



Dr. Vishwanath Karad

**MIT WORLD PEACE
UNIVERSITY** | PUNE
TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

CS312

Database Management Systems

School of Computer Engineering and Technology

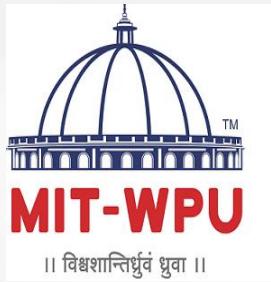
CS312 Database Management Systems

Course Objectives:

- 1) Understand and successfully apply logical database design principles, including E-R diagrams and database normalization.
- 2) Learn Database Programming languages and apply in DBMS application
- 3) Understand transaction processing and concurrency control in DBMS
- 4) Learn database architectures, DBMS advancements and its usage in advance application

Course Outcomes:

- 5) Design ER-models to represent simple database application scenarios and Improve the database design by normalization.
- 6) Design Database Relational Model and apply SQL , PLSQL concepts for database programming
- 7) Describe Transaction Processing and Concurrency Control techniques for databases
- 8) Identify appropriate database architecture for the real world database application



Extensions to SQL (PL/SQL)

What is PL/SQL

PL/SQL:

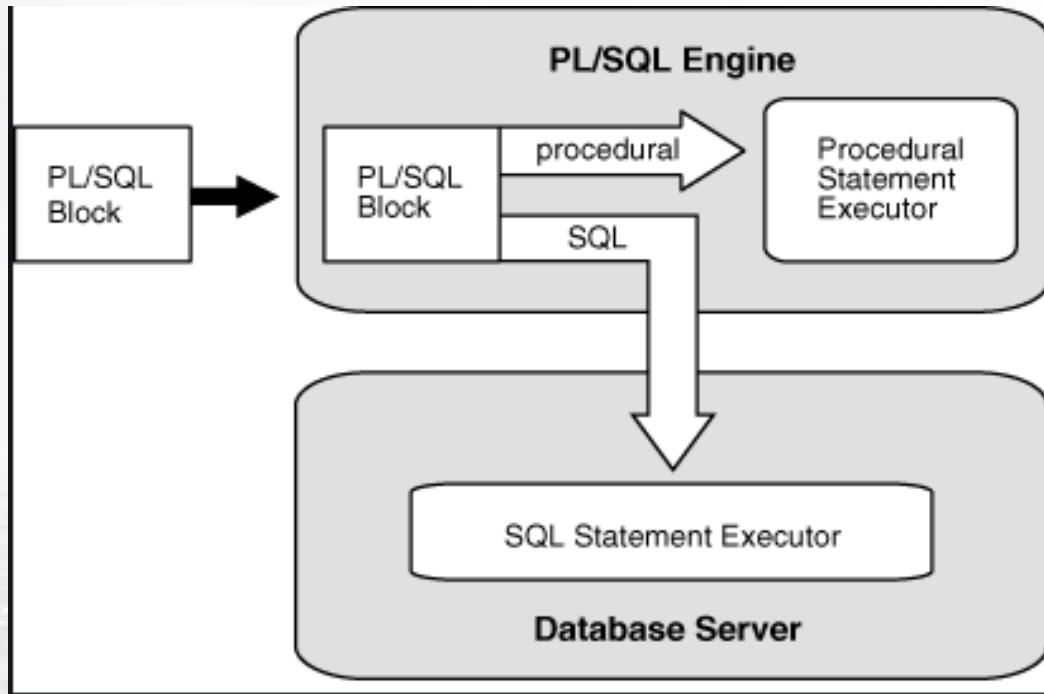
- Stands for Procedural Language extension to SQL
- Is Oracle Corporation's standard data access language for relational databases
- Seamlessly integrates procedural constructs with SQL



PL/SQL Execution

Advantages

- PL/SQL is a completely portable, high-performance transaction-processing language.
- PL/SQL provides a built-in, interpreted and OS independent programming environment.
- It supports structured programming through functions and procedures.
- Direct call can also be made from external programming language calls to database.



PL/SQL Functions and Procedures

- SQL:1999 supports functions and procedures
 - Functions/procedures can be written in SQL itself, or in an external programming language (e.g., C, Java).
 - Some database systems support **table-valued functions**, which can return a relation as a result.
- SQL:1999 also supports a rich set of imperative constructs, including
 - Loops, if-then-else, assignment
- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999.
- Procedures and functions are stored in `mysql.routines` and `mysql.parameters` tables, which are part of the data dictionary.

Stored Function

PL/SQL Functions

- Functions are declared using the following syntax:

```
Create function <function-name> (param_1, ..., param_k)
    returns <return_type>
    [not] deterministic      allow optimization if same output
        for the same input (use RAND not deterministic )
```

Begin

```
    -- execution code
    end;
```

For a **FUNCTION**, parameters are always regarded as **IN** parameters.

For a **Procedure** , parameter as IN, OUT, or INOUT is valid.

Deterministic and Non-deterministic Functions

- A deterministic function always returns the same result for the same input parameters whereas a non-deterministic function returns different results for the same input parameters.
- If you don't use DETERMINISTIC or NOT DETERMINISTIC, MySQL uses the NOT DETERMINISTIC option by default.
- **rand()** is nondeterministic function. That means we do not know what it will return ahead of time.
- Some **deterministic** functions
- ISNULL, ISNUMERIC, DATEDIFF, POWER, CEILING, FLOOR, DATEADD, DAY, MONTH, YEAR, SQUARE, SQRT etc.
- Some **non deterministic** functions
- RAND(), RANK(), SYSDATE()

PL/SQL Functions – Example 1

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.

```
create function dept_count (dept_name varchar(20))
    returns integer
begin
    declare d_count integer;
    select count (*) into d_count
    from instructor
    where instructor.dept_name = dept_name
    return d_count;
end
```

Example 1 (Cont)..

- The function dept_count can be used to find the department names and budget of all departments with more than 12 instructors.

```
select dept_name, budget
from department
where dept_count(dept_name) > 12
```

Example 2

- A function that returns the level of a customer based on credit limit. We use the IF statement to determine the credit limit.

```
1 DELIMITER $$  
2  
3 CREATE FUNCTION CustomerLevel(p_creditLimit double) RETURNS VARCHAR(10)  
4     DETERMINISTIC  
5 BEGIN  
6     DECLARE lvl varchar(10);  
7  
8     IF p_creditLimit > 50000 THEN  
9         SET lvl = 'PLATINUM';  
10    ELSEIF (p_creditLimit <= 50000 AND p_creditLimit >= 10000) THEN  
11        SET lvl = 'GOLD';  
12    ELSEIF p_creditLimit < 10000 THEN  
13        SET lvl = 'SILVER';  
14    END IF;  
15  
16    RETURN (lvl);  
17 END
```

Example 2 (Cont..)

