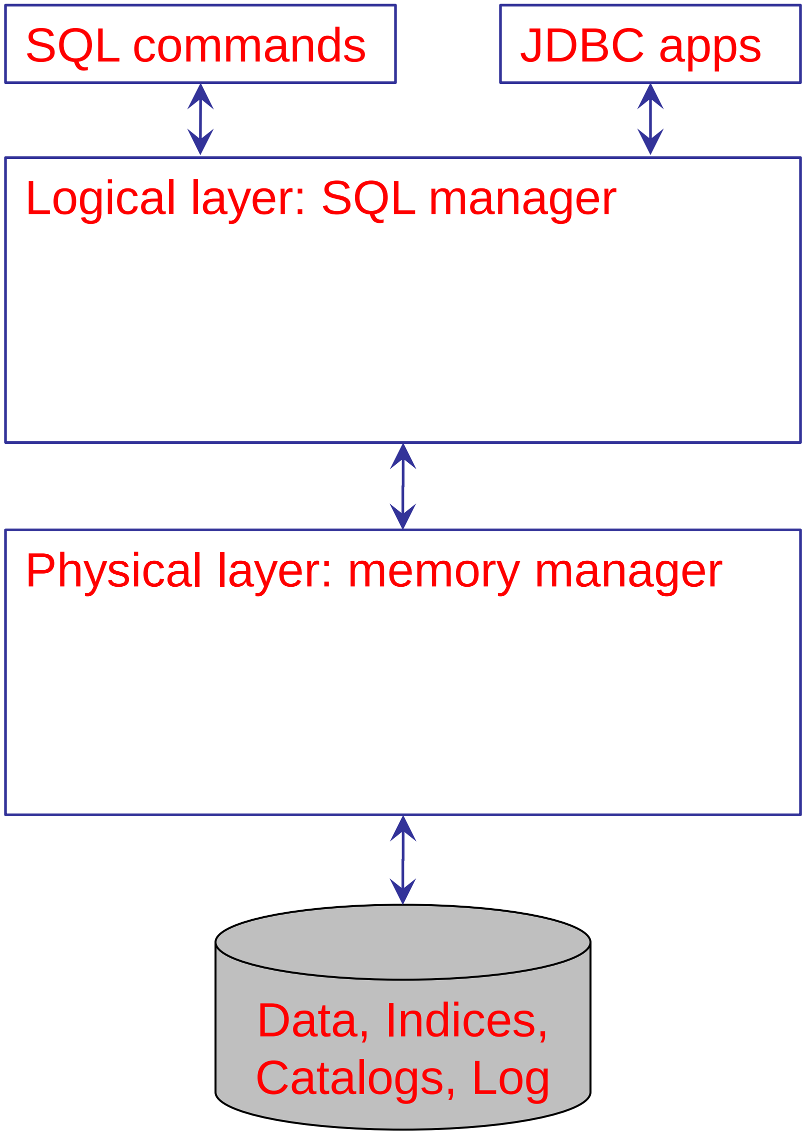
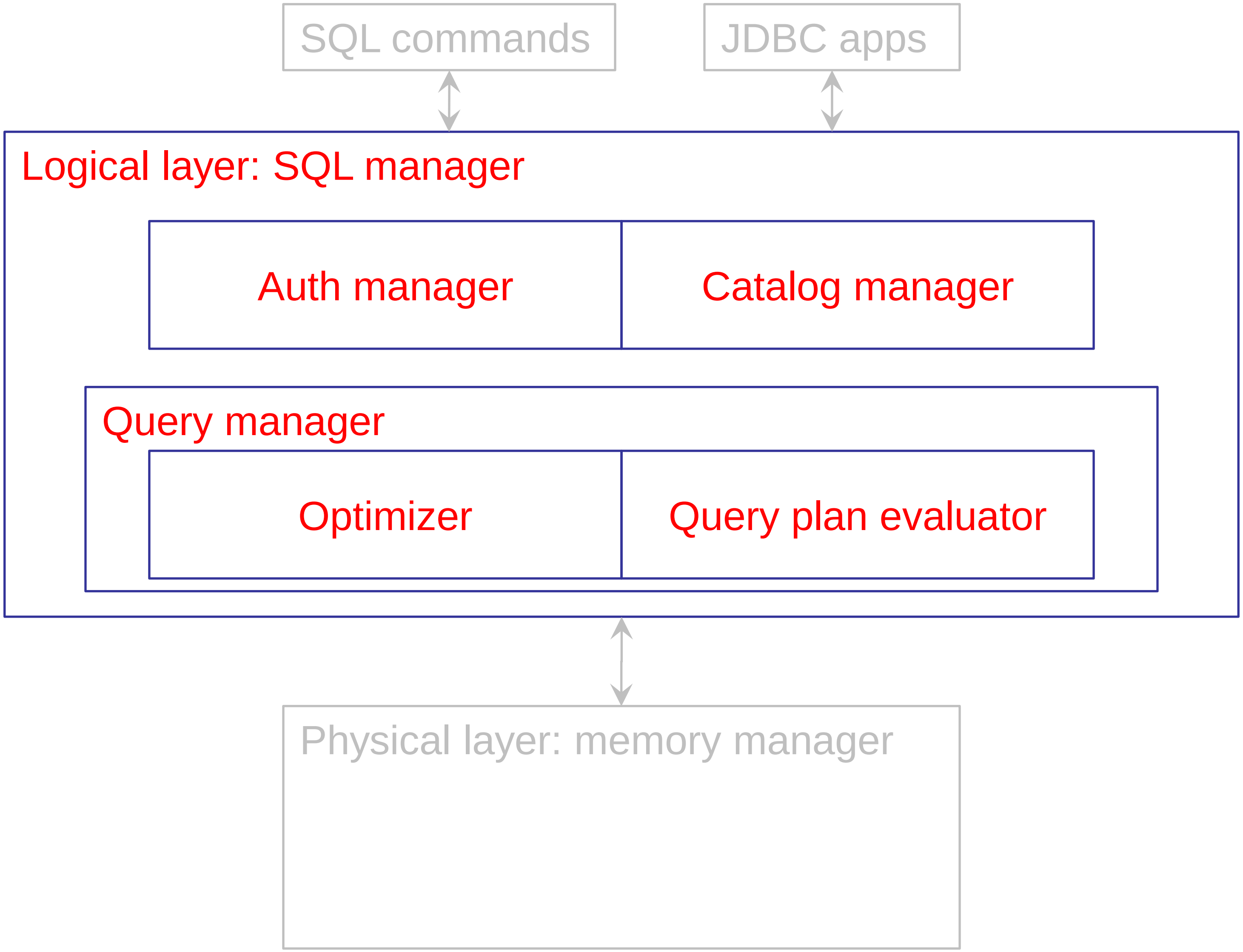
Physical structure of a DBMS

