

# Database Applications (15-415)

SQL-Part III &

Storing Data: Disks and Files (Intro)

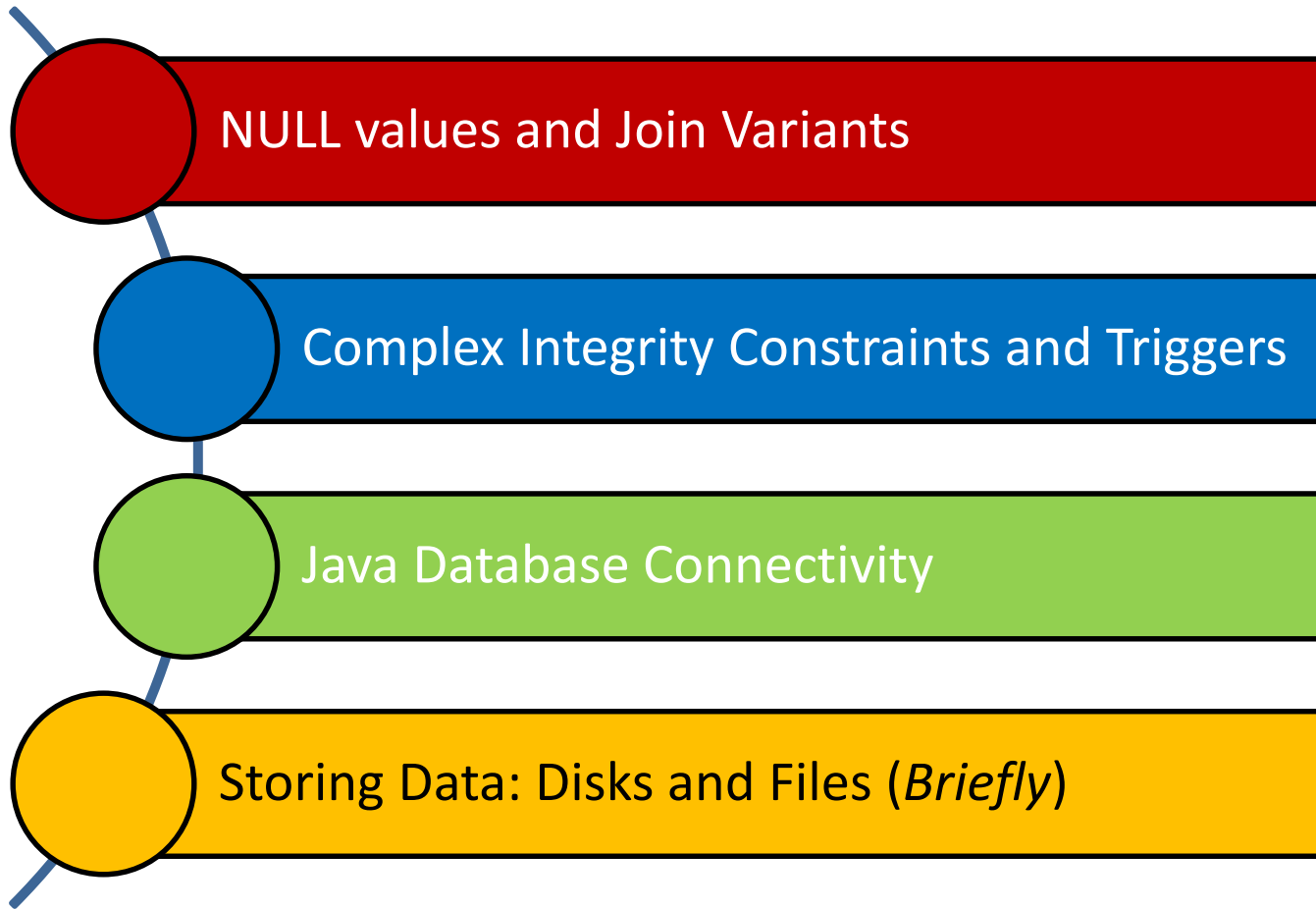
Lecture 9, February 8, 2015

Mohammad Hammoud

# Today...

- Last Session:
  - Standard Query Language (SQL)- Part II
- Today's Session:
  - Standard Query Language (SQL)- Part III
  - Storing Data: Disks and Files- Some Basics (*if time allows*)
- Announcements:
  - PS2 is due today by midnight
  - Quiz I is on Thursday Feb 12, 2015 (all material included except today's lecture)
  - No class on Tuesday Feb 10 due to Qatar National Sports Day
  - Project I is due on Tuesday Feb 17 by midnight

