Normalization, Generated Keys, Disks

CS634
Lecture 3

Slides based on "Database Management Systems" 3rd ed, Ramakrishnan and Gehrke

Normalization in practice

The text has only one example, pg. 640: books, customers, orders
And it’s already normalized!

But often actual tables in use are not normalized and should be

Normalization is important for operational databases, not data warehouses, which are often denormalized.

That’s because normalization helps keep data integrity across changes to the data.

Normalization in practice

Example, pg. 174 (ex. 5-3) and in createdb.sql:

create table flights(
flno **int primary key,**
origin **varchar(20) not null,**
destination **varchar(20) not null,**
distance **int,**
departs **varchar(20),**
arrives **varchar(20),**
price **decimal(7,2)**);

What's distance? it's the distance between the origin and destination airports, so the FD: origin, destination → distance lies in the table and distance is non-key, so the table doesn't qualify as 3NF.

Normalization in practice

To normalize we create another table

create table links(
origin **varchar(20),**
destination **varchar(20) ,**
distance **int,**
primary key(origin,destination)
);
create table flights(
flno **int primary key,**
origin **varchar(20) not null,**
destination **varchar(20) not null,**
departs **varchar(20),**
arrives **varchar(20),**
price **decimal(7,2),**
foreign key (origin, destination) references links
);

Why do we care?

This lack of normalization has well-known problems: pg. 607

Delete anomaly:
Delete all flights from Boston to Ithaca
End up losing distance information on this link

Insert anomaly:
Add a flight from Boston to Ithaca
Need to check if the distance is consistent with other rows

Update anomaly:
Correct the distance: need to check for all the cases.

As a consultant to database-using groups, need to keep an eye on table designs and possibly point out potential problems, esp. early, before the group has invested a lot of development work in their design.

Primary Key Generation

We have seen that entity tables often have an "id" attribute, usually of type integer, that serves as the PK.

In createdb.sql:
student, faculty entities: int PKs
class entity: varchar PK (exception!)
enrolled: a relationship, two-key PK
emp, dept: entities, with int PKs
works: a relationship, two-key PK
flights, aircraft, employees: entities, int PK
…
Reserves: an entity we decided, PK: (sid, bid, day) (exception!)

Primary Key Generation

We can assign ids outside the database, and create a load file like the one we see in our tables directory:

Parts.txt:
1,Left Handed Bacon Stretcher Cover,Red
2,Smoke Shifter End,Black
3,Acme Widget Washer,Red
4,Acme Widget Washer,Silver
5,I Brake for Crop Circles Sticker,Translucent
6,Anti-Gravity Turbine Generator,Cyan
7,Anti-Gravity Turbine Generator,Magenta
…
create table parts(pid int primary key, pname varchar(40) not null, color varchar(15), unique(pname, color));

Primary Keys and Natural Keys

Parts.txt:
1,Left Handed Bacon Stretcher Cover,Red
2,Smoke Shifter End,Black
…
create table parts(pid int primary key, pname varchar(40) not null, color varchar(15), unique(pname, color));

Here pid is an arbitrary key, with no information about the part.
The “natural key” here is shown by the unique constraint.
The natural key is a key made up of meaningful attributes.

Primary Keys and Natural Keys

create table class(name varchar(40) primary key, meets_at varchar(20),
room varchar(10), fid int,
foreign key(fid) references faculty(fid));

Class.txt:
Data Structures,MWF 10,R128,489456522
Database Systems,MWF 12:30-1:45,1320 DCL,142519864
Operating System Design,TuTh 12-1:20,20 AVW,489456522
…

Here the PK is a natural key.

If we decide to change the name of a course, the PK has to change, and any FKs referring to it need to change.

Primary Keys and Natural Keys

- With arbitrary integer values as PKs, if we decide to change the natural key, it's easy and doesn't cause other updates.
- Also, we often join on PKs, and integer ids are smaller and thus faster than natural keys, which are usually varchars.
- Thus arbitrary-integer-valued PKs are in wide use…

Generated Primary Keys

- After the initial load, we may want to insert another part: how should we assign an id then? Would be great to have the DB do it…

- The database can generate new integer values for PKs by mechanisms that, unfortunately, are not covered in SQL-92:
  - Auto-increment in mysql, MS SQL Server, DB2
  - Sequences in Oracle, DB2

- These are covered in SQL 2003, but that was too late for real standardization across DB products

Generated Primary Keys

- Auto-increment: just add a keyword (auto_increment in mysql) to the column spec in the create table, in mysql, etc.