Tecnologie delle Basi di Dati M

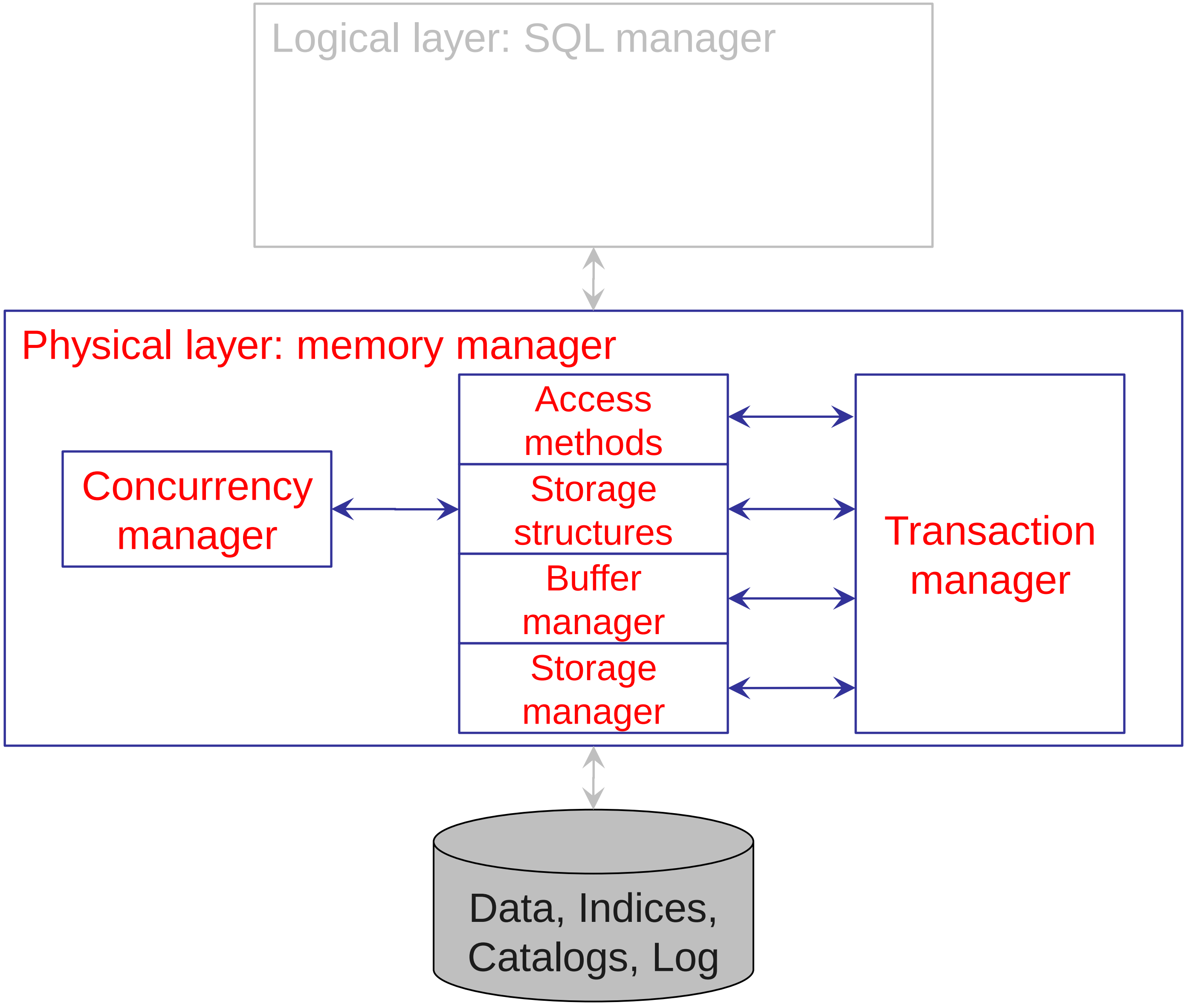
Architecture of a DBMS



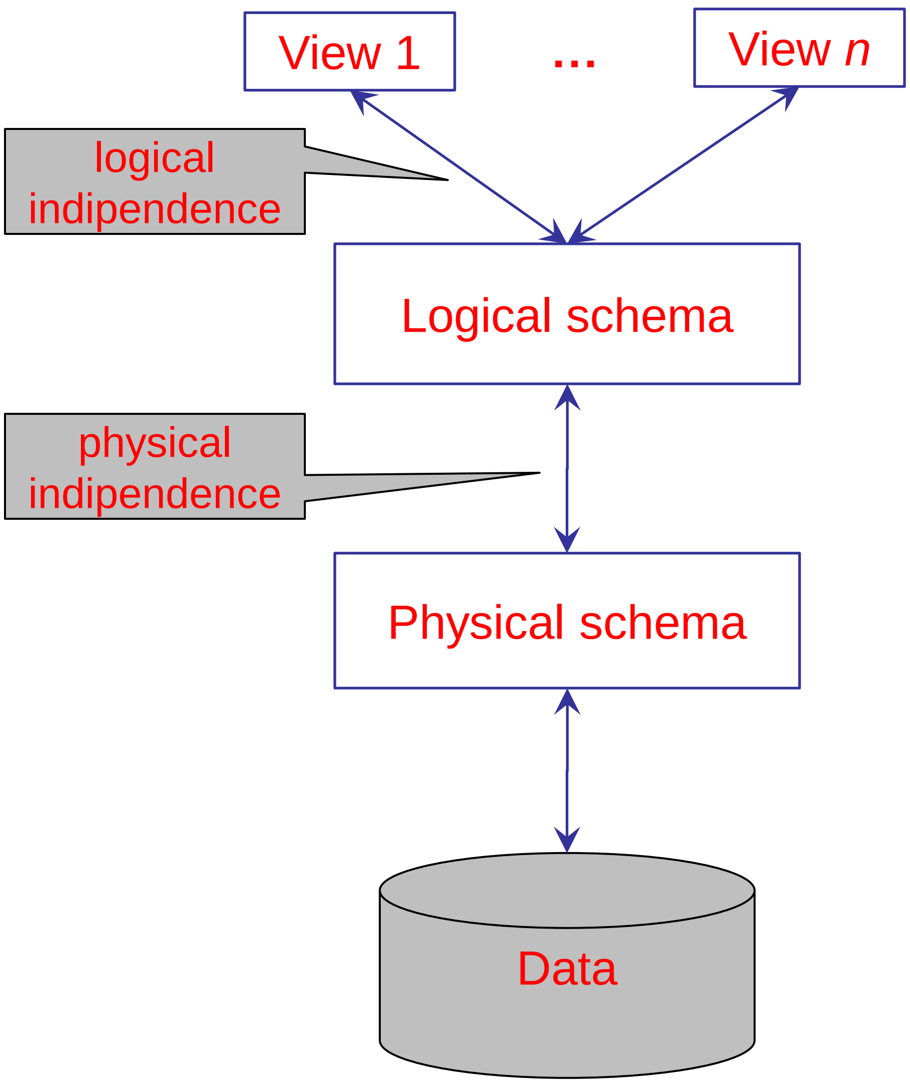
Logical layer



Physical layer



Abstraction levels



- The views describe the users' point of view
- The logical schema defines the logical structure
- The physical schema describes how the data is actually stored on disk

Users of a DBMS

- App users
  - No specific knowledge
- Non-programmer users
  - Interactive access
  - SQL (DML)
- App programmers
  - JDBC
- DB designers
  - Conceptual/logical design
  - SQL (DDL)
- DB Administrators
  - Specific knowledge of the DBMS
  - Tuning of DBMS
- DBMS programmers
  - They code DBMS

Storage manager

- Memory of a computer is organized in a three-level hierarchy:
  1. main memory (RAM)
  2. secondary storage (magnetic disks)
  3. tertiary storage (tape and optical jukeboxes)
- From the DBMS point of view, we can ignore:
  0. internal memory (cache and logs)
  4. off-line memory

Performance of a memory

- Given a memory address, performance is measured in terms of access time, defined as the sum of:
  - latency
    (time needed to access the first byte)
  - transfer time
    (time needed to move data)

$$\textbf{access time} = \textbf{latency} + \frac{\textbf{data size}}{\textbf{transfer speed}}$$

Main memory

- Characteristics of main memory (e.g., DIMM):
  - Access time: ~50 ns
  - Speed: ~3 GB/s
  - Capacity: ~1 GB
  - Volatile
  - Cost: ~30 €/GB

Secondary memory

- Characteristics of secondary memory (e.g., HD):
  - Access time: ~5 ms
  - Speed: ~120 MB/s
  - Capacity: <2 TB
  - Non-volatile
  - Cost: ~0.10 €/GB