- **Calling function:**
- we can call the CustomerLevel() in a SELECT statement as follows:

```
1 SELECT
2     customerName,
3     CustomerLevel(creditLimit)
4 FROM
5     customers
6 ORDER BY
7     customerName;
```

Output:

	customerName	CustomerLevel(creditLimit)
▶	Alpha Cognac	PLATINUM
	American Souvenirs Inc	SILVER
	Amica Models & Co.	PLATINUM
	ANG Resellers	SILVER
	Anna's Decorations, Ltd	PLATINUM
	Anton Designs, Ltd.	SILVER

Example 3

```
mysql> select * from employee;
+----+-----+-----+-----+-----+-----+
| id | name | superid | salary | bdate | dno |
+----+-----+-----+-----+-----+-----+
| 1  | john |       3 | 100000 | 1960-01-01 | 1   |
| 2  | mary |       3 | 50000  | 1964-12-01 | 3   |
| 3  | bob  |     NULL | 80000  | 1974-02-07 | 3   |
| 4  | tom  |       1 | 50000  | 1970-01-17 | 2   |
| 5  | bill |     NULL |    NULL | 1985-01-20 | 1   |
+----+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)

mysql> delimiter ;
mysql> create function giveRaise (oldval double, amount double
-> returns double
-> deterministic
-> begin
->     declare newval double;
->     set newval = oldval * (1 + amount);
->     return newval;
-> end ;
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
```

Example 3 (cont..)

```
mysql> select name, salary, giveRaise(salary, 0.1) as newsal
-> from employee;
+-----+-----+-----+
| name | salary | newsal |
+-----+-----+-----+
| john | 100000 | 110000 |
| mary | 50000  | 55000  |
| bob  | 80000  | 88000  |
| tom  | 50000  | 55000  |
| bill | NULL   | NULL   |
+-----+-----+-----+
5 rows in set (0.00 sec)
```

Stored Procedures

Stored Procedures in MySQL

- A stored procedure contains a sequence of SQL commands stored in the database catalog so that it can be invoked later by a program
- Stored procedures are declared using the following syntax:

```
Create Procedure <proc-name>
    (param_spec1, param_spec2, param_specn)
begin
    -- execution code
end;
```

where each param_spec is of the form:

```
[in | out | inout] <param_name> <param_type>
```

- in mode: allows you to pass values into the procedure,
- out mode: allows you to pass value back from procedure to the calling program, initial value in the procedure is taken as null .
- Inout mode : allows you to pass value back from procedure to the calling program, initial value in the procedure is taken from the caller value

Example 1 – No parameters

- The GetAllProducts() stored procedure selects all products from the products table.

```
mysql> use classicmodels;
Database changed
mysql> DELIMITER //
mysql> CREATE PROCEDURE GetAllProducts()
-> BEGIN
-> SELECT * FROM products;
-> END//
Query OK, 0 rows affected (0.00 sec)

mysql> DELIMITER ;
mysql>
```

Example 1 (Cont..)

- **Calling Procedure:**

```
CALL GetAllProducts();
```

- **Output:**

	productCode	productName	productLine	productScale
	S10_1678	1969 Harley Davidson Ultimate Chopper	Motorcycles	1:10
	S10_1949	1952 Alpine Renault 1300	Classic Cars	1:10
	S10_2016	1996 Moto Guzzi 1100i	Motorcycles	1:10
	S10_4698	2003 Harley-Davidson Eagle Drag Bike	Motorcycles	1:10
	S10_4757	1972 Alfa Romeo GTA	Classic Cars	1:10

Example 2 (with IN parameter)

```
mysql> select * from employee;
+----+-----+-----+-----+-----+
| id | name | superid | salary | bdate      | dno |
+----+-----+-----+-----+-----+
| 1  | john  |      3 | 100000 | 1960-01-01 | 1   |
| 2  | mary   |      3 | 50000  | 1964-12-01 | 3   |
| 3  | bob    |    NULL | 80000  | 1974-02-07 | 3   |
| 4  | tom    |      1 | 50000  | 1978-01-17 | 2   |
| 5  | bill   |    NULL |    NULL | 1985-01-20 | 1   |
+----+-----+-----+-----+-----+
```

```
mysql> select * from department;
+-----+-----+
| dnumber | dname   |
+-----+-----+
|       1 | Payroll |
|       2 | TechSupport |
|       3 | Research |
+-----+-----+
```

- Suppose we want to keep track of the total salaries of employees working for each

```
mysql> create table deptsal as
-> select dnumber, 0 as totalsalary from department;
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
mysql> select * from deptsal;
+-----+-----+
| dnumber | totalsalary |
+-----+-----+
|      1 |          0 |
|      2 |          0 |
|      3 |          0 |
+-----+-----+
```

We need to write a procedure
to update the salaries in
the deptsal table

Example 2 (Cont..)

```
mysql> delimiter //
mysql> create procedure updateSalary (IN param1 int)
-> begin
->     update deptsal
->     set totalsalary = (select sum(salary) from employee where dno = param1)
->     where dnumber = param1;
-> end; //
Query OK, 0 rows affected (0.01 sec)
```

1. Define a procedure called **updateSalary** which takes as input a department number.
2. The body of the procedure is an SQL command to update the totalsalary column of the deptsal table.

Example 2 (Cont..)

Step 3: Call the procedure to update the totalsalary for each department

```
mysql> call updateSalary(1);
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> call updateSalary(2);
Query OK, 1 row affected (0.00 sec)
```

```
mysql> call updateSalary(3);
Query OK, 1 row affected (0.00 sec)
```

Example 2 (Cont..)

Step 4: Show the updated total salary in the deptsal table

```
mysql> select * from deptsal;
+-----+-----+
| dnumber | totalsalary |
+-----+-----+
|      1 |      100000 |
|      2 |       50000 |
|      3 |      130000 |
+-----+-----+
3 rows in set (0.00 sec)
```

Example 3 (with OUT Parameter)

- The following example shows a simple stored procedure that uses an OUT parameter.
- Within the procedure MySQL MAX() function retrieves maximum salary from **MAX_SALARY** of jobs table.

```
mysql> CREATE PROCEDURE my_proc_OUT (OUT highest_salary INT)
-> BEGIN
-> SELECT MAX(MAX_SALARY) INTO highest_salary FROM JOBS;
-> END$$
Query OK, 0 rows affected (0.00 sec)
```

(Cont..)

- **Procedure Call:**

```
mysql> CALL my_proc_OUT(@M)$$
```

Query OK, 1 row affected (0.03 sec)

- To see the result type the following command

```
mysql< SELECT @M$$
```

- **Output:**

```
+-----+
```

```
| @M |
```

```
+-----+
```

```
| 40000 |
```

```
+-----+
```

```
1 row in set (0.00 sec)
```

Example 4 (with INOUT Parameter)

- The following example shows a simple stored procedure that uses an INOUT parameter.
- 'count' is the **INOUT parameter**, which can store and return values and 'increment' is the IN parameter, which accepts the values from user.

```
mysql> DELIMITER // ;
mysql> Create PROCEDURE counter(INOUT count INT, IN increment INT)
      -> BEGIN
      -> SET count = count + increment;
      -> END //
Query OK, 0 rows affected (0.03 sec)
```

Example 4 (Cont..)

**Function
Call:**

```
mysql> DELIMITER ;
mysql> SET @counter = 0;
Query OK, 0 rows affected (0.00 sec)

mysql> CALL counter(@Counter, 1);
Query OK, 0 rows affected (0.00 sec)

mysql> Select @Counter;
+-----+
| @Counter |
+-----+
| 1         |
+-----+
1 row in set (0.00 sec)
```

Stored Procedures (Cont..)