# Outline



# NULL Values

- Column values can be *unknown* (e.g., a sailor may not yet have a rating assigned)
- Column values may be *inapplicable* (e.g., a maiden-name column for men!)
- **NULL** values can be used in such situations
- However, NULL values complicate many issues!
  - Comparing NULL to a valid value returns unknown
  - Comparing NULL to a NULL returns unknown

# NULL Values

- Considering a row with rating = NULL and age = 20; How does it compare with the following Boolean expressions?
  - Rating = 8 OR age < 40 → TRUE
  - Rating = 8 AND age < 40 → unknown
- In general, what about?
  - NOT unknown → unknown
  - True OR unknown → True
  - False OR unknown → unknown
  - False AND unknown → False
  - True AND unknown → unknown

# NULL Values

- Considering a row with rating = NULL and age = 20; How does it compare with the following Boolean expressions?
  - Rating = 8 OR age < 40 → TRUE
  - Rating = 8 AND age < 40 → unknown

- In general, what about?

- NOT unknown → unknown

- True OR unknown → True

***Three-Valued Logic!***

- False OR unknown → unknown

- False AND unknown → False

- True AND unknown → unknown

# Inner Joins

- Tuples of a relation that do not match some row in another relation (according to a join condition  $c$ ) do not appear in the result
  - Such a join is referred to as “**Inner Join**” (*so far, all inner joins*)

```
select ssn, c-name  
from takes, class  
where takes.c-id = class.c-id
```

**Equivalently:**

```
select ssn, c-name  
from takes join class on takes.c-id = class.c-id
```

# An Example of Inner Joins

- Find all SSN(s) taking course s.e.

TAKES		
<u>SSN</u>	<u>c-id</u>	grade
123	15-413	A
234	15-413	B

CLASS		
<u>c-id</u>	c-name	units
15-413	s.e.	2
15-412	o.s.	2

<u>SSN</u>	<u>c-name</u>
123	s.e
234	s.e

**o.s.: gone!**



# Outer Joins

- Tuples of a relation that do not match some row in another relation (according to a join condition  $c$ ) can still appear exactly once in the result
  - Such a join is referred to as “Outer Join”
  - Result columns will be assigned NULL values

```
select ssn, c-name  
from takes outer join class  
on takes.c-id=class.c-id
```

# An Example of Outer Joins

- Find all SSN(s) taking course s.e.

TAKES		
<u>SSN</u>	<u>c-id</u>	grade
123	15-413	A
234	15-413	B

CLASS		
<u>c-id</u>	c-name	units
15-413	s.e.	2
15-412	o.s.	2

<u>SSN</u>	<u>c-name</u>
123	s.e
234	s.e.
null	o.s.



# Joins

- The general SQL syntax:

```
select [column list]
from table_name
[inner | {left | right | full} outer] join
table_name
on qualification_list
```

Outer Join Type	Description
Left Outer Join	<i>A</i> rows without a matching <i>B</i> row appear in the result
Right Outer Join	<i>B</i> rows without a matching <i>A</i> row appear in the result
Full Outer Join	Both <i>A</i> and <i>B</i> rows without a match appear in the result

# Outline

NULL values and Join Variants

Complex Integrity Constraints and Triggers ✓

Java Database Connectivity

Storing Data: Disks and Files (*Briefly*)

# Integrity Constraints- A Review

- An Integrity Constraint (IC) describes conditions that every *legal instance* of a relation must satisfy
- Inserts/deletes/updates that violate IC's are disallowed
- ICs can be used to:
  - Ensure application semantics (e.g., *sid* is a key)
  - Prevent inconsistencies (e.g., *sname* has to be a string, *age* must be  $< 20$ )

# Types of Integrity Constraints- A Review

- IC types:
  - Domain constraints
  - Primary key constraints
  - Foreign key constraints
  - General constraints
    - Useful when more general ICs than keys are involved
    - Can be specified over a single table and across tables