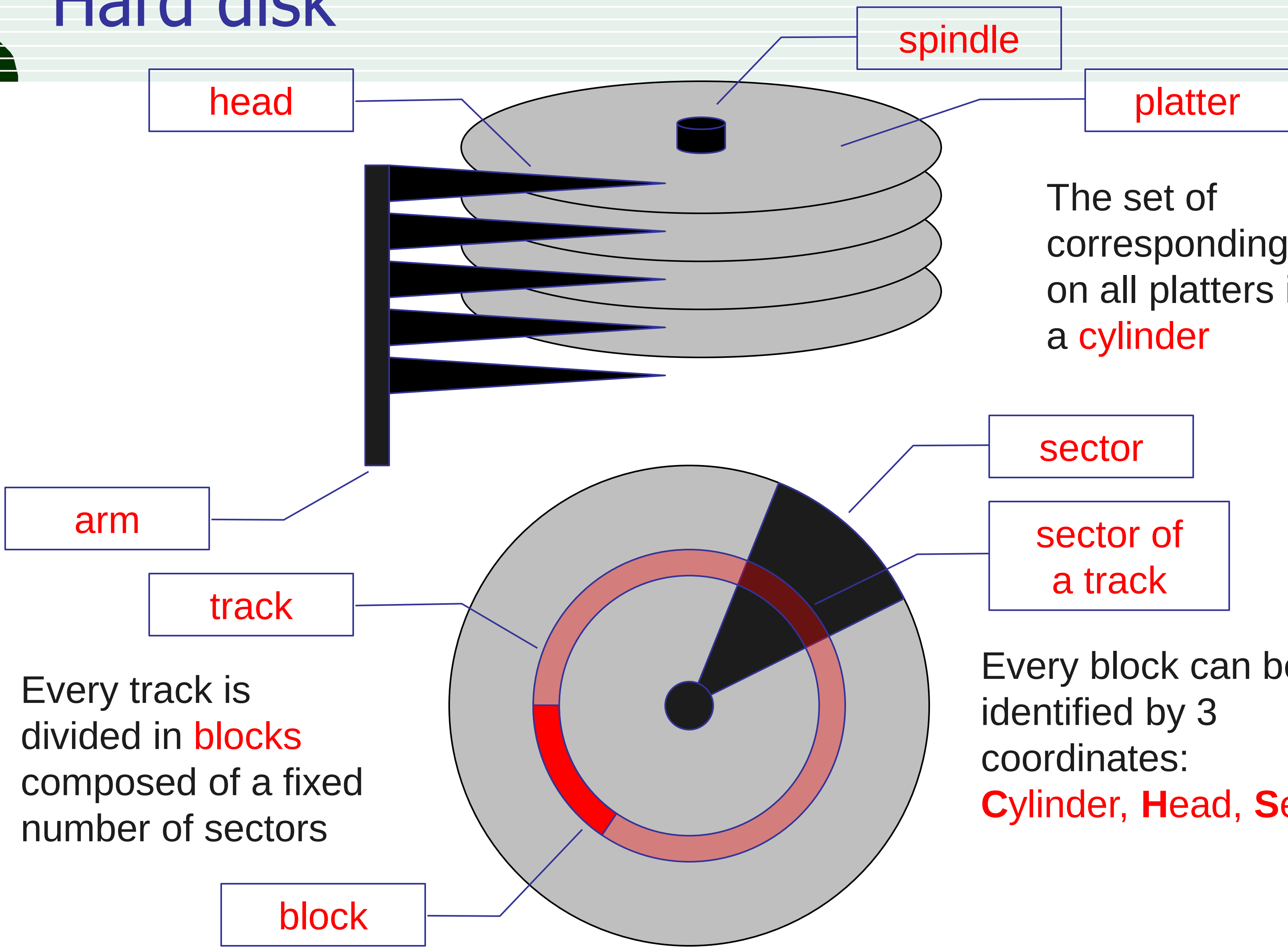
Tertiary memory

- Characteristics of tertiary memory (e.g., DAT72):
  - Access time: ~30 s
  - Speed: ~3 MB/s
  - Capacity: 72 GB
  - Non-volatile
  - Cost: ~0.10 €/GB

Implications for DBMSs

- Because of its size, a DB is normally stored on disk (possibly also on other types of devices)
- Data must be transferred to main memory to be processed by the DBMS
- Data are not transferred as single tuples, rather as blocks (or **pages**, a term commonly used when data are in memory)
- Often, I/O operations are the system bottleneck, it is necessary to optimize the physical implementation of the DB by way of:
  - Efficient organization of tuples on disk
  - Efficient access structures
  - Efficient management of memory buffers
  - Efficient query processing strategies