- Use **show procedure status** to display the list of stored procedures you have

```
mysql> show procedure status;
+-----+-----+-----+-----+-----+-----+-----+
| Db   | Name      | Type     | Definer  | Modified   | Created    | Security_|
| type | Comment   | character_set_client | collation_connection | Database | Collation |
+-----+-----+-----+-----+-----+-----+-----+
| ptan | updateSalary0 | PROCEDURE | ptan@% | 2010-03-16 12:21:55 | 2010-03-16 12:21:55 | DEFINER |
|       |             |           |          |           |           |           |
|       |             | latin1   |          | latin1_swedish_ci |           | latin1_swedish_ci |
+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.02 sec)
```

- Use drop procedure to remove a stored procedure

```
mysql> drop procedure updateSalary;
Query OK, 0 rows affected (0.00 sec)
```

Language Constructs for Procedures & Functions

- SQL supports constructs that gives it almost all the power of a general-purpose programming language.
 - Warning: most database systems implement their own variant of the standard syntax below.
- Compound statement: **begin ... end**,
 - May contain multiple SQL statements between **begin** and **end**.
 - Local variables can be declared within a compound statements

Language Constructs

■ CASE Statement

```
CASE case_expression
    WHEN when_expression_1 THEN commands
    WHEN when_expression_2 THEN commands
    ...
    ELSE commands
END CASE;
```

■ While and repeat statements:

```
while boolean expression do
    sequence of statements ;
end while
```

```
repeat
    sequence of statements ;
until boolean expression
end repeat
```

Language Constructs (Cont.)

- **Loop, Leave and Iterate statements...**
 - Permits iteration over all results of a query.

```
loop_label: LOOP
    IF x > 10 THEN
        LEAVE loop_label;
    END IF;
    SET x = x + 1;
    IF (x mod 2) THEN
        ITERATE loop_label;
    ELSE
        SET str = CONCAT(str,x,',');
    END IF;
END LOOP;
```



MIT-WPU

॥ विश्वान्तर्द्धुवं ध्रुवा ॥

Cursors

Cursors

- To handle a result set inside a stored procedure, we use a cursor.
- A cursor allows us to iterate a set of rows returned by a query and process each row accordingly.
- The set of rows the cursor holds is referred to as the **active set**.

1. We can declare a cursor by using the DECLARE statement:

```
DECLARE cursor_name CURSOR FOR SELECT_statement;
```

- The cursor declaration must be after any variable declaration.
- A cursor must always be associated with a SELECT statement.

Cursors

2. Next, open the cursor by using the OPEN statement.

```
OPEN cursor_name;
```

3. Then, use the FETCH statement to retrieve the next row pointed by the cursor and move the cursor to the next row in the result set.

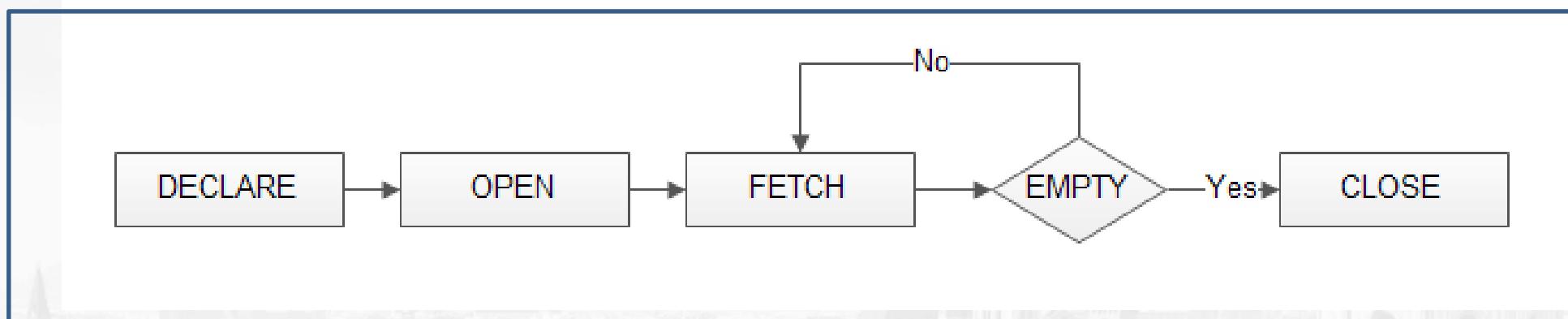
```
FETCH cursor_name INTO variables list;
```

4. Finally, call the CLOSE statement to deactivate the cursor and release the memory associated with it as follows:

```
CLOSE cursor_name;
```

Cursors

The following diagram illustrates how MySQL cursor works.



Example 1 - Cursors

- 1) Retrieve employees one by one and print out a list of those employees currently working in the DEPARTMENT_ID = 80

```
create procedure p_dept()
begin
declare done int default 0;
declare v_eno int;
declare v_ename varchar(10);
declare v_deptno int;
declare c1 cursor for select eno,ename,deptno from employee where deptno=80;
declare continue handler for not found set done =1;
open c1;
repeat
fetch c1 into v_eno,v_ename,v_deptno;
if done=0 then
select v_eno,v_ename,v_deptno;
end if;
until done end repeat;
close c1;
end;

mysql> call p_dept//
```

v_eno	v_ename	v_deptno
1	Anil	80

```
1 row in set (0.00 sec)
```

Example 2

- 2) Use a cursor to retrieve employee numbers and names from employee table and populate a database table, TEMP_LIST, with this information.

```
create procedure p_emp()
begin
declare done int default 0;
declare v_eno int;
declare v_ename varchar(10);
declare v_deptno int;
declare c1 cursor for select eno,ename,deptno from employee ;
declare continue handler for not found set done =1;
open c1;
repeat
fetch c1 into v_eno,v_ename,v_deptno;
if done=0 then
insert into temp_list values(v_eno,v_ename,v_deptno);
end if;
until done end repeat;
close c1;
end;

mysql> call p_emp()//
Query OK, 0 rows affected (0.53 sec)

mysql> select * from temp_list//
```

eno	ename	deptno
1	Anil	80
2	Anita	80
3	Sunita	80
4	Sumita	80
5	Sushmita	10

5 rows in set (0.00 sec)

Example 3

3. Create a PL/SQL block that determines the top employees with respect to salaries. Accept a number n from the user where n represents the number of top n earners from the EMPLOYEES table. For example, to view the top five earners, enter 5. Test a variety of special cases, such as n = 0 or where n is greater than the number of employees in the EMPLOYEES table. The output shown represents the five highest salaries in the EMPLOYEES table.

Cont..

```
create procedure p_top(v_n int)
begin
declare done int default 0;
declare v_eno int;
declare v_ename varchar(10);
declare v_deptno int;
declare v_salary int;
declare v_cnt int default 0;
declare c1 cursor for select eid,ename,dno,salary from emp order by salary desc;
declare continue handler for not found set done =1;
open c1;
repeat
fetch c1 into v_eno,v_ename,v_deptno,v_salary;
if done=0 AND v_cnt<v_n then
select v_eno,v_ename,v_salary;
end if;
set v_cnt=v_cnt+1;
until done end repeat;
close c1;
end;
//////////////////output///////
mysql> call p_top(3)//
+-----+-----+-----+
| v_eno | v_ename | v_salary |
+-----+-----+-----+
|      3 | Sunita  |     51900 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

Example 4

4. Update all the rows in deptsal simultaneously.

First, let's reset the totalsalary in deptsal to zero.