# General Constraints Over a Single Table

- Complex constraints over a single table can be defined using **CHECK conditional-expression**

```
CREATE TABLE Sailors (sid INTEGER,  
                      sname CHAR (10),  
                      rating INTEGER,  
                      age REAL,  
                      PRIMARY KEY (sid),  
                      CHECK (rating >= 1 AND rating <= 10))
```

A domain constraint

A primary key constraint

A general constraint

# General Constraints Over a Single Table

- How can we *enforce* that “Interlake” boats cannot be reserved?

```
CREATE TABLE Reserves (sid INTEGER,  
                        bid INTEGER,  
                        day DATE,  
                        FOREIGN KEY (sid) REFERENCES Sailors,  
                        FOREIGN KEY (bid) REFERENCES Boats,  
                        CONSTRAINT noInterlakeRes,  
                        CHECK ('Interlake' NOT IN  
                              (SELECT B.bname  
                               FROM Boats B  
                               WHERE B.bid = Reserves.bid)))
```

A foreign key constraint



# General Constraints Across Tables-

## Motivation

- How can we *enforce* that the number of boats plus the number of sailors should not exceed 100?

```
CREATE TABLE Sailors (sid INTEGER,  
                        sname CHAR (10),  
                        rating INTEGER,  
                        age REAL,  
                        PRIMARY KEY (sid),  
                        CHECK (rating >= 1 AND rating <= 10)  
                        CHECK ( ((SELECT COUNT (S.sid)  
                                FROM Sailors S) +  
                                (SELECT COUNT (B.bid)  
                                FROM Boats B)) < 100))
```

What if the Sailors table is empty and we insert more than 100 rows into Boats?

# General Constraints Across Tables- Assertions

- How can we *enforce* that the number of boats plus the number of sailors should not exceed 100?

```
CREATE ASSERTION smallClub  
CHECK  
( (SELECT COUNT (S.sid) FROM Sailors S)  
+ (SELECT COUNT (B.bid) FROM Boats B) < 100 )
```

ASSERTION is the right solution; not associated with either table!

# New Domains

- Users can define new domains using the CREATE DOMAIN statement

```
CREATE DOMAIN ratingval1 INTEGER DEFAULT 1  
CHECK ( VALUE >= 1 AND VALUE <= 10)
```



```
CREATE DOMAIN ratingval2 INTEGER DEFAULT 1  
CHECK ( VALUE >= 1 AND VALUE <= 20)
```

ratingval1 and ratingval2 CAN be compared!

Domain constraints will be always enforced (also for new domains)!

# Distinct Types

- Users can define new distinct types using the CREATE TYPE statement

```
CREATE TYPE ratingtype1 AS INTEGER
```

```
CREATE TYPE ratingtype2 AS INTEGER
```

ratingtype1 and ratingtype2 CANNOT be compared!

Domain constraints will be always enforced (also for new types)!

# Triggers

- A trigger is a *procedural* code that is automatically executed in response to certain *events* on a particular table or view in a database
- Triggers can be activated either *before* or *after*
  - Insertions
  - Deletions
  - Updates

# A Trigger Example

- Set a timestamp field whenever a row in the takes table is updated

TAKES		
<u>SSN</u>	<u>c-id</u>	grade
123	15-413	A
234	15-413	B

- First: we need to add our timestamp field

```
ALTER TABLE takes  
ADD COLUMN updated TIMESTAMP
```

# A Trigger Example

- Set a timestamp field whenever a row in the takes table is updated

<b>TAKES</b>		
<b><u>SSN</u></b>	<b><u>c-id</u></b>	<b>grade</b>
<b>123</b>	<b>15-413</b>	<b>A</b>
<b>234</b>	<b>15-413</b>	<b>B</b>

- Second: we need to create a function that sets the “updated” column with the current timestamp

```
CREATE FUNCTION update_col()
  BEGIN
    NEW.updated = NOW();
    RETURN NEW;
  END;
```

# A Trigger Example

- Set a timestamp field whenever a row in the takes table is updated

TAKES		
<u>SSN</u>	<u>c-id</u>	grade
123	15-413	A
234	15-413	B

- Third: we need to Invoke `update_col()` when a row in the takes table is updated