Hard disk

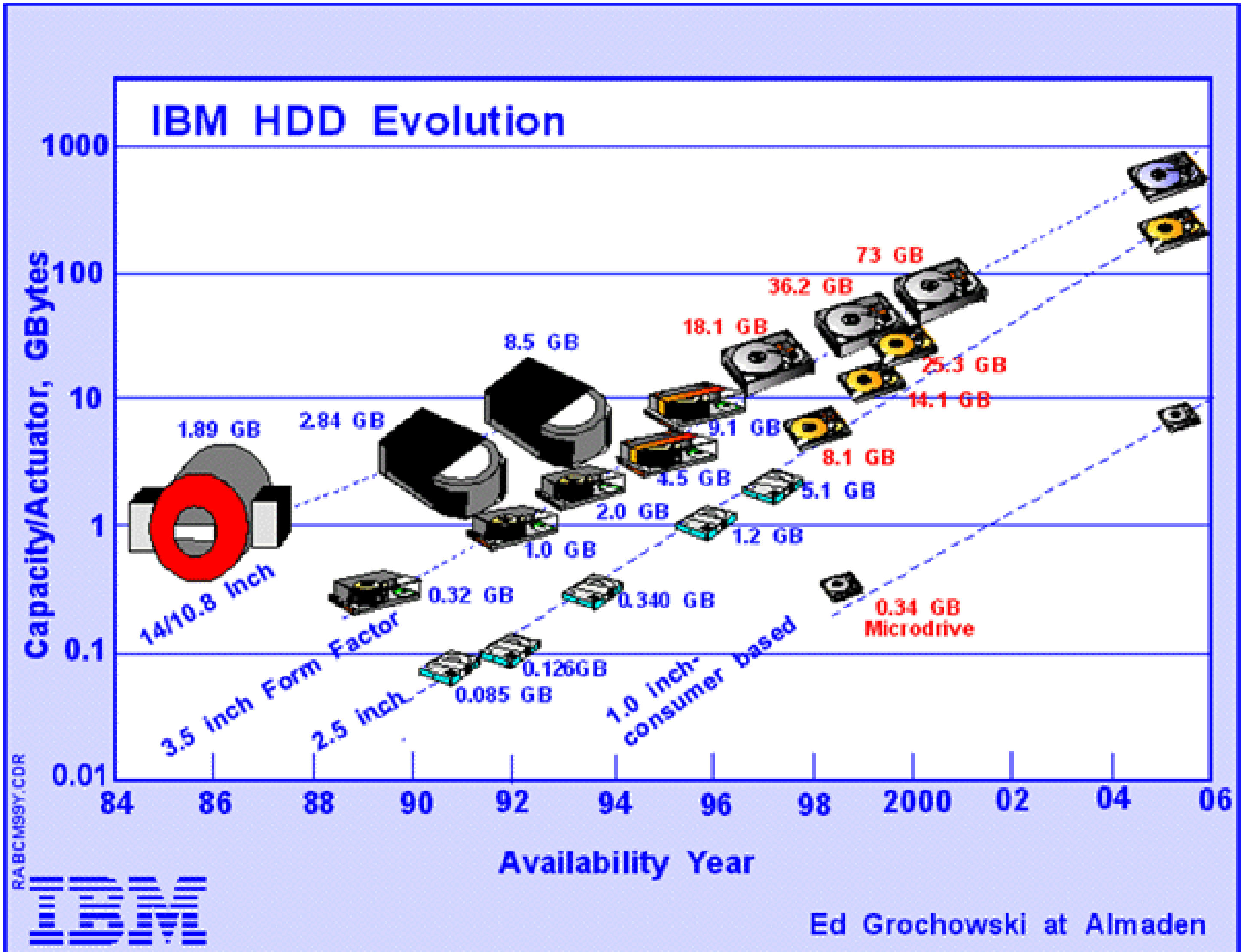


The set of corresponding tracks on all platters is called a cylinder

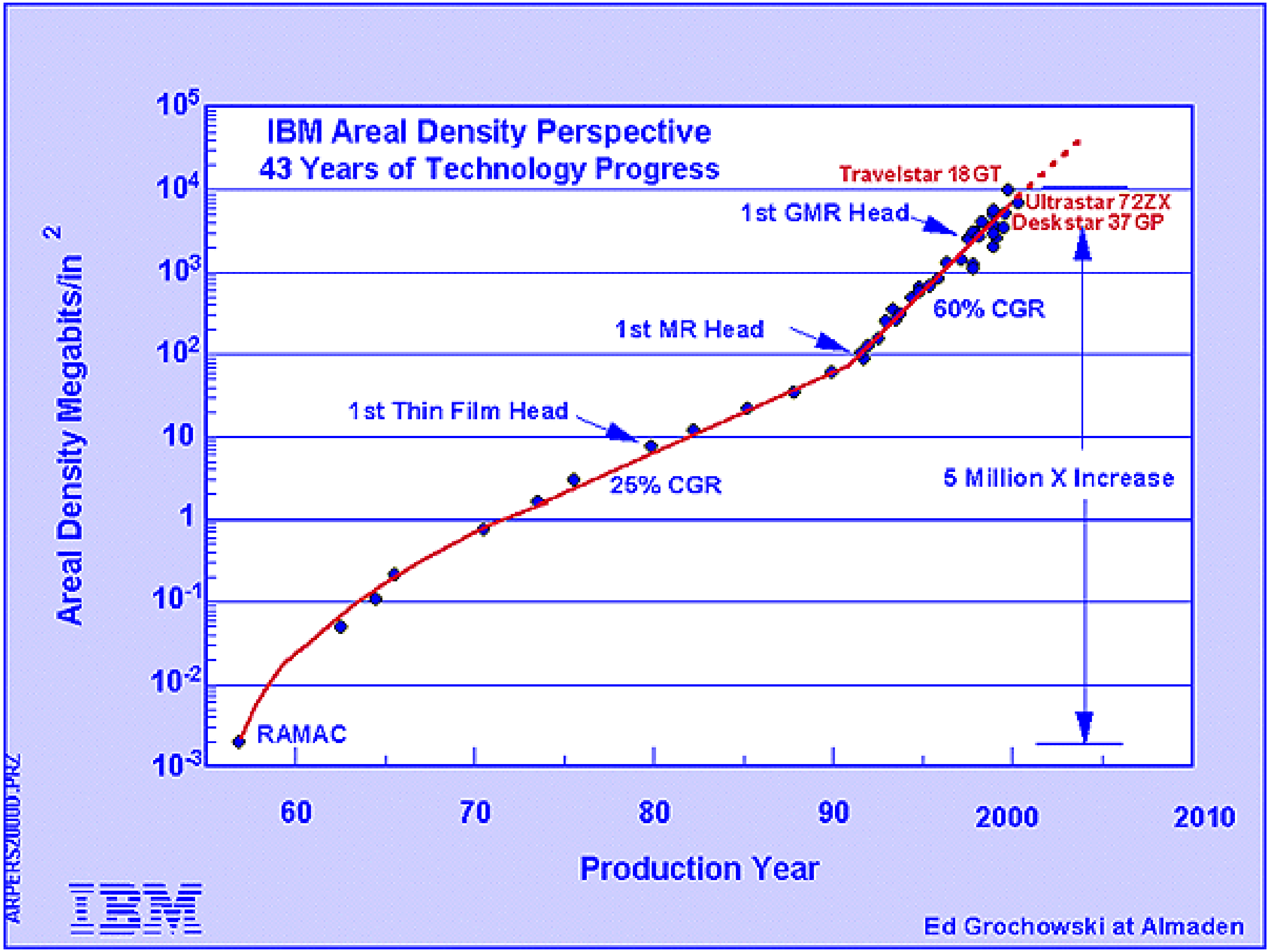
Every track is divided in blocks composed of a fixed number of sectors

Every block can be identified by 3 coordinates:
Cylinder, **H**ead, **S**ector

Capacity evolution

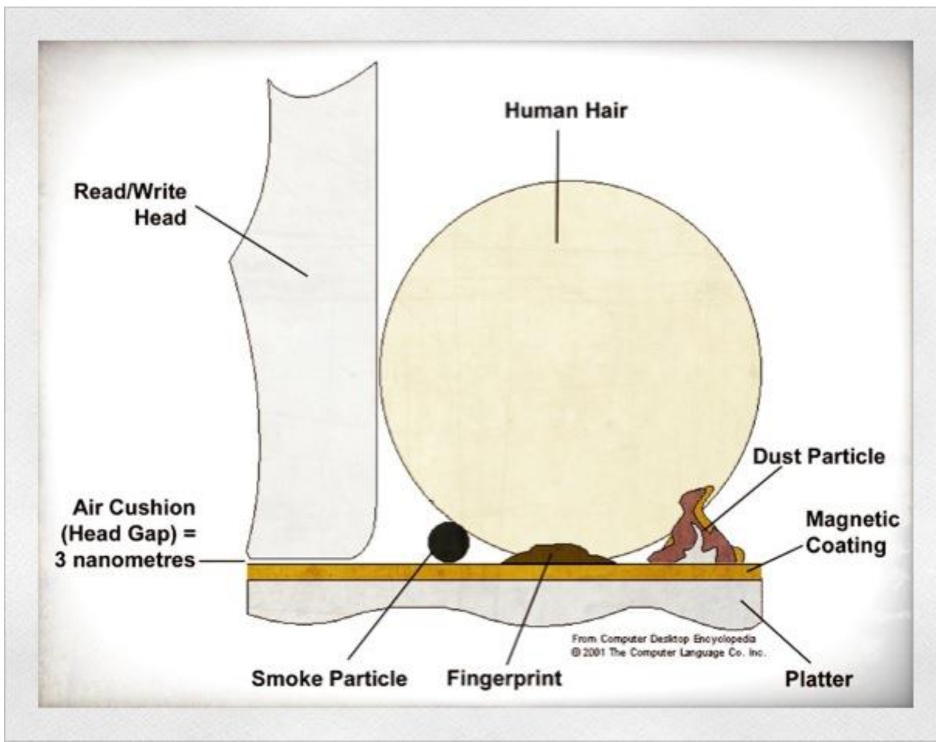


Density evolution



The heads

- During writing, heads convert bits into magnetic pulses, which are then recorded on the disc(s) surface; during reading, heads perform the reverse conversion
- They are very critical components for the effects they have on performance and are the most expensive parts of a HD



An example: Seagate **Barracuda 7200.12**

Drive specification	ST31000528AS	Cache buffer	32 Mbytes
Interface	Serial ATA	Average latency	4.16 msec
Formatted capacity	1000 Gbytes	Power-on to ready	<10.0 sec max
Guaranteed sectors	1,953,525,168	Standby to ready	<10.0 sec max
Heads	4	Track-to-track seek time	<1.0 msec read;
Discs	2		<1.2 msec write
Bytes per sector	512	Average seek, read	<8.5 msec
Sectors per track	63	Average seek, write	<9.5 msec
Read/write heads	16		
Cylinders	16,383		
Recording density	1413 kbits/in max		
Track density	236 ktracks/in avg		
Areal density	329 Gbits/in^2 avg		
Spindle speed	7,200 RPM		
Internal data tr. rate	1695 Mbits/sec max		
Sustained data tr. rate	125 Mbytes/sec max		
I/O data-tr. rate	300 Mbytes/sec max		

Performance

- Internal: they depend on
  - Mechanical characteristics
  - Techniques for encoding and storage of data
  - Disk controller (interface between disk HW and the computer)
    - It accepts high-level commands to read/write sectors and controls the mechanism
    - It adds information between sectors for error checking (checksum)
    - It checks correctness of the writes re-reading written sectors
    - It performs the mapping between logical addresses of blocks and disk sectors
- External: they depend on
  - Interface type
  - Architecture of the I/O sub-system
  - File system

Internal performance

- The most important figure is latency (the time taken to get to information of interest), composed by:
  - Command Overhead Time:
    time needed for issuing commands to the drive
    - It is of the order of 0.5 ms and can be neglected
  - Seek Time (**Ts**):
    time needed by the arm to move to the desired track
    - The average seek time, of the order of 2-10 ms, is, in the case of tracks with a constant number of sectors, 1/3 of the maximum seek time
    - The seek time for writing is higher by about 1 ms
      of the seek time for reading
  - Settle Time:
    time needed by the arm for stabilizing
  - Rotational Latency (**Tr**):
    wait time for the first sector to be read
    - Average rotational latency is 1/2 of the worst case
    - About 2-11 ms

Example: IBM 34GXP drive

Component	Best-Case Figure (ms)	Worst-Case Figure (ms)
Command Overhead	0.5	0.5
Seek Time (Ts)	2.2	15.5
Settle Time	<0.1	<0.1
Rotational Latency (Tr)	0.0	8.3
Total	2.8	28.4

Rotational Latency

(60/Spindle Speed)* 0.5 * 1000

Spindle Speed (RPM)	Worst-Case Latency (Full Rotation) (ms)	Average Latency (Half Rotation) (ms)
3,600	16.7	8.3
4,200	14.2	7.1
4,500	13.3	6.7
4,900	12.2	6.1
5,200	11.5	5.8
5,400	11.1	5.6
7,200	8.3	4.2
10,000	6.0	3.0
12,000	5.0	2.5
15,000	4.0	2.0