- Sequence (Oracle, etc.): create a sequence, which is a database object but not a table, then use it to generate a new value as needed
  - The create table has no special keywords in this case.

- In homework 1, you'll look up the details on this and use it for loading a table.

Generated Primary Keys: Oracle

Example from http://www.techonthenet.com/oracle/sequences.php

```
CREATE SEQUENCE supplier_seq
 START WITH 1 INCREMENT BY 1;

SELECT supplier_seq.nextval FROM dual;  --returns 1 (just testing)
SELECT supplier_seq.nextval FROM dual;  --returns 2

INSERT INTO suppliers (supplier_id, supplier_name)
VALUES (supplier_seq.NEXTVAL, 'Kraft Foods');
…
DROP SEQUENCE supplier_seq;
```

For sqlldr with sequence column, see Case Study 3 in https://docs.oracle.com/cd/B12037_01/server.101/b10825/ldr_cases.htm#i1006494

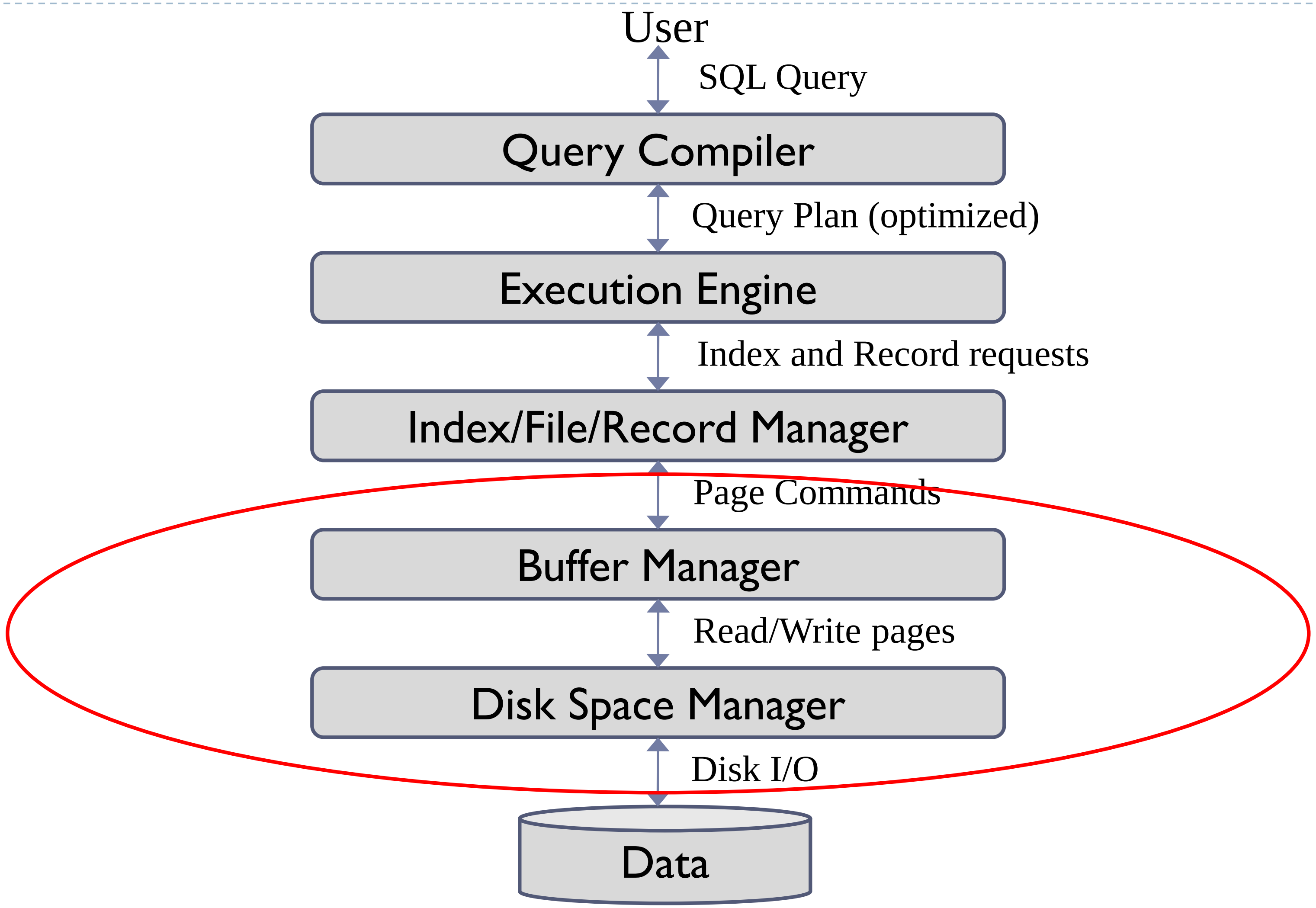
On to the core of this course

Chapters 8-11   Storage and Indexing
Chapters 12-15 Query Processing.
Chapters 16-18 Transactions and Recovery

Storing Data: Disks and Files: Chapter 9

Slides based on “Database Management Systems” 3rd ed, Ramakrishnan and Gehrke

Architecture of a DBMS



A first course in database systems, 3rd ed, Ullman and Widom

Disks and Files

