

# Lecture 5

- Views
  - Logical Independence
  - View Definitions
  - When to Use Views
  - Updating Views
- Embedded SQL
  - Cursors
  - Implementations
- More Database Commands
  - DDL, DML

# Views

- A *view* is a query stored in the database
  - Think of it as a table definition for future use

- Example view definition:

```
CREATE VIEW gstudents AS SELECT S.*  
FROM student S WHERE s.gpa >= 2.5
```

- Views can be used like *base tables*, in any query or in any other view. Like a Macro.

## Views used to make queries simpler

- Completed(StudID, Course)
- Queries are often about “good” students only

```
SELECT S.name, S.phone  
FROM gstudent S NATURAL JOIN completed C  
WHERE C.course = 'CS386';
```

- Now it's easier to write the query.

# How are views implemented?

When you enter a query that mentions a view in the from clause, the DBMS expands/rewrites your query to include the view definition. Then it optimizes/runs it.

```
SELECT S.name, S.phone  
FROM gstudent S NATURAL JOIN completed C  
WHERE C.course = 'CS386';
```

is rewritten as:

```
SELECT S.name, S.phone  
FROM (SELECT S.* FROM student S WHERE s.gpa >= 2.5) AS S  
NATURAL JOIN completed C  
WHERE C.course = 'CS386';
```

## Views used for Security

- This view presents a table (called sstudent) that is the student relation without the gpa field.

```
CREATE VIEW sstudent AS  
SELECT studID, name, address  
FROM student
```

- What's missing from this view that's in the base table?

## Views used to support extensibility

- A company's database includes a relation:  
**Part (PartID: Char(4), weight:real,...)**
- Weight is stored in pounds
- Company is purchased by a firm that uses metric weights
- Databases must be integrated and use Kg.
- But there's much old software using pounds.
- Solution: views!

## Views to support extensibility (ctd)

Solution:

1. Base table with kilograms becomes NewParts, for integrated company.
2. **CREATE VIEW Part AS**  
**SELECT PartID, 2.2046\*weight, ...**  
(no other changes)...  
**FROM NewParts**
3. Old programs still call the table “Part”

## But there's one problem with views

- Views cannot always be updated unambiguously
- Consider Students(StudID, gpa, major,...)

```
CREATE VIEW majorgpa AS
```

```
SELECT major, AVG(gpa)
```

```
FROM Students
```

```
GROUP BY major
```

Majorgpa	major	gpa
	CS	3.5
	ECE	3.5

# Updatability of views

- I want to change the GPA of CS majors from 3.5 to 3.6 .
- How can I do that? (Should I do that?)

## Another example of the view update problem

- Views cannot always be updated unambiguously.  
Consider

emp(empid, ename, address, deptid)  
dept(deptid, dname)

```
CREATE VIEW empdept AS  
SELECT ename, dname  
FROM emp JOIN dept USING (deptid)
```