```
mysql> update deptsal set totalsalary = 0;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 3  Changed: 0  Warnings: 0

mysql> select * from deptsal;
+-----+-----+
| dnumber | totalsalary |
+-----+-----+
|      1 |          0 |
|      2 |          0 |
|      3 |          0 |
+-----+-----+
3 rows in set (0.00 sec)
```

Cont..

```
mysql> delimiter $$  
mysql> drop procedure if exists updateSalary$$  
Query OK, 0 rows affected (0.00 sec)  
  
mysql> create procedure updateSalary()  
    begin  
        declare done int default 0;  
        declare current_dnum int;  
        declare dnumcur cursor for select dnumber from deptsal;  
        declare continue handler for not found set done = 1;  
  
        open dnumcur;  
        repeat  
            fetch dnumcur into current_dnum;  
            update deptsal  
                set totalsalary = (select sum(salary) from employee  
                    where dno = current_dnum)  
                    where dnumber = current_dnum;  
            until done  
        end repeat;  
        close dnumcur;  
    end$$  
Query OK, 0 rows affected (0.00 sec)  
  
mysql> delimiter ;
```

Drop the old procedure

Use cursor to iterate the rows

Cont..

- Call procedure :

```
mysql> select * from deptsal;
+-----+-----+
| dnumber | totalsalary |
+-----+-----+
|      1 |          0 |
|      2 |          0 |
|      3 |          0 |
+-----+-----+
3 rows in set (0.01 sec)

mysql> call updateSalary;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from deptsal;
+-----+-----+
| dnumber | totalsalary |
+-----+-----+
|      1 |    100000 |
|      2 |     50000 |
|      3 |    130000 |
+-----+-----+
3 rows in set (0.00 sec)
```

Example 5

5. Create a procedure to give a raise to all employees

```
mysql> select * from emp;
+----+-----+-----+-----+-----+
| id | name  | superid | salary | bdate      | dno |
+----+-----+-----+-----+-----+
|  1 | john   |      3 | 100000 | 1960-01-01 |   1 |
|  2 | mary   |      3 | 50000  | 1964-12-01 |   3 |
|  3 | bob    |    NULL | 80000  | 1974-02-07 |   3 |
|  4 | tom    |      1 | 50000  | 1978-01-17 |   2 |
|  5 | bill   |    NULL |    NULL | 1985-01-20 |   1 |
|  6 | lucy   |    NULL | 90000  | 1981-01-01 |   1 |
|  7 | george |    NULL | 45000  | 1971-11-11 | NULL|
+----+-----+-----+-----+-----+
7 rows in set (0.00 sec)
```

Cont..

```
mysql> delimiter |
mysql> create procedure giveRaise (in amount double)
      -> begin
      ->     declare done int default 0;
      ->     declare eid int;
      ->     declare sal int;
      ->     declare emprec cursor for select id, salary from employee;
      ->     declare continue handler for not found set done = 1;
      ->
      ->     open emprec;
      ->     repeat
      ->         fetch emprec into eid, sal;
      ->         update employee
      ->             set salary = sal + round(sal * amount)
      ->             where id = eid;
      ->     until done
      ->     end repeat;
      -> end |
Query OK, 0 rows affected (0.00 sec)
```

Cont..

```
mysql> delimiter ;
mysql> call giveRaise(0.1);
Query OK, 0 rows affected (0.00 sec)

mysql> select * from employee;
+----+-----+-----+-----+-----+-----+
| id | name | superid | salary | bdate      | dno |
+----+-----+-----+-----+-----+-----+
| 1  | john |       3 | 110000 | 1960-01-01 | 1   |
| 2  | mary |       3 | 55000  | 1964-12-01 | 3   |
| 3  | bob  |     NULL | 88000  | 1974-02-07 | 3   |
| 4  | tom  |       1 | 55000  | 1978-01-17 | 2   |
| 5  | bill |     NULL |    NULL | 1985-01-20 | 1   |
+----+-----+-----+-----+-----+
5 rows in set (0.00 sec)
```

Triggers

Triggers

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database i.e. when changes are made to the table.
- To monitor a database and take a corrective action when a condition occurs

Examples:

- Charge \$10 overdraft fee if the balance of an account after a withdrawal transaction is less than \$500
 - Limit the salary increase of an employee to no more than 5% raise
- SQL triggers provide an alternative way to check the integrity of data.

Triggering Events and Actions in SQL

- A trigger can be defined to be invoked either before or after the data is changed by INSERT, UPDATE or DELETE.
- MySQL allows you to define maximum six triggers for each table.
 - **BEFORE INSERT** – activated before data is inserted into the table.
 - **AFTER INSERT** – activated after data is inserted into the table.
 - **BEFORE UPDATE** – activated before data in the table is updated.
 - **AFTER UPDATE** – activated after data in the table is updated.
 - **BEFORE DELETE** – activated before data is removed from the table.
 - **AFTER DELETE** – activated after data is removed from the table.

MySQL Trigger Syntax

```
1 CREATE TRIGGER trigger_name trigger_time trigger_event
2   ON table_name
3     FOR EACH ROW
4 BEGIN
5   ...
6 END;
```

Cont..

- In a trigger defined for **INSERT**, you can use NEW keyword only. You cannot use the OLD keyword.
- However, in the trigger defined for **DELETE**, there is no new row so you can use the OLD keyword only.
- In the **UPDATE** trigger, OLD refers to the row before it is updated and NEW refers to the row after it is updated.

Example 1 - Trigger

1. Create a trigger to simulate Recycle Bin for employee table. If any row gets deleted from Emp, same row must get stored in Emp_Bin

Emp(Eno,Ename,Salary)

Emp_Bin(Eno,Ename,Salary)

```
create trigger t_bin
before delete on emp for each row
begin
insert into temp_emp values(OLD.eid,OLD.ename,OLD.dno,OLD.salary);
end;

mysql> delete from emp where eid=1//  
Query OK, 1 row affected (0.16 sec)

mysql> select * from temp_emp//  
+-----+-----+-----+-----+
| eid | ename | dno | salary |
+-----+-----+-----+-----+
|    1 | Anil |   10 | 10900 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

Example 2

2. Create a BEFORE UPDATE trigger that is invoked before a change is made to the employees table.

- we used the OLD keyword to access employeeNumber and lastname column of the row affected by the trigger.

```
1 DELIMITER $$  
2 CREATE TRIGGER before_employee_update  
3     BEFORE UPDATE ON employees  
4     FOR EACH ROW  
5 BEGIN  
6     INSERT INTO employees_audit  
7     SET action = 'update',  
8         employeeNumber = OLD.employeeNumber,  
9         lastname = OLD.lastname,  
10        changedat = NOW();  
11    END$$  
12 DELIMITER ;
```

Cont..

- Update the employees table to check whether the trigger is invoked.

```
1 UPDATE employees
2 SET
3     lastName = 'Phan'
4 WHERE
5     employeeNumber = 1056;
```

- Finally, to check if the trigger was invoked by the UPDATE statement, we can query the employees_audit table using the following query:

```
1 SELECT
2     *
3 FROM
4     employees_audit;
```

Cont..