Transfer Rate

- It is the maximum drive speed for reading/writing data
  - Typically in the order of 10 MB/s
  - It refers to the bit transfer speed from/to platters to/from the controller's cache
- It can be estimated as:

$$\frac{\text{(bytes/sector) x (sectors/track)}}{\text{rotation time}}$$

Example: With 512 bytes/sector, 368 sectors/track, 7200 rpm transfer rate equals (512 x 368) / (60 / 7200) = 21.56 MB/s

- In practice, transfer rate from/to HD are lower than the nominal values (4-10 MB/s)

Pages

- A block (or **page**) is:
  - a contiguous sequence of sectors on a single track
  - it is the (atomic) unit of I/O fro transferring data to/from main memory
- Typical values for the page size are some KB (4 - 64 KB)
  - Small pages means a higher number of I/O operations
  - Large pages often cause internal fragmentation (pages only partially full) and require more memory to be loaded
- The page transfer time (**Tt**) from disk to main memory depends on:
  - the page size (**P**)
  - the transfer rate (**Tr**)

Example

- transfer rate of 21.56 MB/sec

- Page size P = 4 KB
- Tt = 4/(21.56*1024) = 0.18 ms

- Page size P = 64 KB
- Tt = 64 /(21.56*1024) = 2.9 ms

The physical DB

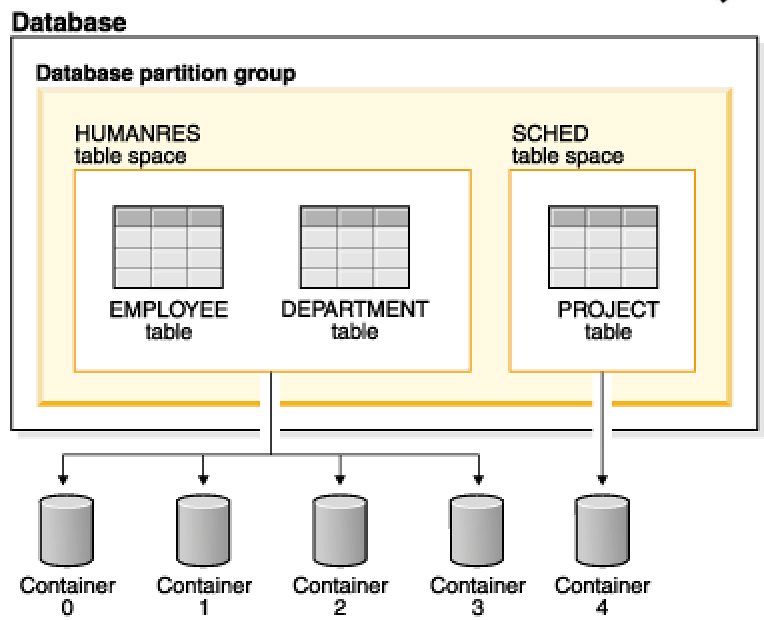
- At the physical level, a DB consists of a set of files,
  each seen as a collection of pages, of fixed size (e.g., 4 KB)
- Each page stores several records
  (corresponding to logical tuples)
- In turns, a record consists of several fields,
  with fixed and/or vaiable size,
  representing the tuple’s attributes

The physical DB(cont.)

- The DBMS “files” considered here do not necessarily correspond to those stored in the OS file system
- Extreme cases:
  - Every DB relations is stored in a separate file
  - The whole DB is stored in a single file
- In practice, at the physical level, every DBMS exploits specific solutions, very complex and flexible

The storage model of DB2

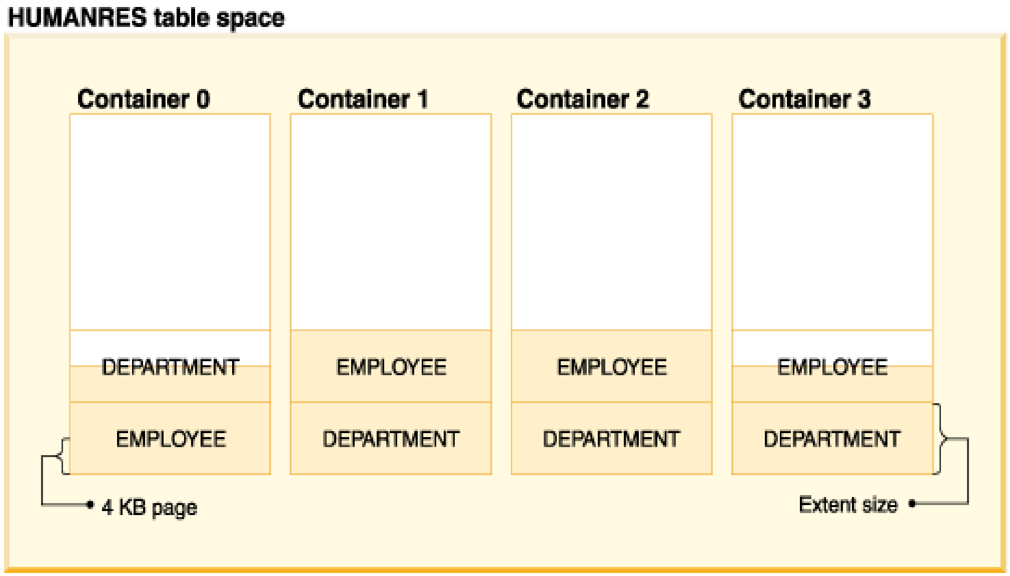
- DB2 organizes the physical space into tablespaces, each composed as a collection of containers
- Pro: flexibility
  - We can store different things in different devices
  - To add new tables we can add tablespaces



- Every relation is stored into a single tablespace, but a single tablespace can store several relations
- Every container can be a device, a directory, or a file
- The DBMS automatically balances data into containers

Using extents

- Every container is divided into extents, representing the minimum allocation entity on disk, composed by a set of contiguous pages with size 4 KB (default value for P)
  - The extent size can differ among tablespaces and is chosen when the tablespace is created
- Every extent stores data of a single relation



Tablespace

- Every database should contain at least three tablespaces, used for storing different data:
  - catalogs (system tablespace)
  - user tables (one or more user tablespaces)
  - temporary tables (one or more user tablespaces)

Tablespace types

- 3 different types of tablespace exist:
  - SMS (System Managed Space)
    storage is managed by the OS
  - DMS (Database Managed Space)
    storage is managed by the user
  - Automatic Storage
    is managed by the DBMS

Comparison among types (i)

- Creation
  - SMS: `CREATE TABLESPACE … MANAGED BY SYSTEM`
  - DMS: `CREATE TABLESPACE … MANAGED BY DATABASE`
  - AS: `CREATE TABLESPACE … [MANAGED BY AUTOMATIC STORAGE]`
- Defining containers
  - SMS: directory name
  - DMS: device or file (fixed size)
  - AS: automatic, containers exist in every path associated to the DB

Comparison among types (ii)

- Initial allocation
  - SMS: OS-based (fragmentation likely)
  - DMS: at creation time (fragmentation unlikely when container=device)
  - AS: system/user: at creation time
    temporary: whenever needed
- Modifying containers
  - SMS: not allowed
  - DMS: creating/removing containers
    (automatic balance)
  - AS: automatic

Comparison among types (iii)

- Additional memory request
  - SMS: until file system is full
  - DMS: containers can be extended
  - AS: automatic extension of containers
- Maintenance
  - SMS: none
  - DMS: creating/removing containers
    balancing containers
  - AS: reducing tablespace
    balancing containers

Comparison among types (iv)

- Maximum size
  - SMS: n x maximum file size
  - DMS: 512 GB (64 TB for large tablespace)
  - AS: file system size
- Object separability (e.g., tables and indices)
  - SMS: no, everything in the same tablespace
  - DMS: objects can be stored into different tablespaces
  - AS: objects can be stored into different tablespaces

Which is the best type?