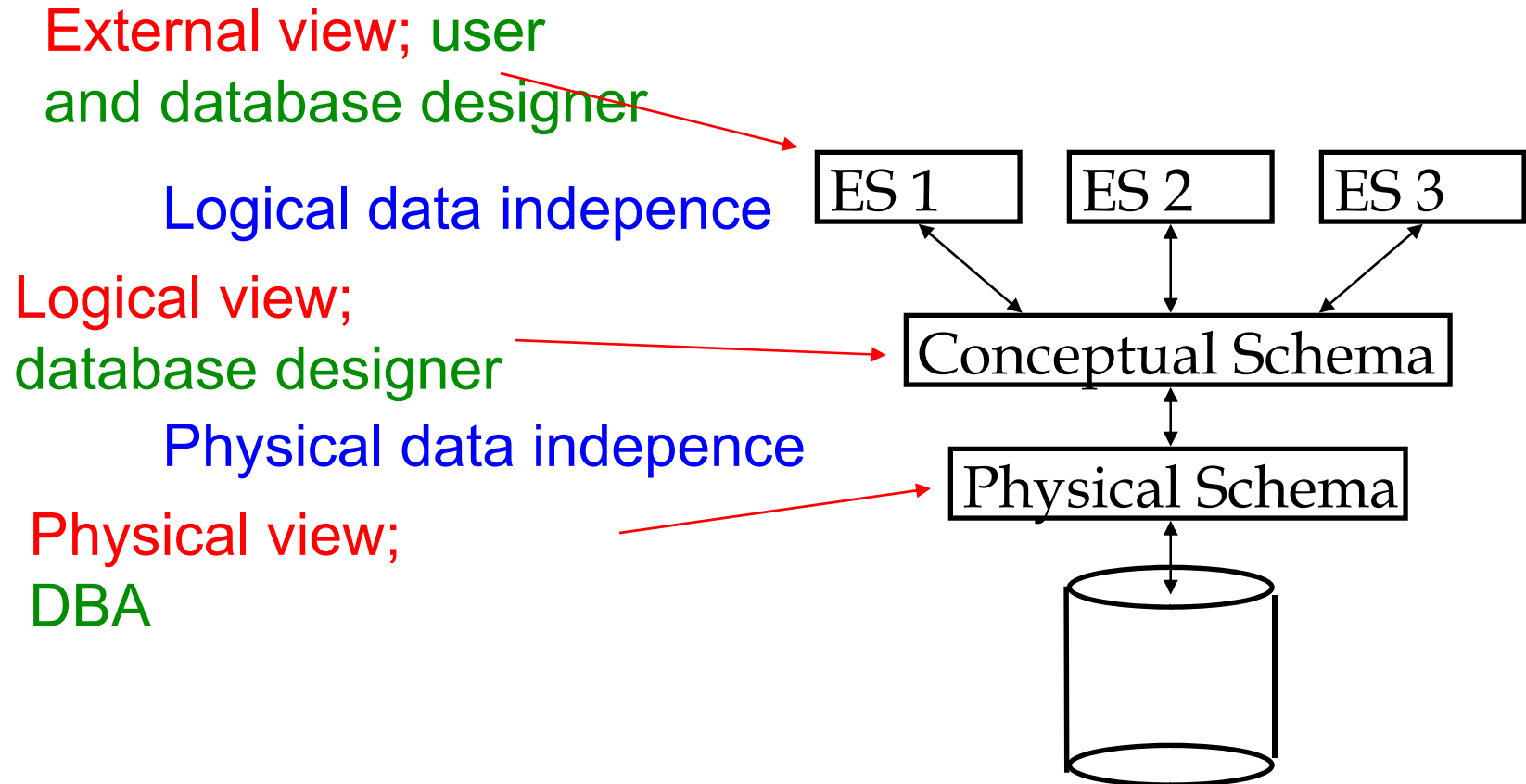
EMPDEPT	ename	dname
	jim	shoe
	joe	suit

- I want to delete (jim, shoe) from EMPDEPT.
  - How can I do that?
-

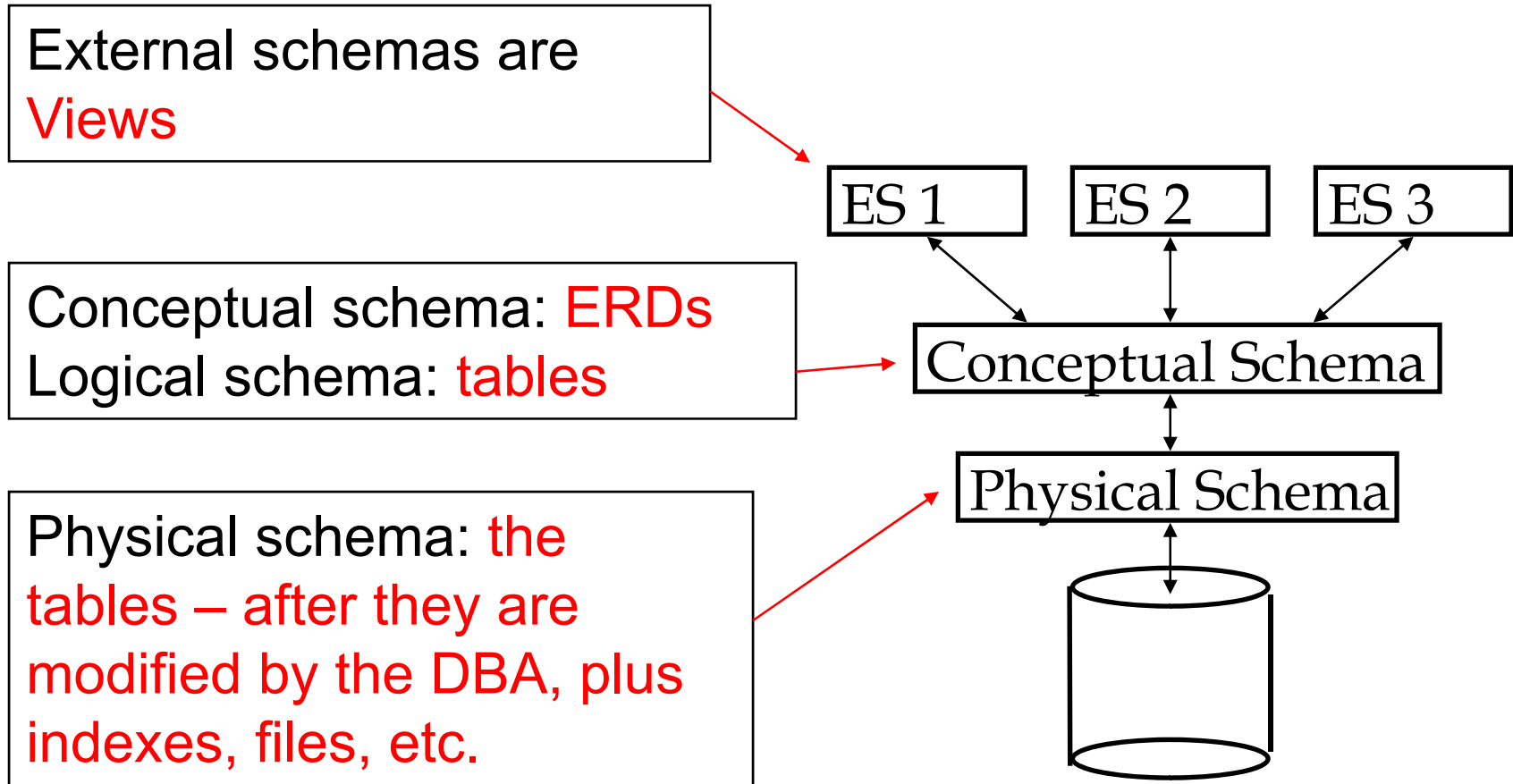
# The good news

- A view can be updated if
  - It is defined on a single base table
  - Using only selection and projection
  - No aggregates, group by
  - No DISTINCT
  - No set operations
- Note: the details of whether or not a view is updateable varies among products
- SQL:1999 extends the def'n of updateable view

# Levels of Abstraction – support Data Independence



# Levels of Abstraction in a relational DBMS



# Data Independence

- A database model possesses *data independence* if application programs are immune to changes in the conceptual and physical schemas.
- Why is this important? Everything changes.
- How does the relational model achieve logical (conceptual) data independence?

# Summary

- A view is a stored query definition
- Views can be very useful
  - Easier query writing, security, extensibility
- But views cannot always be unambiguously updated
- Three levels of abstraction in a DBMS
  - Yields data independence: logical and physical
  - Note: there may be ways (yet to be invented) that support better logical or physical data independence

# Final Thoughts on Views

- View versus temporary table
  - What is the difference?
  - Which is better?
- Some DBMS' offer a way around the view update problem
  - Triggers
  - Rules
  - Active research

# Embedded SQL: What and Why?

- Embedded SQL allows data from a DBMS to be accessed from a regular program (in some programming or scripting language)
- Embedded SQL Programmers can:
  - Convert/modify data before it is presented to users
  - Control what data is visible to users
  - Generate SQL dynamically based on user preferences

# Embedded SQL

- SQL commands can be called from within a host language (e.g., C/C++, Basic, .NET, PHP) program.
  - Must include a statement to *connect* to the right database.
  - SQL statements can refer to *host variables – the regular variables in your program* (plus special SQL variables used to return status).
- SQL relations are (multi-) sets of records, which may be very large. No data structure is available in most languages to hold such a large query answer.
  - SQL supports a mechanism called a *cursor* to handle this.