- DBMS stores information on disks (hard disks and SSD)
- This has major implications for DBMS design
  - READ: transfer data from disk to main memory (RAM)
  - WRITE: transfer data from RAM to disk
  - Both are high-cost operations, relative to in-memory operations, so must be planned carefully!

Why Not Store Everything in Main Memory?

- Capacity is limited on a system: max RAM is 64GB on many machines, disk easily holds many TBs
- *Costs too much.*

RAM ~ $10/GB (vs. $30/MB in 1995) graph 1957-2017

Disk ~ $0.02/GB (vs. $200/GB in 1996)

RAM is 500x more expensive! (vs. 7000x in 95-96)

- *Main memory is volatile.*
  - We want data to be saved long-term.
- Newer contender: SSD solid-state disk, ~$.30/GB(2017), still much more expensive (~10x) than hard disk.

Tape is still in use, mainly for backup

- Typical Classic DB storage hierarchy:
  - Main memory (RAM) for currently used data.
  - Disk for the main database (secondary storage).
  - Tapes for archiving older versions of the data (tertiary storage).

  - Disk ~ $0.02/GB
  - Tape ~ $.01/GB
    - Unlike disks, tapes do not support random access
    - A tape cartridge can hold 6.25 TB, easily moved off-site

  - Another way: backup data to the cloud, but much slower.