- AS
  - Large tables
  - Simplified management of container enlargement
  - Storing different objects (e.g., indices, tables) into different tablespaces (performance)
- DMS
  - Large tables
  - Control over where data are stored
  - Control over storage status
  - Storing different objects (e.g., indices, tables) into different tablespaces (performance)
- SMS
  - Small tables
  - Control over where data are stored
  - Control over storage status

Tablespace attributes

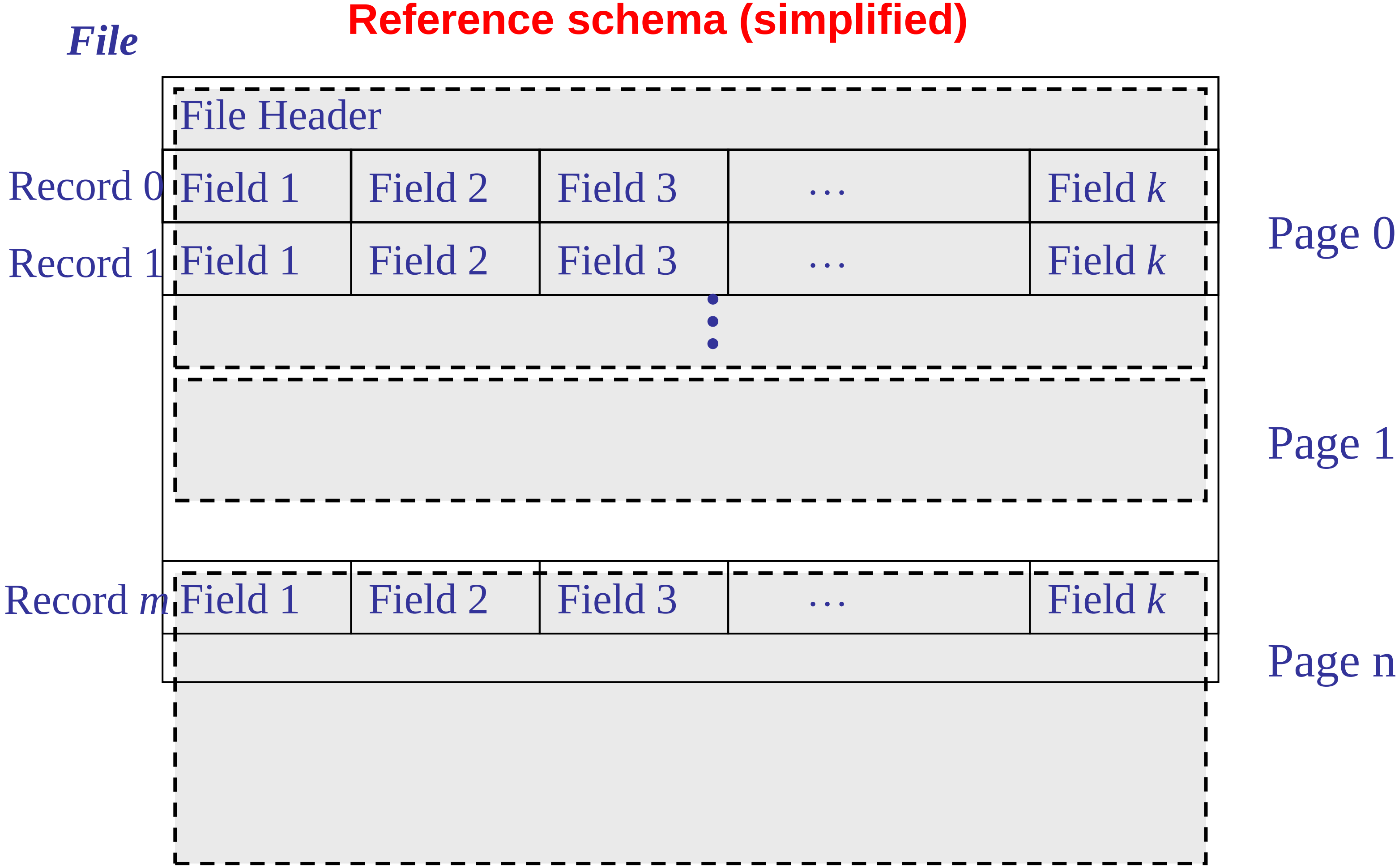
- When a tablespace is created, it is possible to specify a set of parameters
- For example:
  - EXTENTSIZE: number of extent blocks
  - BUFFERPOOL: name of the buffer pool associated to the tablespace
  - PREFETCHSIZE: number of pages to be transferred to memory before they are actually requested
  - OVERHEAD: estimate of average latency time for an I/O operation
  - TRANSFERRATE: estimate of average transfer time for a page transfer
- The last two parameters are used by the optimizer

Why don't we simply use the file system?

- We will show that, performance of a DBMS highly depend on the organization of data on disk
- Intuitively, data allocation should aim to reduce data access time
  - For this, we should know how (logically) data are to be processed and which are (logical) relations existing between data
- The file system could be oblivious to all such information
  Examples:
  - If two relations contain correlated data (e.g., by a join), storing them in continuous cylinders could be a good idea, so as to reduce seek time
  - If a relation contains BLOB attributes, storing them in a different place (with respect to other attributes) could be a good idea

Organizing data in a file

Reference schema (simplified)



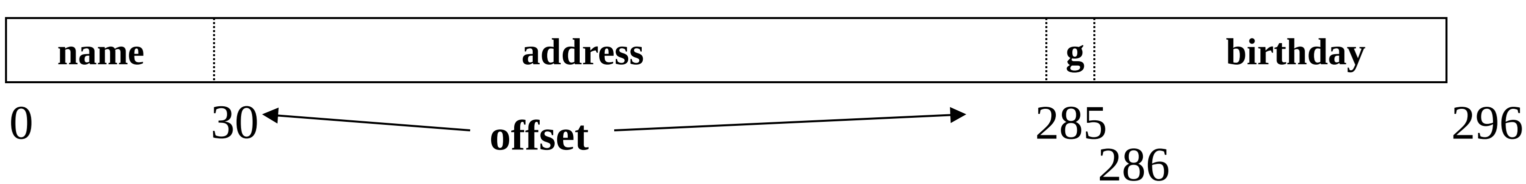
Representing values

- For every SQL data type a representation format is provided, for example:
  - Fixed-length strings: CHAR(n)
    - We use n bytes, possibly using a special character for values shorter than n
    - Example: if A is typed CHAR(5), ‘cat’ is stored as cat⊥⊥
  - Variable-length strings: VARCHAR(n)
    - We use m+p bytes, with m (≤ n) bytes used to store actual characters and p bytes to store the value of m (for n ≤ 254, p = 1)
    - Example: if A is typed VARCHAR(10), ‘cat’ is stored in 4 bytes as 3cat
  - DATE and TIME are usually represented as fixed-length strings
    - DATE: 10 characters YYYY-MM-DD; TIME: 8 characters HH:MM:SS
  - Enumerated types: we exploit an integer coding
    - Example: week = {SUN, MON, TUE, …, SAT} a single byte is needed
    - SUN: 00000001, MON: 00000010, TUE: 00000011, …

Fixed-length records

- For every record type in the DB, we should define a (physical) schema allowing to correctly interpret the meaning of each byte composing the record
- The simplest case arises (clearly) when all records have fixed length, since (besides logical information), we simply need to specify the order in which the attributes are stored in the records (if this is different from the default)

```
CREATE TABLE MovieStar (
name       CHAR(30) PRIMARY KEY,
address    CHAR(255),
gender     CHAR(1),
birthdate DATE )
```