A *row-level trigger*;  
otherwise, it will be a  
*statement-level trigger*

```
CREATE TRIGGER update_takes_modtime
  AFTER UPDATE ON takes
  FOR EACH ROW
  EXECUTE PROCEDURE update_col();
```



# Outline

NULL values and Join Variants

Complex Integrity Constraints and Triggers

Java Database Connectivity ✓

Storing Data: Disks and Files (*Briefly*)

# Java Database Connectivity

- SQL commands can be *embedded* in host language programs
- A popular data access technology which provides an API for querying and manipulating data in (any) storage system is called **Java Database Connectivity (JDBC)**
- Direct interactions with a DBMS occurs through a DBMS-specific **driver**
- A driver is a software program that translates JDBC calls into DBMS-specific calls
  - Drivers do not necessarily interact with a DBMS that understands SQL
  - Thus, a DBMS in JDBC's parlance is usually referred to as **data source**

# Establishing a Connection

- With JDBC, a database is represented by a URL
- With PostgreSQL™, this takes one of the following forms:
  - `jdbc:postgresql:database`
  - `jdbc:postgresql://host/database`
  - `jdbc:postgresql://host:port/database`
- To connect to a database, a Connection instance from JDBC can be used

```
Connection db = DriverManager.getConnection(url, username, password);
```

# Establishing a Connection

- A number of additional properties can be used to specify additional driver behavior specific to PostgreSQL™

```
String url = "jdbc:postgresql://localhost/test";
Properties props = new Properties();
props.setProperty("user", "Hammoud");
props.setProperty("password", "secret");
props.setProperty("ssl", "true");
Connection conn = DriverManager.getConnection(url, props);
```

## Equivalently:

```
String url = "jdbc:postgresql://localhost/test?user=Hammoud&password=secret&ssl=true";
Connection conn = DriverManager.getConnection(url);
```

# Establishing a Connection

- Putting it all together, you can create the following function:

```
public Connection getConnection() throws SQLException {  
  
    String url = "jdbc:postgresql://localhost/test";  
    Properties props = new Properties();  
    props.setProperty("user","Hammoud");  
    props.setProperty("password","secret");  
    props.setProperty("ssl","true");  
    Connection conn = DriverManager.getConnection(url, props);  
  
    System.out.println("Connected to database");  
    return conn;  
}
```

# Creating Tables

- Assume the following students table:

Sid	Name
1	Hammoud
2	Esam

**SQL:**

```
CREATE TABLE students( sid INTEGER, name CHAR(30), PRIMARY KEY (sid))
```

**JDBC:**

```
public void createTable() throws SQLException {
    String createT = "create table students (sid INTEGER, " +
        "name CHAR(30) " +
        "PRIMARY KEY (sid))";
    Statement stmt = null;
    try {
        stmt = conn.createStatement();
        stmt.executeUpdate(createT);
    } catch (SQLException e) { e.printStackTrace(e); }
    finally { if (stmt != null) { stmt.close(); } }
}
```

# Populating Tables

- Assume the following students table:

Sid	Name
1	Hammoud
2	Esam

**SQL:**

```
INSERT INTO students values (1, 'Hammoud')  
INSERT INTO students values (2, 'Esam')
```

**JDBC:**

```
public void populateTable() throws SQLException {  
    Statement stmt = null;  
    try {  
        stmt = conn.createStatement();  
        stmt.executeUpdate( "insert into students values(1, 'Hammoud')");  
        stmt.executeUpdate( "insert into students values(2, 'Esam')");  
    } catch (SQLException e) {}  
    finally { if (stmt != null) { stmt.close(); } }  
}
```

# Querying Tables

- Assume the following students table:

Sid	Name
1	Hammoud
2	Esam

**SQL:** SELECT sid, name from students

A "cursor" that points to one row of data at a time

**JDBC:**

```
public static void viewTable() throws SQLException {
    Statement stmt = null;
    String query = "select sid, name from students";
    try {
        stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
            int sID = rs.getInt("sid");
            String sName = rs.getString("name");
            System.out.println(sName + "\t" + sID); }
        } catch (SQLException e ) {} finally { if (stmt != null) { stmt.close(); } }
}
```

Columns retrieved by names



# Querying Tables

- Assume the following students table:

Sid	Name
1	Hammoud
2	Esam

**SQL:** SELECT sid, name from students

**JDBC:**

```
public static void viewTable() throws SQLException {
    Statement stmt = null;
    String query = "select sid, name from students";
    try {
        stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
            int sID = rs.getInt(1);
            String sName = rs.getString(2);
            System.out.println(sName + "\t" + sID); }
        } catch (SQLException e ) {} finally { if (stmt != null) { stmt.close(); } }
}
```

OR: Columns retrieved by numbers

# Cursor Methods

- Methods available to move the cursor of a result set:
  - `next()`
  - `previous()`
  - `first()`
  - `Last()`
  - `beforeFirst()`
  - `afterLast()`
  - `relative(int rows)`
  - `absolute(int row)`

By default, you can  
call only `next()`!

# Updating Tables

- By default, ResultSet objects cannot be updated, and their cursors can only be moved forward
- ResultSet objects can be though defined to be *scrollable* (the cursor can move backwards or move to an absolute position) and *updatable*

```
public void modifyStudents() throws SQLException {
    Statement stmt = null;
    try {
        /* stmt = con.createStatement(); */
        stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                   ResultSet.CONCUR_UPDATABLE);
        ResultSet uprs = stmt.executeQuery( "SELECT * FROM students");
        while (uprs.next()) {
            String old_n = uprs.getString("name");
            uprs.updateString( "name", "Mohammad" + old_n);
            uprs.updateRow(); }
        } catch (SQLException e ) {} finally { if (stmt != null) { stmt.close(); } }
}
```

# Result Set Types

- `TYPE_FORWARD_ONLY` (the default)
  - The result set is not scrollable
- `TYPE_SCROLL_INSENSITIVE`
  - The result set is scrollable
  - The result set is insensitive to changes made to the underlying data source while it is open
- `TYPE_SCROLL_SENSITIVE`
  - The result set is scrollable
  - The result set is sensitive to changes made to the underlying data source while it is open

# Result Set Concurrency

- The concurrency of a ResultSet object determines what level of update functionality is supported
- Concurrency levels:
  - `CONCUR_READ_ONLY` (the default)
    - The result set cannot be updated
  - `CONCUR_UPDATABLE`
    - The result set can be updated

# Prepared Statements

- JDBC allows using a PreparedStatement object for sending SQL statements to a database
- This way, the same statement can be used with different values many times

```
...  
String sql = "INSERT into students values(?, ?)";  
PreparedStatement ps = conn.prepareStatement(sql);  
ps.clearParameters();  
ps.setInt(1, 111);  
ps.setString(2, "Hammoud");  
int numRows1 = ps.executeUpdate();  
  
ps.setInt(1, 222);  
ps.setString(2, "Esam");  
int numRows2 = ps.executeUpdate();  
...
```

More about  
JDBC in the  
upcoming two  
recitations!