- The following is the output of the query:

	id	employeeNumber	lastname	changedat	action
▶	1	1056	Phan	2015-11-14 21:39:12	update

Example 3

3. We want to create a trigger to update the total salary of a department when a new employee is hired.

```
mysql> select * from employee;
+----+-----+-----+-----+-----+-----+
| id | name | superid | salary | bdate | dno |
+----+-----+-----+-----+-----+-----+
| 1  | john |         3 | 100000 | 1960-01-01 | 1   |
| 2  | mary |         3 | 50000  | 1964-12-01 | 3   |
| 3  | bob  |        NULL | 80000  | 1974-02-07 | 3   |
| 4  | tom  |         1 | 50000  | 1970-01-17 | 2   |
| 5  | bill |        NULL | NULL   | 1985-01-20 | 1   |
+----+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)

mysql> select * from deptsal;
+-----+-----+
| dnumber | totalsalary |
+-----+-----+
| 1       | 100000      |
| 2       | 50000       |
| 3       | 130000      |
+-----+-----+
3 rows in set (0.00 sec)
```

Cont..

Create a trigger to update the total salary of a department when a new employee is hired.

```
mysql> delimiter !
mysql> create trigger update_salary
-> after insert on employee
-> for each row
-> begin
->     if new.dno is not null then
->         update deptsal
->             set totalsalary = totalsalary + new.salary
->             where dnumber = new.dno;
->     end if;
-> end ;
Query OK, 0 rows affected (0.06 sec)

mysql> delimiter ;
```

- The keyword “new” refers to the new row inserted

Cont..

```
mysql> select * from deptsal;
+-----+-----+
| dnumber | totalsalary |
+-----+-----+
|      1  |     100000 |
|      2  |      50000 |
|      3  |    130000 |
+-----+-----+
3 rows in set <0.00 sec>

mysql> insert into employee values (6,'lucy',null,90000,'1981-01-01',1);
Query OK, 1 row affected <0.08 sec>

mysql> select * from deptsal;
+-----+-----+
| dnumber | totalsalary |
+-----+-----+
|      1  |     190000 |
|      2  |      50000 |
|      3  |    130000 |
+-----+-----+
3 rows in set <0.00 sec>

mysql> insert into employee values (7,'george',null,45000,'1971-11-11',null);
Query OK, 1 row affected <0.02 sec>

mysql> select * from deptsal;
+-----+-----+
| dnumber | totalsalary |
+-----+-----+
|      1  |     190000 |
|      2  |      50000 |
|      3  |    130000 |
+-----+-----+
3 rows in set <0.00 sec>

mysql> drop trigger update_salary;
Query OK, 0 rows affected <0.00 sec>
```

totalsalary increases by 90K

totalsalary did not change

Trigger

- To list all the triggers we have created:

```
mysql> show triggers;
```

```
1 SHOW TRIGGERS;
```

Trigger	Event	Table	Statement	Timing	Created	sql_mode	Definer
before_employee_update	UPDATE	employees	BEGIN INSERT INTO employ... BEFORE	2015-11-14 21:39:09.08	STRICT_TRANS_TABLES,NO_AUTO_CREATE_U...	root@localhost	

- To drop a trigger

```
mysql> drop trigger <trigger name>
```

XML (Extensible Markup Language)

XML is a markup language similar to HTML, but without predefined tags to use. Instead, you define your own tags designed specifically for your needs. This is a powerful way to store data in a format that can be stored, searched, and shared.

- Structure of XML Data
- XML Document Schema
- Querying and Transformation
- Application Program Interfaces to XML
- Storage of XML Data
- XML Applications



- The ability to specify new tags, and to create nested tag structures make XML a great way to exchange **data**, not just documents.
 - Much of the use of XML has been in data exchange applications, not as a replacement for HTML
- Tags make data (relatively) self-documenting
 - E.g.

```
<university>
  <department>
    <dept_name> Comp. Sci. </dept_name>
    <building> Taylor </building>
    <budget> 100000 </budget>
  </department>
  <course>
    <course_id> CS-101 </course_id>
    <title> Intro. to Computer Science </title>
    <dept_name> Comp. Sci </dept_name>
    <credits> 4 </credits>
  </course>
</university>
```

Structure of XML data

- **Tag**: label for a section of data
- **Element**: section of data beginning with `<tagname>` and ending with matching `</tagname>`
- Elements must be properly **nested**
 - Proper nesting
 - `<course> ... <title> </title> </course>`
 - Improper nesting
 - `<course> ... <title> </course> </title>`
 - Formally: every start tag must have a unique matching end tag, that is in the context of the same parent element.
- Every document must have a single top-level element

XML Element

- An element can contain:
 - other elements
 - text
 - attributes
 - or a mix of all of the above...

Attributes

- Elements can have **attributes**

```
<course course_id= "CS-101">  
    <title> Intro. to Computer Science</title>  
    <dept name> Comp. Sci. </dept name>  
    <credits> 4 </credits>  
</course>
```

- Attributes are specified by *name=value* pairs inside the starting tag of an element
- An element may have several attributes, but each attribute name can only occur once

```
<course course_id = "CS-101" credits="4">
```

Attributes vs. Subelements

- Distinction between subelement and attribute
 - In the context of documents, attributes are part of markup, while subelement contents are part of the basic document contents
 - In the context of data representation, the difference is unclear and may be confusing
 - Same information can be represented in two ways
 - <course course_id= “CS-101”> ... </course>
 - <course>
 <course_id>CS-101</course_id> ...
 </course>
 - Suggestion: use attributes for identifiers of elements, and use subelements for contents

Namespaces

- XML data has to be exchanged between organizations
- Same tag name may have different meaning in different organizations, causing confusion on exchanged documents
- Specifying a unique string as an element name avoids confusion
- Better solution: use **unique-name:element-name**
- Avoid using long unique names all over document by using XML Namespaces

```
<university xmlns:yale="http://www.yale.edu">  
    ...  
    <yale:course>  
        <yale:course_id> CS-101 </yale:course_id>  
        <yale:title> Intro. to Computer Science</yale:title>  
        <yale:dept_name> Comp. Sci. </yale:dept_name>  
        <yale:credits> 4 </yale:credits>  
    </yale:course>  
    ...  
</university>
```

Document Type Definition (DTD)

- The type of an XML document can be specified using a DTD
- DTD constraints structure of XML data
 - What elements can occur
 - What attributes can/must an element have
 - What subelements can/must occur inside each element, and how many times.
- DTD does not constrain data types
 - All values represented as strings in XML
- DTD syntax
 - `<!ELEMENT element (subelements-specification) >`
 - `<!ATTLIST element (attributes) >`

Element Specification in DTD

- Subelements can be specified as
 - names of elements, or
 - #PCDATA (parsed character data), i.e., character strings
 - EMPTY (no subelements) or ANY (anything can be a subelement)
- Example

```
<! ELEMENT department (dept_name building, budget)>
<! ELEMENT dept_name (#PCDATA)>
<! ELEMENT budget (#PCDATA)>
```