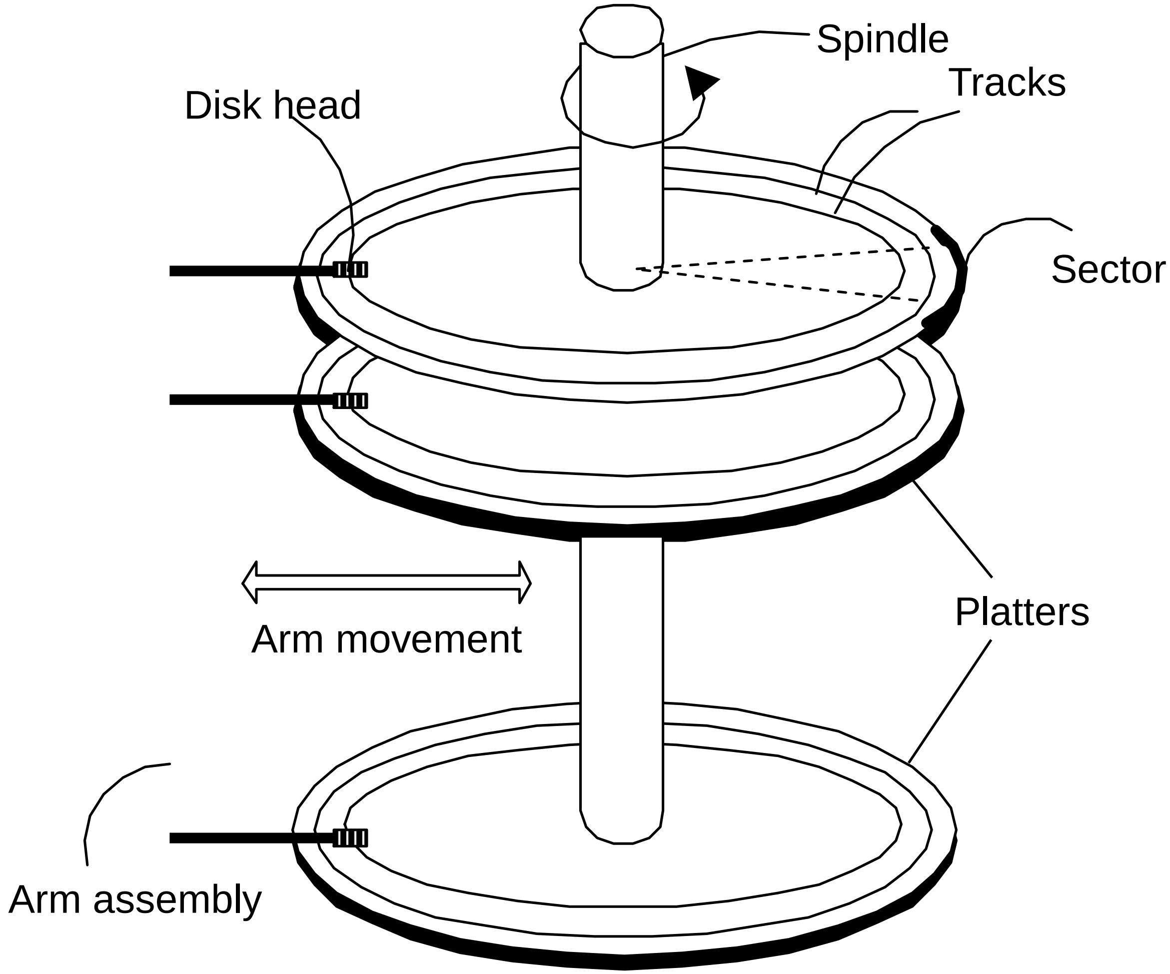
Persistent storage: HDD vs. SSD

- HDD typical values:
  - 100 io/s random reads/writes
  - 100 MB/s sequential read or write
  - Means 100*8KB/s = 800 KB/s = .8MB/s using 8KB random reads
  - That's less than 1% of sequential reading speed!
- SSD typical values: 5x faster sequential i/o, but 10x cost/GB.
  - 500 MB/s sequential read, also write on new SSD
  - Reads can be random without penalty (unless really tiny)
  - Writes slow down on full disk (needs to erase before write)
  - 8KB ios: (500MB/s)/8KB = 64K io/s Wow!
  - See even higher numbers in product literature, but need many i/os in progress to do that.

Disks (hard disk)

- Unlike RAM or SSD, time to retrieve a disk block varies depending upon location on disk.
  - Relative placement of pages on disk has major impact on DBMS performance!

Components of a Disk



Components of a Disk

- The platters spin constantly while in use
- The arm assembly is moved in or out to position a head on a desired track. Tracks under heads make a *cylinder* (imaginary!).
- Only one head reads/writes at any one time.
- *Block size* is a multiple of *sector size* (which is fixed at 512 bytes). Typical 4KB, 8KB, for filesystems, larger for data warehousing: 256KB, 1MB

Accessing a Disk Block

- Time to access (read/write) a disk block:
  - *seek time* (moving arms to position disk head on track)
  - *rotational delay* (waiting for block to rotate under head)
  - *transfer time* (actually moving data to/from disk surface)
- Seek time and rotational delay dominate.
  - Seek time varies from about 1 to 20ms (typical <= 4ms)
  - Rotational delay varies from 0 to 10ms, average 4ms for 7200 RPM (60/7200 = .008s/rev = 8ms/rev, half on average)
  - Transfer time is under 1ms per 4KB page, rate~100M/s, so 10 ms for 1MB, about same as seek+rotational delay.

- Key to lower I/O cost: reduce seek/rotation delays!
- One idea: use 1MB transfers, but not flexible enough for all cases (i.e. small tables)

Arranging Pages on Disk

- `*Next*' block concept:
  - blocks on same track, followed by
  - blocks on same cylinder, followed by
  - blocks on adjacent cylinder
- Blocks that are accessed together frequently should be sequentially on disk (by `next'), to minimize access time
- For a sequential scan, ***pre-fetching*** several pages at a time is a big win!