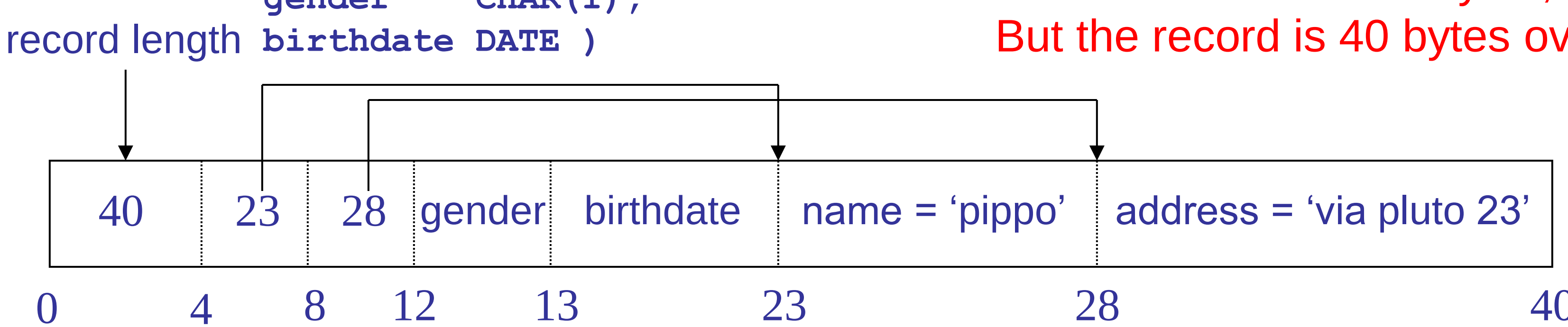


Variable-length records

- In case of records with a variable length, we have some alternatives, also considering problems introduced by updates, which can modify the length of some attributes (and thus of the record)
- A typical solution consists in storing all fixed-length attributes first, followed by all variable length attributes; for every variable length attribute we have a "prefix pointer", containing the address of the first byte of the field

```
CREATE TABLE MovieStar (
name       VARCHAR(30) PRIMARY KEY,
address    VARCHAR(255),
gender     CHAR(1),
birthdate  DATE )
```

Size of the data is 28 bytes,
But the record is 40 bytes overall



Record Header

- In general, every record includes a header that, besides the record length, can contain:
  - The ID of the relation of the record
  - The ID of the record in the DB (univocal)
  - A timestamp, indicating when the record was entered or last modified
- The specific format of the header is clearly different from DBMS to DBMS

Organizing records into pages

- Normally, the size of a record is (very) lower of that of a page
  - Special techniques exist (but we will not see them) to manage cases of “long tuples”, whose length exceeds the page size
- For the case of fixed-length records, the page organization could look like this:

Page header	record 1	record 2	…	record n	

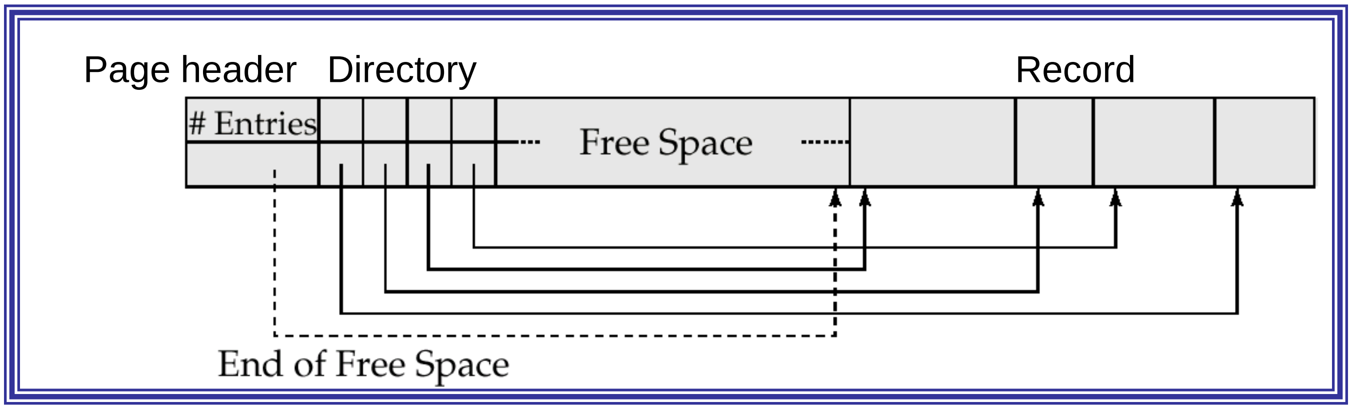
- Page header stores information like:
  - ID of the page in the DB
  - A timestamp indicating when the page was last modified
  - The ID of the relation of the records stored in the page, etc.
- Normally, a record is completely contained in a page
  - Some wasted space might exist

A simple example

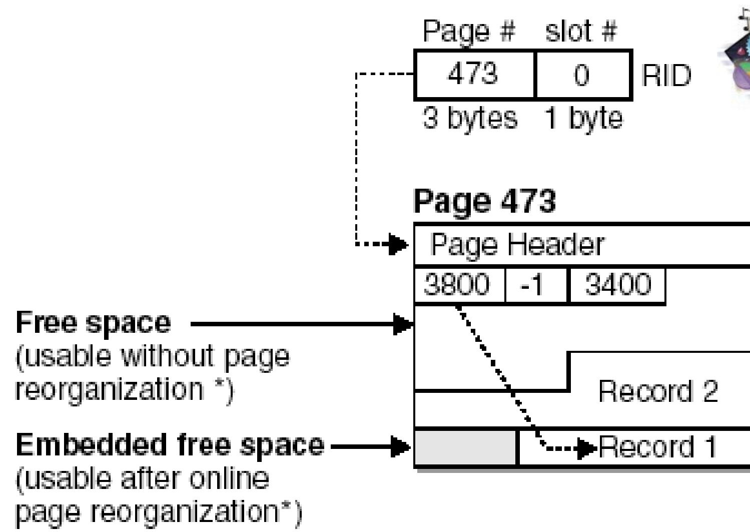
- In the previous example, with fixed-length records of 296 bytes, we suppose a page size of P = 4 KB = 4096 byte
- Supposing that the page header requires 12 bytes, 4084 bytes are available for data
- Thus, a page can contain as much as 13 records ($\lfloor 4084/296 \rfloor$)
  - In every page, at least 236 byte will always remain unused
- ... if the MovieStar relation contains 10000 tuples, at least 770 pages are needed to store it ($\lceil 10000/13 \rceil$)
- ... if a page read from disk requires 10 ms, reading all the tuples would require about 7.7 seconds

Organizing pages into slots

- The typical format of a page in a DBMS looks like this

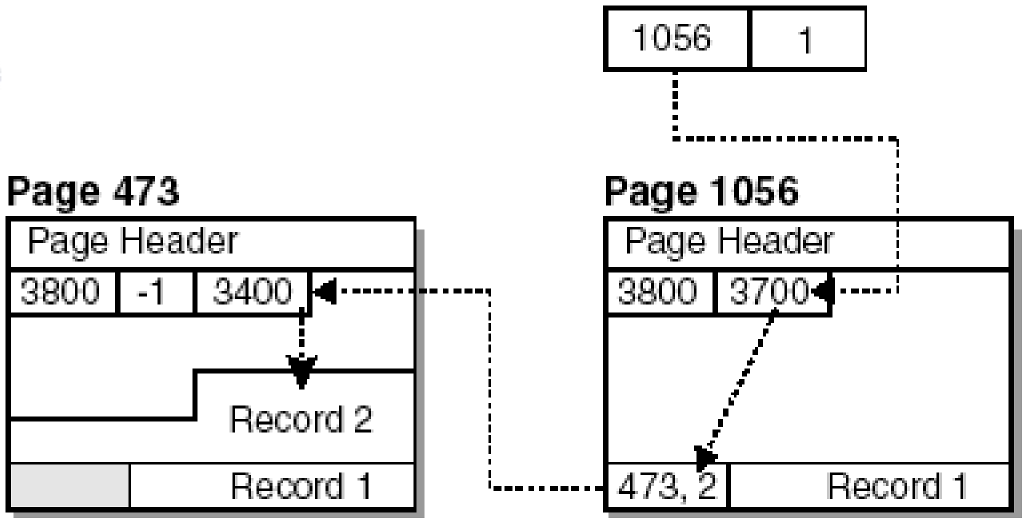


- *The directory contains a pointer for every record in the page*
- *This way, the record identifier (**RID**) in the DB consists of a pair:*
  - ***PID**: page identifier*
  - ***Slot**: location within the directory*
- *This allows us to both quickly locate a record and reallocating it within the page without changing its RID*



Overflow records

- If an update increases the record size and no space is left on the page, the record is moved to another page (it "overflows")
- The record RID, however, does not change, but a level of indirection is introduced
- Having several overflow records clearly degrades performance, thus a periodical file reorganization is needed



Reading and writing pages

- Reading a single tuple requires bringing the corresponding page into main memory, in a DBMS-managed area called buffer pool
- Every buffer in the pool could host the copy of a disk page
- Managing the buffer pool is fundamental for performance, and a DBMS module is devoted to this, the Buffer Manager (BM)
- The BM is called also when writing to disk, because we should update a modified page on disk
- The BM plays a fundamental role, as we shall see, in transaction management, in order to guarantee the DB integrity when faults are present