# Cursors

- Can declare a cursor on a **relation** or **query statement** (which generates a relation).
- Special cursor-specific SQL clause: ORDER BY
- Can *open* a cursor, and repeatedly *fetch* a tuple then *move* the cursor, until all tuples have been retrieved.
- Can also modify/delete tuple pointed to by a cursor.
- Must be able to report data-generated errors.

# Cursors, Continued

- Capabilities vary from one DBMS to another:
  - Move forward only
  - Move forward or backward, one row at a time
  - Move to arbitrary location
  - Advantages/disadvantages?
- Cursor placement
  - Cursor is placed BEFORE first row of the result
  - Special “end of result” sentinel – when you’re past the end of the last row

# Embedded SQL Implementations

- We will show a small sample of two implementations of Embedded SQL (actual details may vary)
  - C (Pro\*C from Oracle is an example)
  - Scripting (Using PHP & PostgreSQL)

# Example Table Schema

Imagine we are tracking products and categories for a retailer

**Products** (int ProductID, int CategoryID, String ProductName, currency UnitPrice)

**Categories** (int CategoryID, String CategoryName)



What are the foreign keys?

# Embedding SQL in C: An Example

```
void ListProducts(short Max)
{
    char SQLSTATE[6];

    EXEC SQL BEGIN DECLARE SECTION
        char ProductName[20];
        float ProductPrice;
        short MaxPrice = Max;
    EXEC SQL END DECLARE SECTION
```

SQLSTATE holds the return value from the DBMS

EXEC SQL denotes embedded SQL section

DECLARE SECTION binds variables into SQL

Max is a parameter into this procedure; used to initialize MaxPrice.

# Embedding SQL in C: (continued)

```
EXEC SQL DECLARE pinfo CURSOR FOR
SELECT P.ProductName, P.UnitPrice
FROM Products P, Categories C
WHERE C.CategoryName="Beverages"
AND P.UnitPrice < :MaxPrice
AND P.CategoryID=C.CategoryID
ORDER BY P.UnitPrice;
```

DECLARE pinfo CURSOR defines a name for the cursor for this query for use in this program.

SELECT P.ProductName ... is the SQL query that the programmer writes

:MaxPrice - a variable, defined on the preceding slide

# Embedding SQL in C: (continued)

```
EXEC SQL OPEN pinfo;
```

Variables declared  
earlier

```
EXEC SQL FETCH pinfo INTO :ProductName, :ProductPrice;  
while (SQLSTATE != "02000") {  
    printf("%s costs %f each\n", ProductName, ProductPrice);  
    EXEC SQL FETCH pinfo INTO :ProductName, :ProductPrice;  
};
```

```
EXEC SQL CLOSE pinfo;
```

**OPEN pinfo** - opens the query we are interested in

**FETCH pinfo INTO** - assigns data into our variables, for use in the output and advances the cursor to the next row

**ProductName** - Use of our variable in and out of SQL EXEC

**CLOSE pinfo** - We're done with the cursor, free up its resources

# Embedded SQL using PostgreSQL and PHP

- Scripting languages such as Perl, Python, and PHP have database support
  - We'll focus on PHP
- To work with a PostgreSQL database in a scripting language, you must use a set of functions designed to communicate with PostgreSQL
- PHP includes the PostgreSQL functions - you do not need an additional library

# Embedded SQL using PostgreSQL and PHP

```
$sql_query = "SELECT P.ProductName, P.UnitPrice ";  
$sql_query .= "FROM Products P, Categories C ";  
$sql_query .= "WHERE C.CategoryName = 'Beverages' ";  
$sql_query .= " AND P.CategoryID = C.CategoryID ";  
$res = pg_exec($conn, $sql_query) or  
die("Error executing query: ".pg_errormessage());
```

- \$conn – Connection to an open PostgreSQL database
- \$res – Result index ( $\approx$  Cursor)
- pg\_exec  $\approx$  SQL EXEC OPEN
- The ‘.’=’ performs string catenation, ‘\$’ denotes variable
- die – terminate execution if the statement fails, with the given message