Physical Address on Disk

- To locate a block on disk, the disk itself uses CHS address
  - Cylinder address
    - Where to position the head, i.e., "seek" movement
  - Head address
    - Which head to activate
    - Identifies the platter and side, hence the track, since cylinder is already known
  - Sector address
    - The address of first sector in the block
    - Wait until disk rotates in the proper position
- But current disks (SCSI, SAS, etc.) accept LBNs, logical block numbers, one number per block across whole disk in "next" order. See http://en.wikipedia.org/wiki/Logical_block_addressing

RAID

- Redundant Array of Independent Disks
  - Arrangement of several disks that gives abstraction of a single, large disk, with LBNs across the whole thing.
- Improves performance
  - Data is carefully spread over several disks: striping
  - Requests for sequence of blocks answered by several disks
  - Disk transfer bandwidth is effectively aggregated
- Increases reliability
  - Redundant information stored to recover from disk crashes
  - Mirroring is simplest scheme
  - Parity schemes: data disks and check disks

RAID Levels

- Level 0: Striping but no redundancy
  - Maximum transfer rate = aggregate bandwidth
  - Stripe size can be many blocks, example 256KB
  - With *N* data disks, read/write bandwidth improves up to *N* times
- Level 1: Mirroring
  - Each data disk has a mirror image (check disk)
  - Parallel reads possible, but a write involves both disks
  - So max read transfer rate = aggregate bandwidth, but write rate only half
- Level 1+0: Striping and Mirroring (AKA RAID 10)
  - Need minimum of 4 disks, making two mirrored disks to stripe
  - Maximum read transfer rate = aggregate bandwidth
  - Write still involves two disks
  - More popular now that disk is so inexpensive, for operational DBs
  - Faster recovery from failed disk than RAID 5 or 50
- Level 0+1: mirror a striped set. Not a great idea.

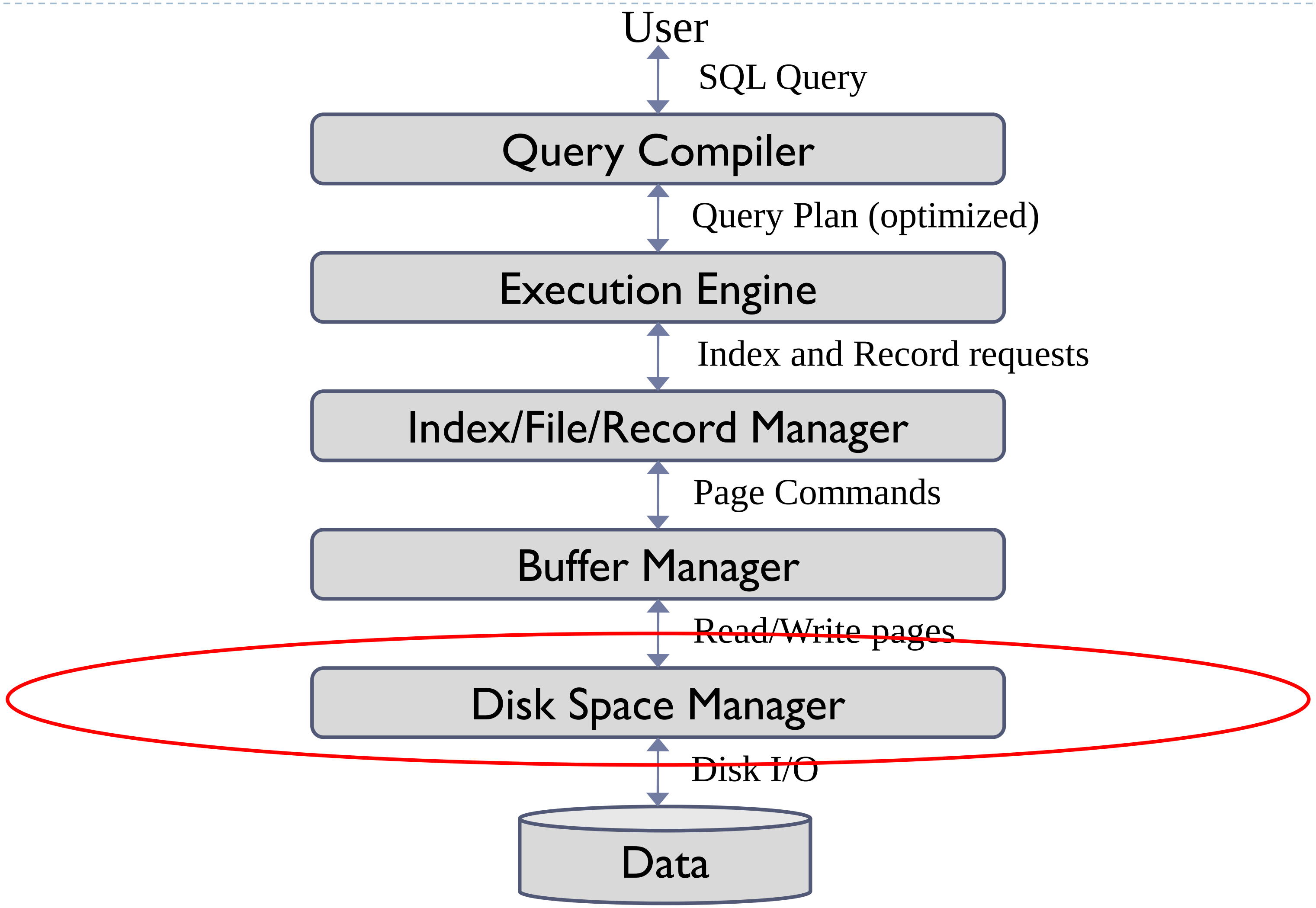
RAID Levels (Contd.)