# Outline

NULL values and Join Variants

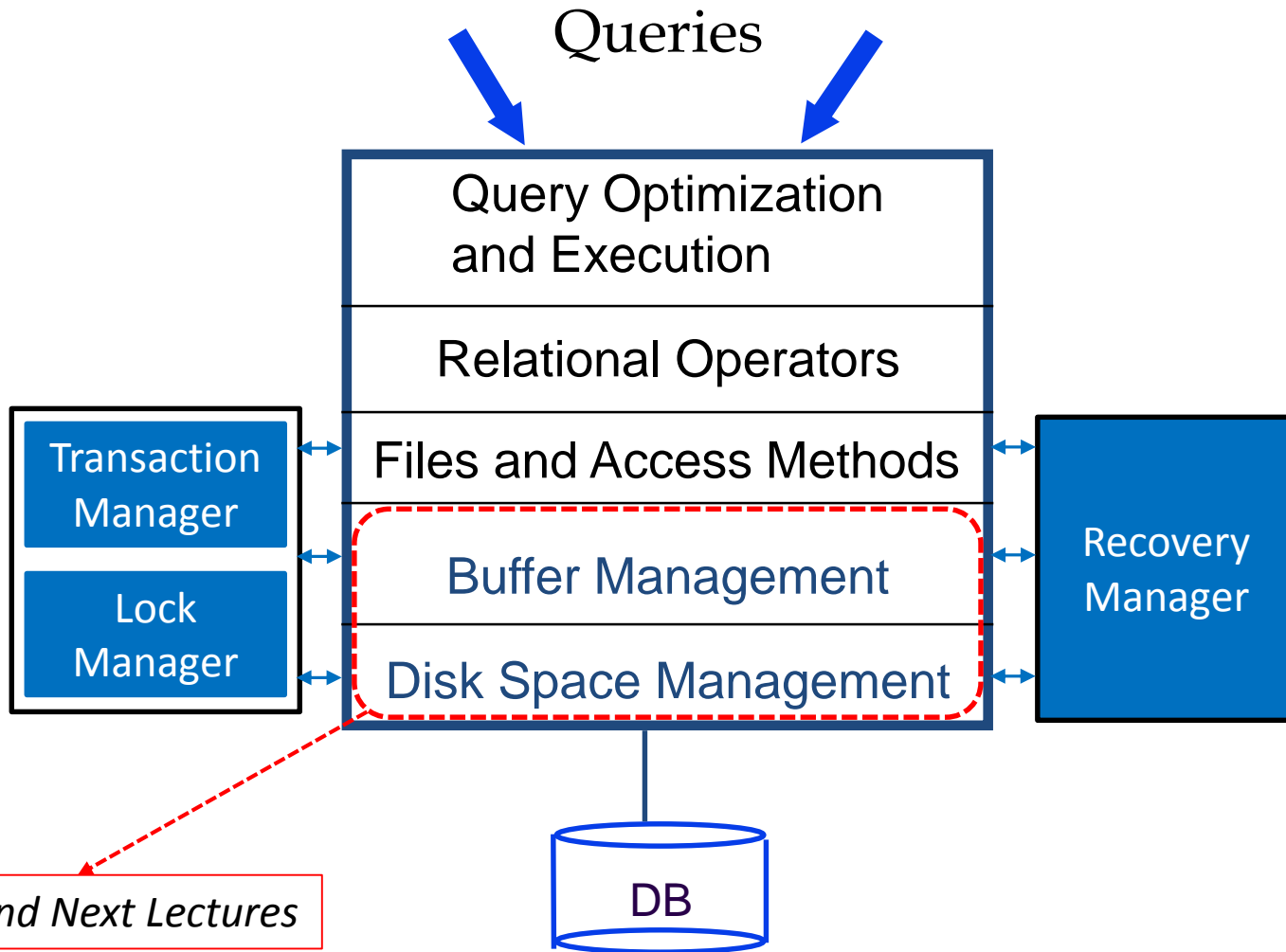
Complex Integrity Constraints and Triggers

Java Database Connectivity

Storing Data: Disks and Files (*Briefly*)