- Subelement specification may have regular expressions

```
<!ELEMENT university ( ( department | course | instructor | teaches )+)>
```

 - Notation:
 - “|” - alternatives
 - “+” - 1 or more occurrences
 - “*” - 0 or more occurrences

University DTD

```
<!DOCTYPE university [  
    <!ELEMENT university ( (department|course|instructor|teaches)+)>  
    <!ELEMENT department ( dept name, building, budget)>  
    <!ELEMENT course ( course id, title, dept name, credits)>  
    <!ELEMENT instructor (IID, name, dept name, salary)>  
    <!ELEMENT teaches (IID, course id)>  
    <!ELEMENT dept name( #PCDATA )>  
    <!ELEMENT building( #PCDATA )>  
    <!ELEMENT budget( #PCDATA )>  
    <!ELEMENT course id ( #PCDATA )>  
    <!ELEMENT title ( #PCDATA )>  
    <!ELEMENT credits( #PCDATA )>  
    <!ELEMENT IID( #PCDATA )>  
    <!ELEMENT name( #PCDATA )>  
    <!ELEMENT salary( #PCDATA )>  
]>
```

Attribute Specification in DTD

- Attribute specification : for each attribute
 - Name
 - Type of attribute
 - ▶ CDATA
 - ▶ ID (identifier) or IDREF (ID reference) or IDREFS (multiple IDREFs)
 - more on this later
 - Whether
 - ▶ mandatory (#REQUIRED)
 - ▶ has a default value (value),
 - ▶ or neither (#IMPLIED)

Attribute Specification in DTD : examples

■ Examples

- `<!ATTLIST course course_id CDATA #REQUIRED>`, or
- `<!ATTLIST course
course_id ID #REQUIRED
dept_name IDREF #REQUIRED
instructors IDREFS #IMPLIED >`

IDs and IDREFs

- An element can have at most one attribute of type ID
- The ID attribute value of each element in an XML document must be distinct
 - Thus the ID attribute value is an object identifier
- An attribute of type IDREF must contain the ID value of an element in the same document
- An attribute of type IDREFS contains a set of (0 or more) ID values. Each ID value must contain the ID value of an element in the same document

University DTD with Attributes

- University DTD with ID and IDREF attribute types.

```
<!DOCTYPE university-3 [  
    <!ELEMENT university ( (department|course|instructor)+)>  
    <!ELEMENT department ( building, budget )>  
    <!ATTLIST department  
        dept_name ID #REQUIRED >  
    <!ELEMENT course (title, credits )>  
    <!ATTLIST course  
        course_id ID #REQUIRED  
        dept_name IDREF #REQUIRED  
        instructors IDREFS #IMPLIED >  
    <!ELEMENT instructor ( name, salary )>  
    <!ATTLIST instructor  
        IID ID #REQUIRED  
        dept_name IDREF #REQUIRED >  
    · · · declarations for title, credits, building,  
        budget, name and salary · · ·  
]>
```

XML data with ID and IDREF attributes

```
<university-3>
    <department dept name="Comp. Sci.">
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department dept name="Biology">
        <building> Watson </building>
        <budget> 90000 </budget>
    </department>
    <course course id="CS-101" dept name="Comp. Sci"
            instructors="10101 83821">
        <title> Intro. to Computer Science </title>
        <credits> 4 </credits>
    </course>
    ...
    <instructor IID="10101" dept name="Comp. Sci.">
        <name> Srinivasan </name>
        <salary> 65000 </salary>
    </instructor>
    ...
</university-3>
```



XML Schema

- XML Schema is a more sophisticated schema language which addresses the drawbacks of DTDs. Supports
 - Typing of values
 - E.g. integer, string, etc
 - Also, constraints on min/max values
 - User-defined, complex types
 - Many more features, including
 - uniqueness and foreign key constraints, inheritance
- XML Schema is itself specified in XML syntax, unlike DTDs
 - More-standard representation, but verbose
- XML Schema is integrated with namespaces
- BUT: XML Schema is significantly more complicated than DTDs.

XML Schema Version of Univ. DTD

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="university" type="universityType" />
<xs:element name="department">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="dept name" type="xs:string"/>
            <xs:element name="building" type="xs:string"/>
            <xs:element name="budget" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
.....
<xs:element name="instructor">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="IID" type="xs:string"/>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="dept name" type="xs:string"/>
            <xs:element name="salary" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
Contd.
```



XML Schema Version of Univ. DTD (Cont.)

```
....  
<xs:complexType name="UniversityType">  
  <xs:sequence>  
    <xs:element ref="department" minOccurs="0" maxOccurs="unbounded"/>  
    <xs:element ref="course" minOccurs="0" maxOccurs="unbounded"/>  
    <xs:element ref="instructor" minOccurs="0" maxOccurs="unbounded"/>  
    <xs:element ref="teaches" minOccurs="0" maxOccurs="unbounded"/>  
  </xs:sequence>  
</xs:complexType>  
</xs:schema>
```

- Choice of “xs:” was ours -- any other namespace prefix could be chosen
- Element “university” has type “universityType”, which is defined separately
 - xs:complexType is used later to create the named complex type “UniversityType”



More features of XML Schema

- Attributes specified by xs:attribute tag:
 - <xs:attribute name = "dept_name"/>
 - adding the attribute use = "required" means value must be specified
- Key constraint: "department names form a key for department elements under the root university element:

```
<xs:key name = "deptKey">
    <xs:selector xpath = "/university/department"/>
    <xs:field xpath = "dept_name"/>
</xs:key>
```
- Foreign key constraint from course to department:

```
<xs:keyref name = "courseDeptFKey" refer="deptKey">
    <xs:selector xpath = "/university/course"/>
    <xs:field xpath = "dept_name"/>
</xs:keyref>
```

Querying and Transforming XML Data

- Translation of information from one XML schema to another
- Querying on XML data
- Above two are closely related, and handled by the same tools
- Standard XML querying/translation languages
 - XPath
 - Simple language consisting of path expressions
 - XSLT
 - Simple language designed for translation from XML to XML and XML to HTML
 - XQuery
 - An XML query language with a rich set of features

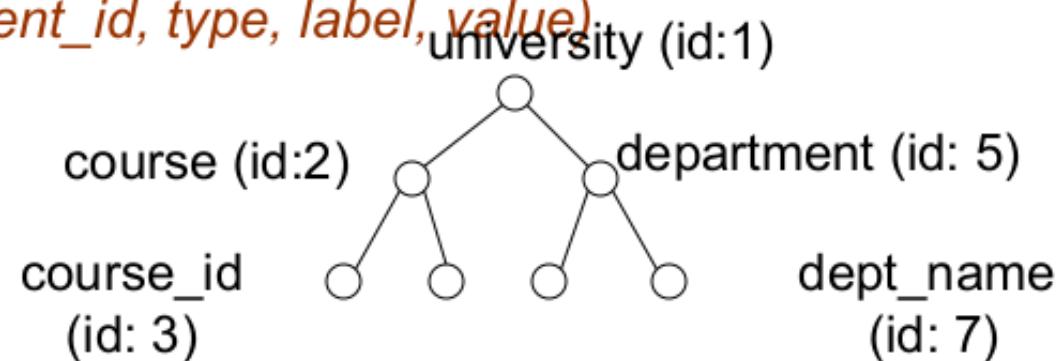
Tree Model of XML Data

- Query and transformation languages are based on a **tree model** of XML data
- An XML document is modeled as a tree, with **nodes** corresponding to elements and attributes
 - Element nodes have child nodes, which can be attributes or subelements
 - Text in an element is modeled as a text node child of the element
 - Children of a node are ordered according to their order in the XML document
 - Element and attribute nodes (except for the root node) have a single parent, which is an element node
 - The root node has a single child, which is the root element of the document

Tree Representation

- **Tree representation:** model XML data as tree and store using relations

nodes(id, parent_id, type, label, value)



- Each element/attribute is given a unique identifier
- Type indicates element/attribute
- Label specifies the tag name of the element/name of attribute
- Value is the text value of the element/attribute
- Can add an extra attribute *position* to record ordering of children

XPath

- XPath is used to address (select) parts of documents using **path expressions**
- A path expression is a sequence of steps separated by “/”
 - Think of file names in a directory hierarchy
- Result of path expression: set of values that along with their containing elements/attributes match the specified path
- E.g. **/university-3/instructor/name** evaluated on the university-3 data we saw earlier returns
 - <name>Srinivasan</name>
 - <name>Brandt</name>
- E.g. **/university-3/instructor/name/text()**
 - returns the same names, but without the enclosing tags

XPath (Cont.)

- The initial “/” denotes root of the document (above the top-level tag)
- Path expressions are evaluated left to right
 - Each step operates on the set of instances produced by the previous step
- Selection predicates may follow any step in a path, in []
 - E.g. `/university-3/course[credits >= 4]`
 - ▶ returns account elements with a balance value greater than 400
 - ▶ `/university-3/course[credits]` returns account elements containing a credits subelement
- Attributes are accessed using “@”
 - E.g. `/university-3/course[credits >= 4]/@course_id`
 - ▶ returns the course identifiers of courses with credits ≥ 4
 - IDREF attributes are not dereferenced automatically (more on this later)

Functions in XPath

- XPath provides several functions
 - The function **count()** at the end of a path counts the number of elements in the set generated by the path
 - E.g. **/university-2/instructor[count(./teaches/course)> 2]**
 - Returns instructors teaching more than 2 courses (on university-2 schema)
 - Also function for testing position (1, 2, ..) of node w.r.t. siblings
 - Boolean connectives **and** and **or** and function **not()** can be used in predicates
 - IDREFs can be referenced using function **id()**
 - **id()** can also be applied to sets of references such as IDREFS and even to strings containing multiple references separated by blanks
 - E.g. **/university-3/course/id(@dept_name)**
 - returns all department elements referred to from the dept_name attribute of course elements.

More XPath Features

- Operator “|” used to implement union
 - E.g. `/university-3/course[@dept name=“Comp. Sci”] | /university-3/course[@dept name=“Biology”]`
 - ▶ Gives union of Comp. Sci. and Biology courses
 - ▶ However, “|” cannot be nested inside other operators.
- “//” can be used to skip multiple levels of nodes
 - E.g. `/university-3//name`
 - ▶ finds any **name** element *anywhere* under the `/university-3` element, regardless of the element in which it is contained.
- A step in the path can go to parents, siblings, ancestors and descendants of the nodes generated by the previous step, not just to the children
 - “//”, described above, is a short form for specifying “all descendants”
 - “..” specifies the parent.
- `doc(name)` returns the root of a named document

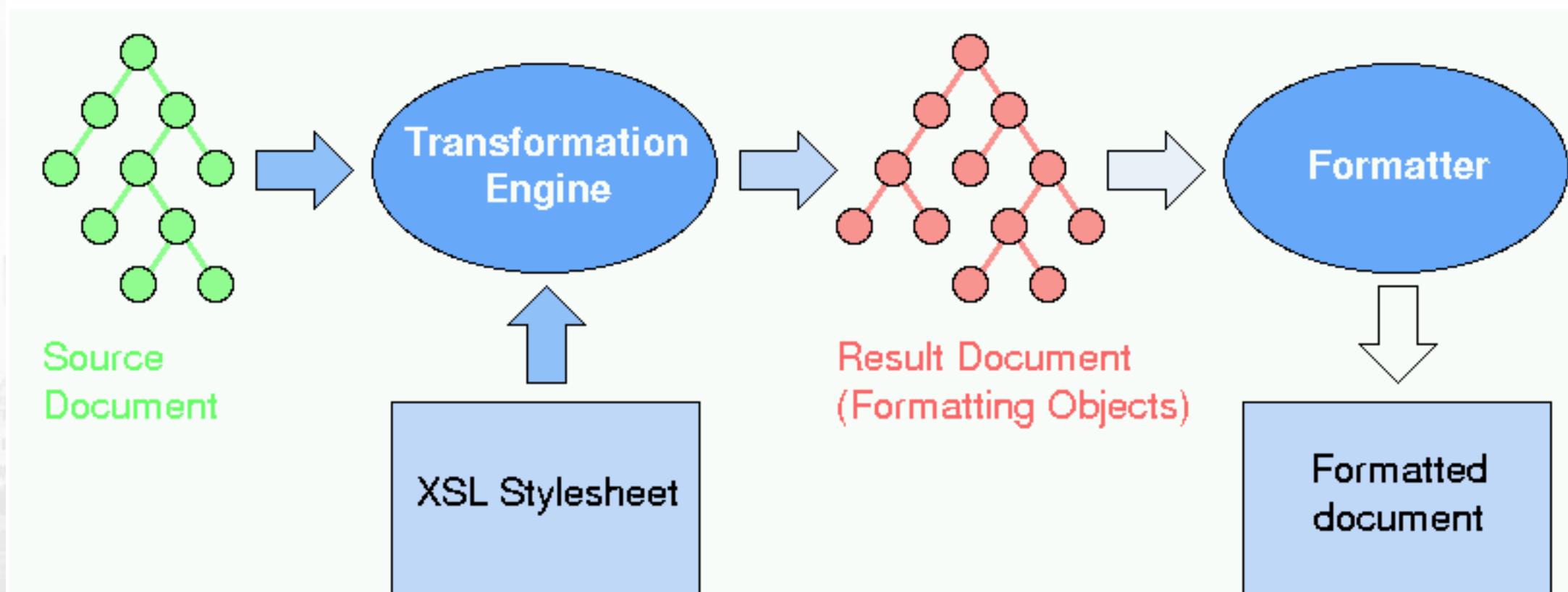


Extensible Stylesheet Language (XSL)

XSL is a language for expressing stylesheets

- support for browsing, printing, and aural rendering
- formatting highly structured documents (XML)
- performing complex publishing tasks: tables of contents, indexes, reports,...
- addressing accessibility and internationalization issues
- written in XML

XSL Architecture



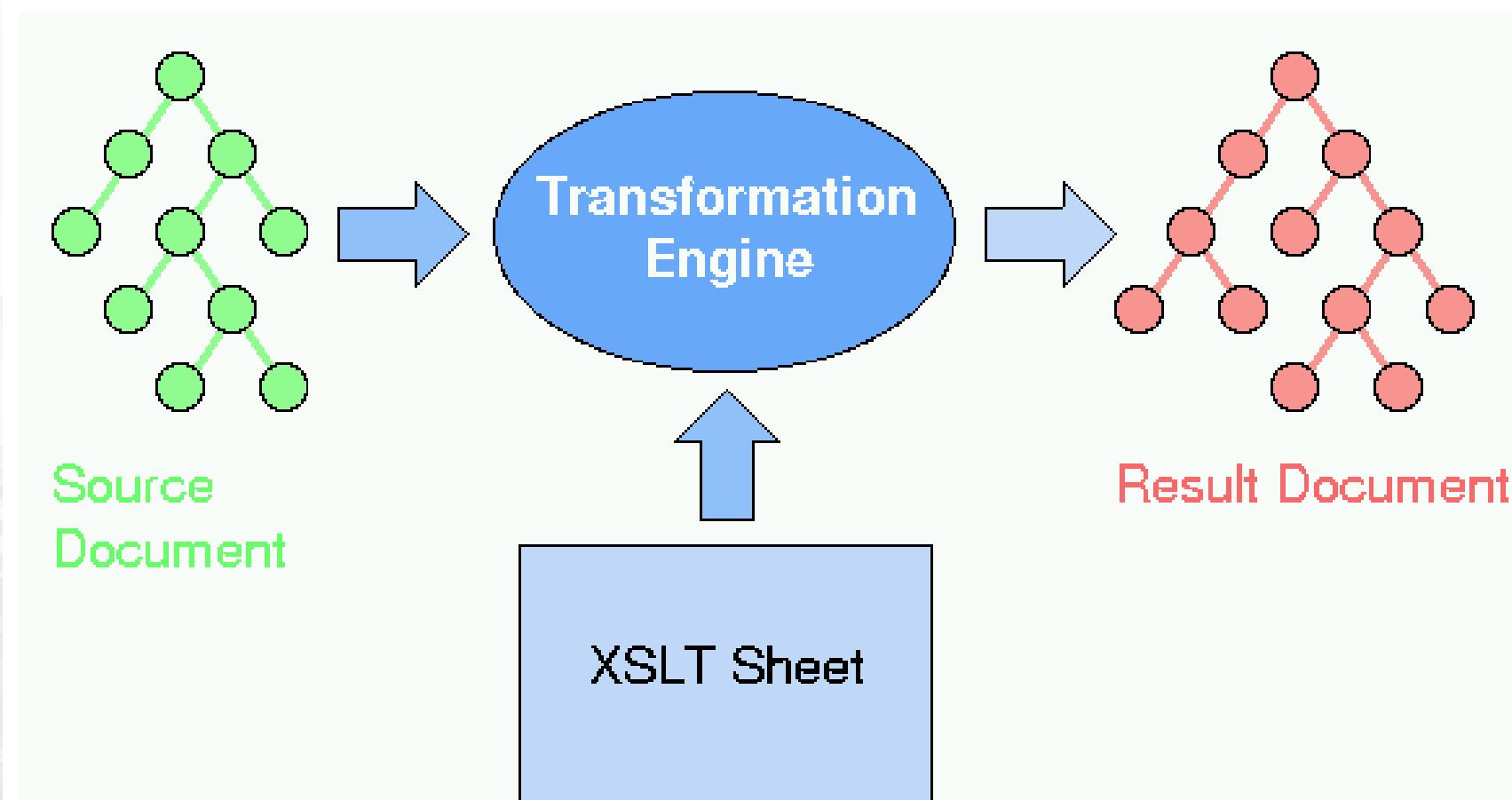
XSL Components

XSL is constituted of three main components:

- **XSLT**: a transformation language
- **XPath**: an expression language for addressing parts of XML documents
- **FO**: a vocabulary of *formatting objects* with their associated formatting properties

XSL uses XSLT which uses XPath

XSL Transformations



XSLT - Basic Principle

Patterns and Templates

- A style sheets describes transformation rules
- A transformation rule: a pattern + a template
- Pattern: a configuration in the source tree
- Template: a structure to be instantiated in the result tree
- When a pattern is matched in the source tree, the corresponding pattern is generated in the result tree

An Example: Transformation