- Level 4: Block-Interleaved Parity (not important in itself)
  - Striping Unit: One disk block
  - There are multiple data disks (*N*), single check disk
  - Check disk block = XOR of corresponding data disk blocks
  - Can reconstruct one failed disk
  - Read bandwidth is up to *N* times higher than single disk
  - Writes involve modified block and check disk
  - RAID-3 is similar in concept, but interleaving done at bit level

- Level 5: Block-Interleaved Distributed Parity (in wide use)
  - In RAID-4, check disk writes represent bottleneck
  - In RAID-5, parity blocks are distributed over all disks
  - Every disk acts as data disk for some blocks, and check disk for other blocks
  - Most popular of the higher RAID levels (over 1+0).
  - Level 50: stripe across 2 or more RAID 5 sets

- Level 6: More redundancy, can handle two failed disks
  - Level 60: stripe across 2 or more RAID 6 sets

Architecture of a DBMS

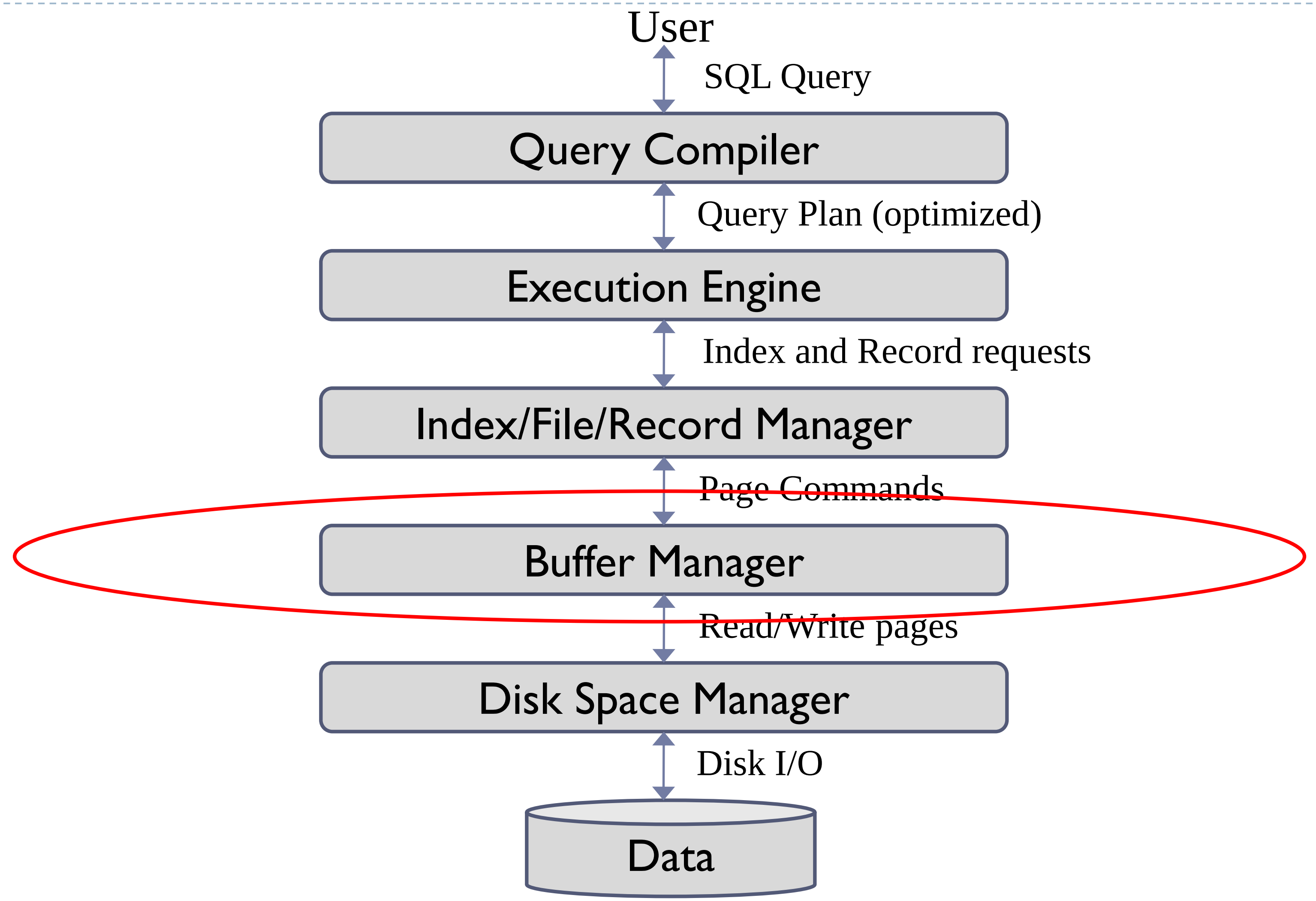


A first course in database systems, 3rd ed, Ullman and Widom

Disk Space Manager

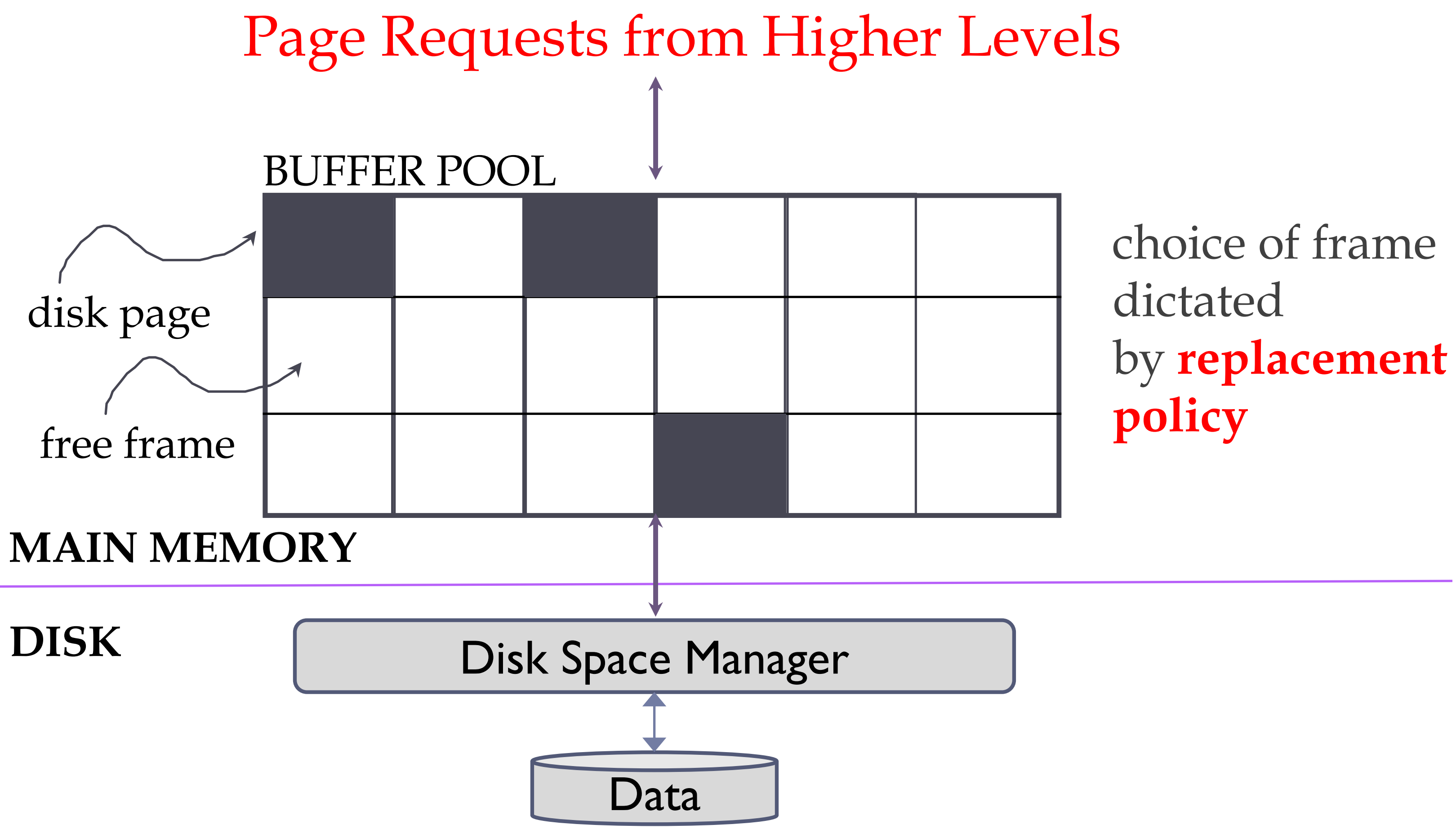
- Lowest layer of DBMS, manages space on disk
  - Provides abstraction of data as collection of pages
- Higher levels call upon this layer to:
  - allocate/de-allocate a page on disk
  - read/write a page
  - keep track of free space on disk
- Tracking free blocks on disk
  - Linked list or bitmap (latter can identify contiguous regions)
- Must support request for allocating *sequence* of pages
  - Pages must be allocated according to "next-block" concept