In DB2, we could define several buffer pools, but every tablespace should be associated to a single buffer pool

The Buffer Manager

- When a page is requested, the Buffer Manager (BM) operates as follows:
  - If the page is already contained in a buffer, the buffer address is returned to the caller
  - If the page is not already in main memory:
    - The BM selects a buffer for the requested page
    - If such buffer contains another page, this is written on disk, only if it was modified an no one is using it
    - At this point, the BM can read the page, copying it in the chosen buffer, thus replacing the previous page

Interface of the Buffer Manager

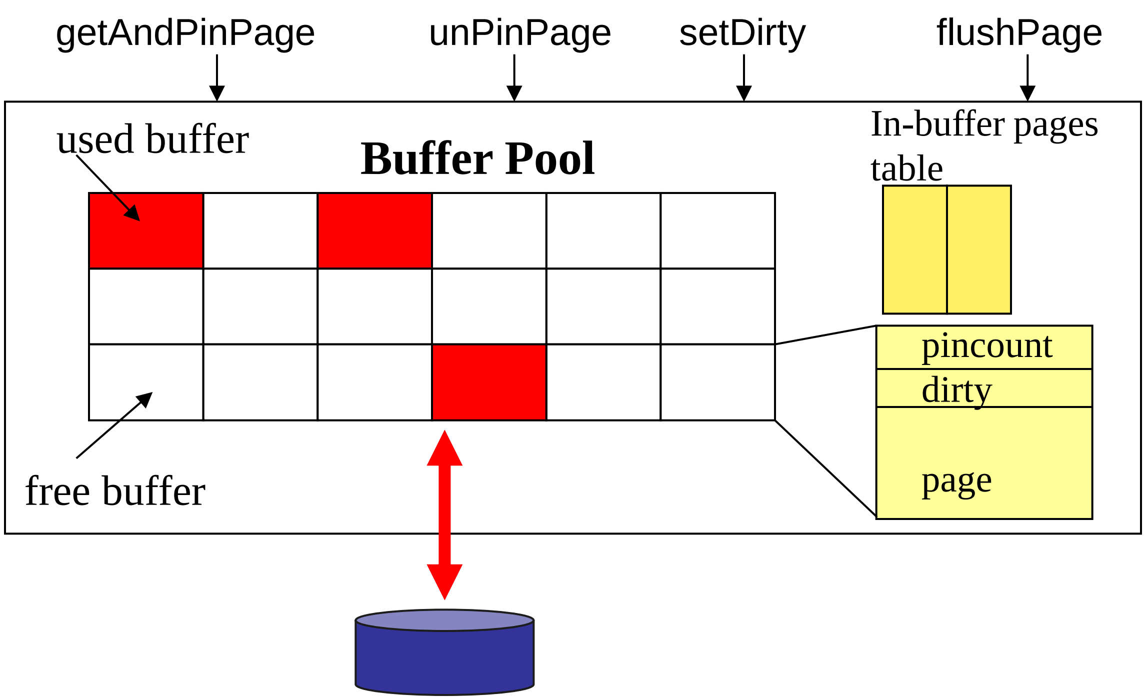
- The interface offered by the BM to other DBMS modules includes four basic methods:

getAndPinPage: requests the page to the BM an puts a "pin" on it, indicating that it is used

unPinPage: releases the page, clearing a pin

setDirty: indicates that the page has been modified, it is now "dirty"

flushPage: forces the to be written on disk, making it "clean"



Buffer replacement policies

- For operating systems, a common policy for choosing the page to be replaced is the LRU (Least Recently Used), thus the page chosen is the one which has been unused for the most time
- In DBMSs LRU is often a bad choice, since for some queries the "access pattern" to data is known, and it could thus be used to operate more accurate choices, also able to greatly improve performance
- The hit ratio, or the ratio of requests needing no I/O operation, is a synthetic indicator of the quality of a replacement policy

  Example: we will see that join algorithms exist scanning tuples of a relation for N times.
  In this case, the best policy would be a MRU (Most Recently Used), thus replacing the most recently used page!
- … and this is another reason why DBMSs do not use (all) services provided by OSs …

File organization

- The way records are organized within the file affects both efficiency of data access and storage occupation
- In the following, we will see some basic organizations, namely:
  - Heap file
  - Sequential file

  … and we will evaluate them according to some typical operations
- For the sake of simplicity, we will:
  - Consider fixed-length records
  - Evaluate "costs" as number of I/O operations, assuming that every page request results in a single I/O operation
- In order to evaluate costs we however need some basic information…

SQL catalogs statistics

- Every DBMS keeps some **catalogs**, that is relations describing the DB at both the logical and physical levels
- Catalogs which are interesting for us at this moment are those reposting statistic information about relations, in particular:

SQL catalog	SQL attribute	Description	Symbol
SYSSTAT.TABLES	CARD	Number of tuples in the relation	NR or NR(table)
SYSSTAT.TABLES	NPAGES	Number of pages storing the relation	NP or NP(table)
SYSSTAT.COLUMNS	COLCARD	Number of distinct values for the attribute	NK or NK(attribute)
SYSSTAT.COLUMNS	LOW2KEY	Second-lowest value	LK or LK(attribute)
SYSSTAT.COLUMNS	HIGH2KEY	Second-highest value	HK or HK(attribute)

Cost model

- We are interested in estimating the cost of the following operations:
  - Search by key
  - Range search
  - Insertion of a new record
  - Deletion of a record
  - Updating the value of a record attribute (key/non key)
- We will assume as base cost the number of accesses to secondary storage
  - Simplifying hypothesis (why?)

Heap file

- Also known as serial file, is the simplest one since it is characterized by appending new records at the end of the file
- If some record is deleted, in order to be able to re-use its space without reading the whole file, a mechanism is needed to quickly locate free space

header				
record 0	H. Fonda	LA	male	1-1-11
record 1				
record 2	Basinger	Chicago	female	3-3-33
record 3				
record 4	Baldwin	NYC	male	2-2-22

Page management: linked list

- The first option is to keep two double linked lists:
  - a list for full pages
  - a list for pages with some free space
- Typically, with variable-length records, almost all pages will contain some free space
  - In order to find a page with enough space to contain a new record, we could be forced to scan the whole list
- Alternative solution…

Page management: directory

- A second option is to keep a page directory :
  - The directory itself is organized as a linked list of pages
  - Every directory entry identifies a page in the file and can report the free space for each page
  - In order to find a page with enough space to contain a new record, we can simply scan the directory (normally, much smaller than the file)
  - Con: larger file size

The DB2 solution

- Data pages grouped into extents
- A page every 500 contains a Free Space Control Record (FSCR), with a directory of free space within the next 500 pages (until the next FSCR)
- The page size (4/8/16/32 KB) can be specified when the tablespace is created
  - Larger size for sequential access
  - Smaller size for random access

Heap file: operations and costs

- The table summarizes the costs for basic operations:

Operation	Description	Cost
Search by key	Search is performed by sequentially scanning all the pages	NP/2 average NP maximum NP if non existing
Range search	We have to look all the pages, anyway	NP
Insertion	We append at the end of the file	2
Deletion	Only a record is deleted	C(search) + 1
Update	Only a record is updated	C(search) + 1

Sequential file

- In a sequential file, records are kept sorted according to the values of a given attribute (or of a combination of attributes)
- Clearly, now insertion should be performed by preserving the sort order
  - Usually, some free space is left within each page (otherwise we can tolerate overflow records and reorganize the file periodically)

Brighton	A-127	750
Downtown	A-101	500
Downtown	A-101	600
Mianus	A-215	700
Perryridge	A-102	400
Perryridge	A-201	900

Sequential file: operations and costs

- For simplicity, we consider that the file is sorted according to values of the primary key (or another key)

Operation	Description	Cost
Search by key	We use binary search to locate the page containing the record	$\lceil \log_2(NP) \rceil$
Range search	We only read pages containing key values in the range [L,H]	$C(search) - 1 + \frac{(H - L) * NP}{HK - LK}$
Insertion	We suppose there is enough free space	C(search) + 1
Deletion	Only a record is deleted	C(search) + 1
Update	Only a record is updated	C(search) + 1