```
<xsl:template match="Title">
  <H1>
    <xsl:apply-templates/>
  </H1>
</xsl:template>
```

Input : <Title>Introduction</Title>

Output : <H1>Introduction</H1>

An Example: Formatting

```
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:fo="http://www.w3.org/1999/XSL/Format"
    result-ns="fo">
    <xsl:template match="/">
        <fo:page-sequence font-family="serif">
            <xsl:apply-templates/>
        </fo:page-sequence>
    </xsl:template>
    <xsl:template match="para">
        <fo:block font-size="10pt" space-before="12pt">
            <xsl:apply-templates/>
        </fo:block>
    </xsl:template>
</xsl:stylesheet>
```

XPath: XML Path Language

An elementary XPath expression contains

- an axis, which specifies the tree relationship: child, descendants, ancestors, siblings, attributes,...
- a node test, which specifies the node type
- predicates, to further refine the set of nodes selected

Example: all *para* children that have a *type* attribute with value warning

child::para[attribute::type="warning"]

XSL Usage

- Format XML documents by generating FOs
- Generate HTML or XHTML pages from XML data/documents
- Transform XML documents into other XML documents
- Generate some textual representation of an XML document
- ...and more

XSL may be used server-side or client-side,
but is not intended to send FOs over the wire

JSON

- 1) **JSON** (JavaScript Object Notation) is a lightweight data-interchange format.
- 2) JSON is a syntax for storing and exchanging data.
- 3) JSON is an easier-to-use alternative to XML.
- 4) It is based on a subset of the JavaScript Programming Language

Data Types

- Strings
 - Sequence of 0 or more Unicode characters
 - Wrapped in "double quotes"
 - Backslash escapement
- Numbers
 - Integer
 - Real
 - Scientific
 - No octal or hex
 - No None or Infinity – Use null instead.



- Booleans & Null
 - Booleans: true or false
 - Null: A value that specifies nothing or no value.
- Objects & Arrays
 - Objects: Unordered key/value pairs wrapped in { }
 - Arrays: Ordered key/value pairs wrapped in []

JSON Object Syntax

- Unordered sets of name/value pairs (hash/dictionary)
 - Begins with { (left brace)
 - Ends with } (right brace)
 - Each name is followed by : (colon)
 - Name/value pairs are separated by , (comma)
 - Commas are used to separate multiple data values.
 - Objects are enclosed within curly braces.
 - square brackets are used to store arrays.
 - Json keys must be enclosed within double quotes.

Example:

```
{ "employee_id": 1234567, "name": "Jeff Fox", "hire_date": "1/1/2013", "location":  
"Norwalk, CT", "consultant": false }
```

Arrays in JSON

- An ordered collection of values
 - Begins with [(left bracket)
 - Ends with] (right bracket)
 - Name/value pairs are separated by , (comma)

Example:

```
{ "employee_id": 1236937, "name": "Jeff Fox", "hire_date": "1/1/2013", "location": "Norwalk, CT", "consultant": false, "random_nums": [ 24,65,12,94 ] }
```

How & When to use JSON

- Transfer data to and from a server(ex: Browser, mobile Apps)
- Perform asynchronous data calls without requiring a page refresh
- Working with data stores
- Compile and save form or user data for local storage

Thank You!