks are the smallest practical page size
- Larger pages are also possible:
  - all data on one track
  - all data on one cylinder
  - same block, same cylinder on all surfaces

- typical page size: 1-10 Kbytes
- typical page transfer time (ptt): 1-10 msec

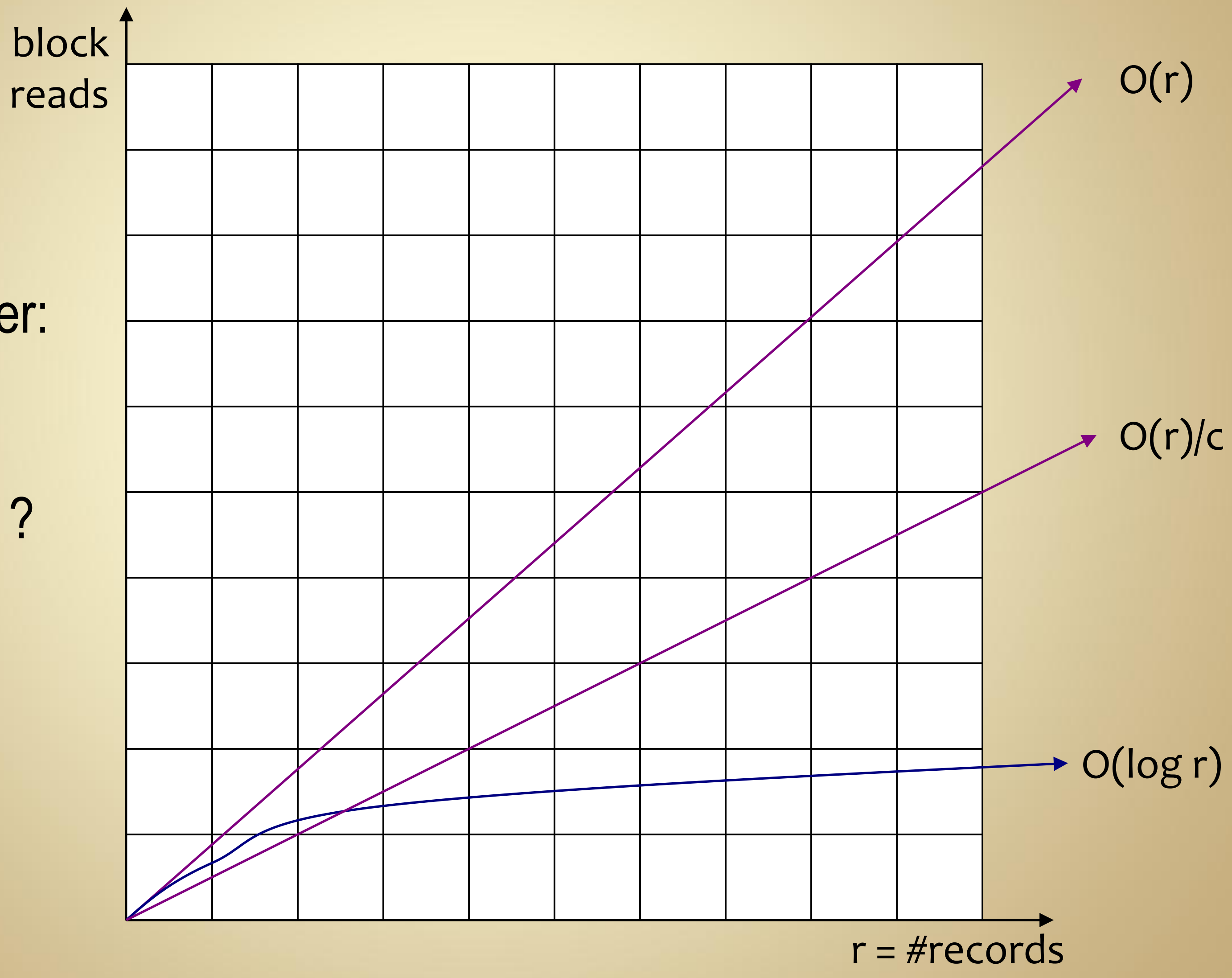
for this course: we'll generally assume one page = one block

Disk Parameters

parameter		typical value	source
T	track size	50 KB	fixed
B	block size	1 KB	formatted
s	seek time	10 msec	fixed
p	rotational velocity	1,000 rps	fixed
rd	(average) rotational delay	3 msec	.5*(1/p)
tr	transfer rate	1MB/sec	T*p
btt	block transfer time	1 msec	B/tr
rbtt	random block transfer time	13 msec	s + rd + btt
cbtt	contiguous block transfer time	1 msec	B/tr

Logarithmic Behavior

Which is better:
O(n)*cbtt
or
O(log n)*rbtt ?