# DBMS Layers

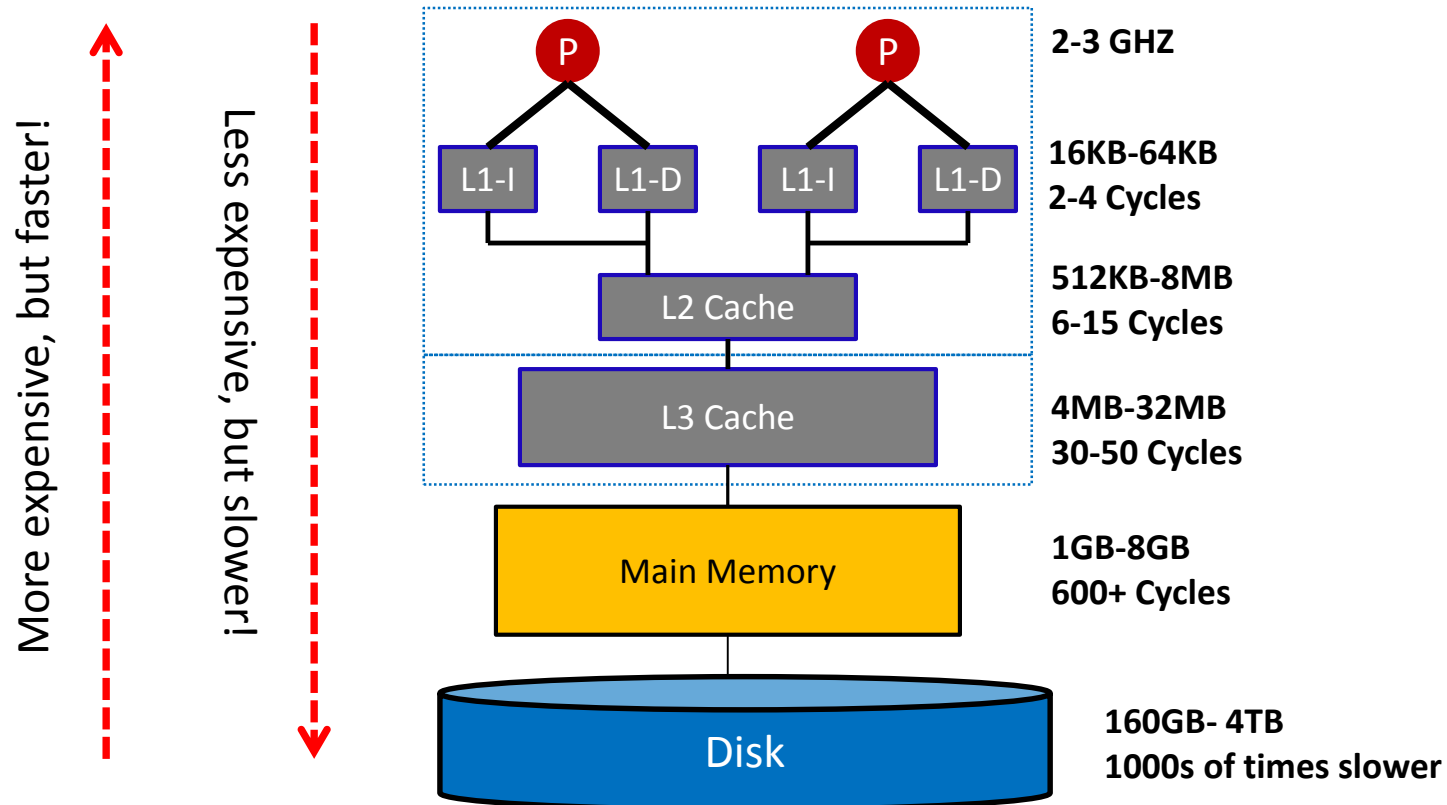


*Today and Next Lectures*



# The Memory Hierarchy

- Storage devices play an important role in database systems
- How systems arrange storage?



# Where to Store Data?

- Where does DBMS store information?
  - DBMSs can store large amount of data (e.g., Big Data; but not necessarily centralized!)
  - Buying enough memory to store all data might be prohibitively expensive (let alone that memories are *volatile*)
  - Thus, databases are usually stored on disks (or tapes for backups)

# But, What Will Do With Memory?

- Data must be brought into memory to be processed!

- **READ:** transfer data from disk to main memory (RAM)