# Embedded SQL using PostgreSQL and PHP

```
## First, connect to the PostgreSQL Server
```

```
$connection=pg_connect("host=localhost dbname=Products user=me")  
or die("Error: Cannot open the database.\n".pg_errormessage());
```

```
## Construct the query, .= is the concatenation operator
```

```
$sql_query = "SELECT P.ProductName, P.UnitPrice ";  
$sql_query .= "FROM Products P, Categories C ";  
$sql_query .= "WHERE P.UnitPrice < " . $MaxPrice;  
$sql_query .= " AND C.CategoryName = 'Beverages' ";  
$sql_query .= " AND P.CategoryID = C.CategoryID ";  
$sql_query .= "ORDER BY P.UnitPrice";
```

```
## Issue the query to PostgreSQL
```

```
$query_results = pg_exec($connection, $sql_query) or  
die("Error executing query: ".pg_errormessage());
```

# Embedded SQL using PostgreSQL and PHP

```
## Now that we have the "cursor", we can do something with it

## .. We can determine the number of rows in the result ..
$query_num_rows = pg_numrows($query_results);
echo "The query has ".$query_num_rows." rows...\n";

## .. We can also step through the recordset
while ( $row = pg_fetch_row($query_results) ) {
    ## $row is an array containing the values of a single tuple
    ## or $row is false when there are no more tuples
    ## $row can be indexed by attribute name
    printf("%s costs ", $row["ProductName"]);
    printf("%.2f dollars\n", $row["UnitPrice"]);
}
```

# PHP/PostgreSQL references

- <http://www.postgresql.org>
- <http://www.php.net>

# Other Embedded SQL Solutions

- ODBC – Open Database Connectivity
  - Old standard, proposed by Microsoft but driven by the database community
  - Many vendors, including Oracle, make ODBC drivers available
- Haskell
  - HaskellDB (currently uses ADO)
  - Cursors are first-class objects (each attribute is a type)

# Rowsets

- Different from a RecordSet!
- Rowset objects are available in both Java and .Net implementations
  - .Net calls them “DataTables”
- **In-memory abstraction of a table**
- Allows for in-place editing
  - Don’t forget to commit!

# Database Language Commands

- DDL
  - Data definition language
  - Schema-level commands
- DML
  - Data manipulation language
  - Row-level commands

# DDL: Data Definition Language

- Create, edit, or delete database objects
  - Tables
  - Stored Procedures
  - Data Types
  - NOT ROWS!
- Drop table:

`DROP TABLE Patient;`

# CREATE TABLE (SQL SERVER)

```
CREATE TABLE Patient (  
    id uniqueidentifier NOT NULL,  
    FirstName varchar(50),  
    MiddleName varchar(30),  
    LastName varchar(50),  
    Provider uniqueidentifier,  
    DateOfBirth datetime,  
    CONSTRAINT pk_Patient PRIMARY KEY (id)  
    CONSTRAINT fk_Patient FOREIGN KEY (Provider)  
    REFERENCES (Staff.id)  
)
```

# ALTER TABLE

ALTER TABLE Patient

\* Plus any of the following:

[ADD COLUMN]

[ALTER COLUMN]

[DROP COLUMN]

[ADD CONSTRAINT]

[DROP CONSTRAINT]

# DML: Data Manipulation Language

- Inserting, updating, or deleting rows
- Delete:

```
DELETE FROM Patient  
WHERE FirstName LIKE 'B%'
```

Note: this will delete multiple patients.

# INSERT INTO

```
INSERT INTO Patient (id, FirstName, LastName)  
VALUES (4, 'Bob', 'Thomas'), (6, 'Ned', 'Flanders')
```

```
INSERT INTO Patient (id, FirstName, DateOfBirth)  
SELECT Guid, Fname, DOB FROM Staff WHERE  
IsPatient = 1
```

This will insert just one patient – with data from the one row returned by this query.

# UPDATE

```
UPDATE TABLE Patient
SET  FirstName = 'Bob',
     DateOfBirth = AddDays(DateOfBirth, 1)
WHERE Provider = 8
```

This will change the FirstName and DateOfBirth  
for all patients where Provider = 8

(This is probably not something you would  
actually do in real life.)