Disk Access Costs

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

	insert	delete	select (on key)
unsorted non-consecutive	O(1)*rbtt	O(n)*rbtt	O(n)*rbtt
unsorted consecutive	O(1)*rbtt	O(n)*cbtt	O(n)*cbtt
sorted consecutive	O(n)*cbtt	O(log n)*rbtt	O(log n)*rbtt

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

binary search

deletion cost is same as selection:
we have to find the record
that we want to delete

Logarithms and Exponents

- logarithms are the inverse of exponents

<u>base 2</u>

$2^1 = 2$	$\log_2(2) = 1$
$2^2 = 4$	$\log_2(4) = 2$
$2^3 = 8$	$\log_2(8) = 3$
$2^{10} = 1024$	$\log_2(1024) = 10$

Quick Quiz:
What is $\log_{10}(345{,}768)$?

<u>base 8</u>

$8^1 = 8$	$\log_{10}(8) = 1$
$8^2 = 64$	$\log_8(64) = 2$
$8^3 = 512$	$\log_8(512) = 3$
$8^{10} = 1{,}073{,}741{,}824$	$\log_8(1{,}073{,}741{,}824) = 10$

Review

```
struct Person
{
  int ID;
  string name;
    …
};
```

What are the average costs of the following functions?

```
int findID(Person p[10000], string name)

int findName(Person p[10000], int id)

sortByName(Person p[10000])

sortByName(Person p[10000])
```

Can we make them faster?

Review

```
struct Person
{
  int ID;
  string name;
    …
};
```

```
int findID(Person p[10000], string name)
int findName(Person p[10000], int id)
sortByName(Person p[10000])
sortByName(Person p[10000])
```

	unordered	ordered by ID	ordered by name
findId	5000	5000	$\log_2(5000) = 12$
findName	5000	$\log_2(5000) = 12$	5000
sort	$10000 * \log(10000) = 140{,}000$		

If we have a lot of lookups of the same kind,
it may be worth the cost of sorting,
but we don't want to continually re-sort for mixed lookups.

Review

```
struct Person
{
  int ID;
  string name;
    …
};
```

```
int findID(Person p[10000], string name)
int findName(Person p[10000], int id)
sortByName(Person p[10000])
sortByName(Person p[10000])
```

	ordered by ID	search tree by name
findId	$\log_2(5000) = 12$	$\log_2(5000) = 12$
findName	$\log_2(5000) = 12$	$\log_2(5000) = 12$

Since we can only have on sort order,
add access structures for other common search parameters.

Access Structures

- Access structures are auxiliary data structures constructed to assist in locating records in a file
- Benefit of access structures
  - allow for efficient data access that is not necessarily possible through file organization
- Cost of access structures
  - storage space: access structures must persist along with the data
  - time: access structures must be modified when file is modified

Single-level Primary Indexes

- If a file is ordered on the primary key, we can construct an ordered index consisting of keys and pointers to blocks
  - the index key is the key of the first record in a block

- 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

  - note: cost of finding the record in the data block is negligible (why?)