Architecture of a DBMS



A first course in database systems, 3rd ed, Ullman and Widom

Buffer Management



- *A mapping table of <frame#, pageid> pairs is maintained*

Buffer Pool Sizing

- As DBA, you are responsible for sizing the buffer pool.
- Ideally, you want to have a big enough buffer pool to hold all the commonly-accessed data.
- Many databases are delivered with very small buffer pools, say 200MB. You need to fix this before serious use.
  - Mysql is delivered with a setting to allow filesystem buffering, so data ends up in both the buffer pool and the file system buffer.
- If it's too small, pages will be read and reread, and some activities may have to wait for space in the buffer pool.
- If the server is only a database server (for large data), use most of its main memory for this, say 80%.
- If the server is also a web server, say, allocate half the memory to the DB, quarter to the web server.

When a Page is Requested …

- If requested page is not in pool:
  - Choose a destination frame
  - Read requested page into chosen frame
  - *Pin* the page and return its address
  - a *pin count* is used to track how many requests a page has
  - Requestor must *unpin* it, and set the *dirty* bit if modified
- If no frame is currently free:
  - Choose a frame for *replacement among those with pin count* = 0
  - If frame is dirty, write it to disk
- If requests can be predicted (e.g., sequential scans) pages can be pre-fetched several pages at a time!

Buffer Replacement Policy

- Frame is chosen for replacement by a *replacement policy*
  - Least-recently-used (LRU), MRU, Clock, FIFO, random
  - LRU-2 could be used (O'Neil et al)
  - See https://en.wikipedia.org/wiki/Page_replacement_algorithm
- Policy can have big impact on number of required I/O's
  - depending on the page *access pattern*
- Sequential flooding
  - worst-case situation caused when using LRU with repeated sequential scans if #buffer frames < #pages in scan
  - each page request causes an I/O
  - MRU much better in this situation, LRU-2 is also better
  - no single policy is best for all access patterns

DBMS vs OS Disk/Buffer Management

- DBMS have specific needs and access characteristics
- And it has the resources to save more info than an OS is allowed to do. OS is required to be lean and mean.
- DBMS do not rely just on OS because
  - OS does not support files spanning several devices
  - Special physical write functionality required (recovery)
  - DBMS can keep track of frequent access patterns (e.g., sequential scans) can lead to more efficient optimization
    - Pre-fetching, smart page replacement
- DBMS can use files as disk resource, take over their i/o characteristics. Best to build database files on "brand new" disk: reinitialize partition if necessary.

Next time

- Examples of RAID systems
- Examples of SSD-HDD hybrid systems