## I/O Time

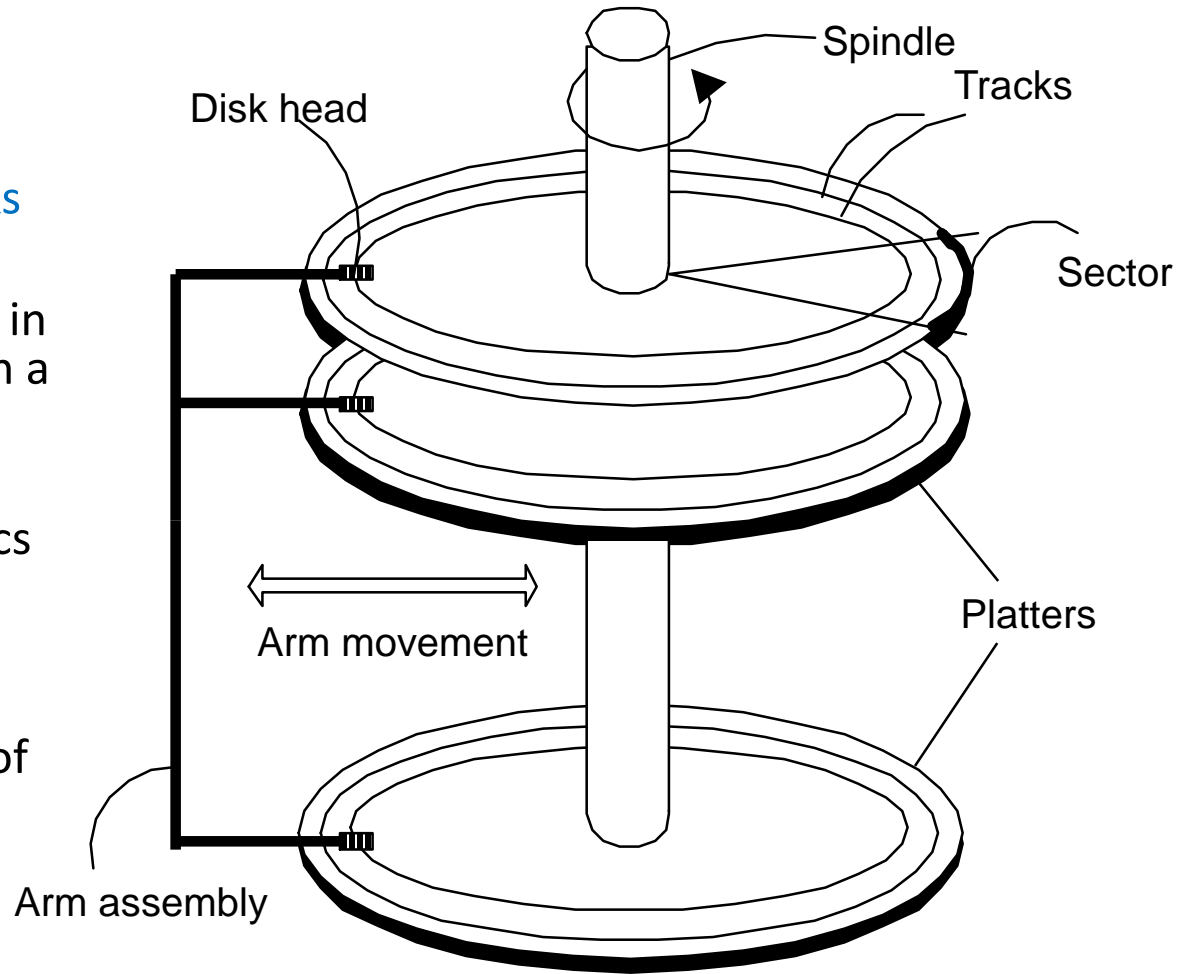
- **WRITE:** transfer data from RAM to disk

- I/O time dominates the time taken for database operations!

- To minimize I/O time, it is necessary to store and locate data strategically

# Magnetic Disks

- Data is stored in disk **blocks**
- Blocks are arranged in concentric rings called **tracks**
- The arm assembly is moved in or out to position a head on a desired track
- Each track is divided into arcs called **sectors** (whose size is fixed)
- The block size is a multiple of sector size
- The set of all tracks with the same diameter is called **cylinder**



# Accessing a Disk Block

- What is I/O time?
  - The time to move the disk heads to the track on which a desired block is located
  - The waiting time for the desired block to rotate under the disk head
  - The time to actually read or write the data in the block once the head is positioned

# Accessing a Disk Block

- What is I/O time?

- Seek Time

- Rotational Time

- Transfer Time

- $I/O \text{ time} = \text{seek time} + \text{rotational time} + \text{transfer time}$

# Implications on DBMSs

- Seek time and rotational delay dominate!
- Key to lower I/O cost: **reduce seek/rotation delays!**
- How to minimize seek and rotational delays?
  - Blocks on same track, followed by
  - Blocks on same cylinder, followed by
  - Blocks on adjacent cylinder
  - Hence, sequential arrangement of blocks in a file is a big win!

Who can take care of that?

# Storing Data: Concluding Remarks

- Disks provide cheap, non-volatile storage for DBMSs
- However, data must be in memory for the DBMS to operate on it
- I/O time dominates!
- The cost depends on the locations of pages on disk
- It is important to arrange data sequentially to minimize *seek* and *rotation* delays



# Next Class

## Quiz I & Storing Data: Disks and Files (*Cont'd*)