Review: Blocking Factors

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

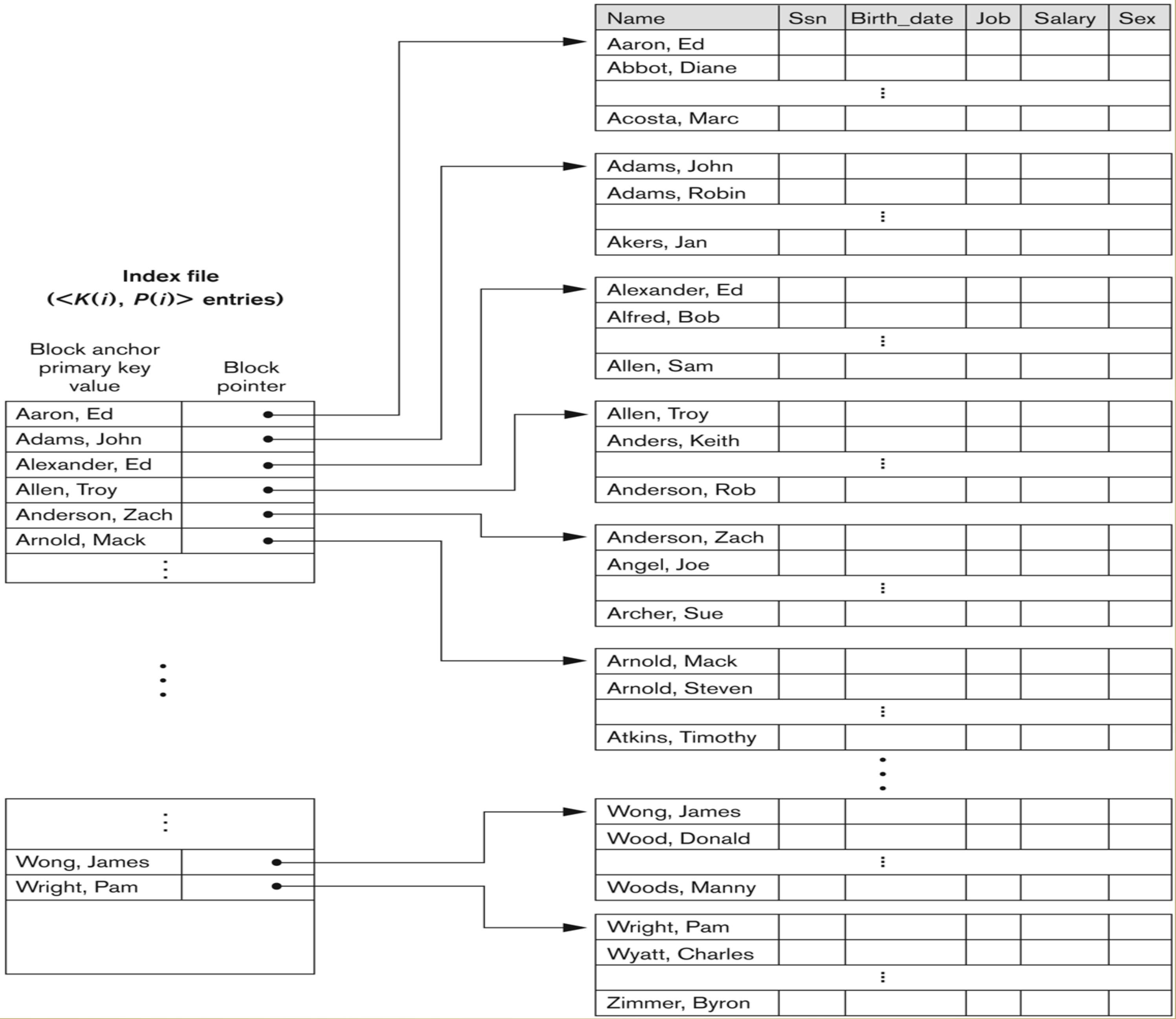
- *Blocking factor* determines how many records can fit in a block
  - bfr_i = floor(B/R_i) where B = block size and R_i = node size

- Blocks required is determined by the blocking factor and the number of records
  - b_i = ceil(r_i / bfr_i) where r_i = number of nodes

Index Classifications

- **dense index:** index entry for every search key value (and hence every record) in the data file.
- **sparse (nondense) index:** index entries for only some of the search values
- **primary index:** defined for a ordering key field in the data records.
  - file must be ordered on the key
- **secondary index:** non-key field or unordered file

Figure 14.1
Primary index on the ordering key field of the file shown in Figure 13.7.



1-level Primary Index Example

# records	r = 200,000 records
record size	R = 100 bytes
block size	B = 1024 bytes
key size	V = 9 bytes
block pointer size	P = 6 bytes

$bfr = \text{floor}(B/R) = 10$ records/block

$b = \text{ceil}(r/bfr) = 20{,}000$ blocks

$R_i = V + P = 15$ bytes

$r_i = b = 20{,}000$ index entries

$bfr_i = \text{floor}(B/R_i) = 68$ index entries / block

$b_i = \text{ceil}(r_i/bfr_i) = 295$

binary search cost

with index:

$\log_2(b_i) + 1 = 10$

without index:

$\log_2(b) = 15$