CS245: Databases

SQL

Vijaya saradhi

Department of Computer Science and Engineering
Indian Institute of Technology Guwahati

Cursor - I

Impedance Model Mis-match

- SQL always returns relations
- Other programming languages has data types that are not relations
- These languages cannot hold relations returned by SQL
- C language has pointers; where as SQL do not have any such construct
- As a result, passing data between SQL and other languages is not straightforward
- Mechanisms must be devised to allow the development of programs that use both SQL and other languages

Cursor - I

Impedance Model Mis-match

- Versatile way to connect SQL queries to a host language is with a cursor
- Cursor runs through the tuples of a relation
- This relation can be stored table, or it can be something that is generated by a query

Cursor - I

Details

- SELECT will return a relation
- Returned relation will not be stored
- Often the need to process one row at a time of returned relation arise
- Cursor helps examining one row at a time

Cursor - II

Details

- Assume the returned relation to be a file in itself
- Operations required for reading a file are
  - Declare file pointer
  - Open the file
  - Read one line at a time repeatedly
  - close the file
- Similar tasks are associated with cursor

Cursor - III

Declare cursor

```
DECLARE cursor_name CURSOR FOR SELECT statement;

OPEN  cursor_name;

FETCH cursor_name INTO variable_list;

CLOSE cursor_name
```

Cursor - Example

Example

```
DELIMITER //
CREATE PROCEDURE f11()
BEGIN
-- Declare variables
        DECLARE i INT DEFAULT 1;
        DECLARE sno INT;
        DECLARE sname char(50);
        DECLARE rating INT DEFAULT 10;
        DECLARE age INT DEFAULT 16;

-- Declare cursors
        DECLARE my_first_cursor CURSOR FOR
        SELECT   *
        FROM     Sailors
        WHERE    age > 20 AND rating BETWEEN 5 AND 7;

-- Declare cursor handler
        DECLARE CONTINUE HANDLER FOR NOT FOUND SET NO_records = 1;
```

Cursor - Example

Example

```
  OPEN my_first_cursor;

-- loop through all the rows
loop_1: REPEAT
-- Get one roll number from list of registered students into variable rn
    FETCH my_first_cursor INTO (sno, sname, rating, age);
-- Check number of records in the cursor
    IF NO_records = 1 THEN
      LEAVE loop_1;
    END IF;

    UNTIL NO_records
  END REPEAT loop_1;
  CLOSE my_first_cursor;
END; //
DELIMITER ;
```

Cursor - IV

Scrolling

- Cursor gives us flexibility as how to move through the tuples of the relation
- The default choice is to start at the beginning of the relation and fetch the tuples in order
- Fetch all tuples until end of the relation
- Other orders in which tuples may be fetched
- These options are not available in `MySQL` yet we will discuss these

Cursor - V

Scrolling

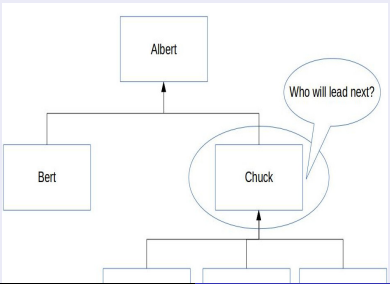
- Instruct the cursor to open in `SCROLL` model before the keyword `CURSOR`
- `EXEC SQL DECLARE name SCROLL CURSOR FOR MovieExec;`
- This will tell SQL that cursor may be used in a manner other than moving in forward direction alone
- The FETCH is responsible for specifying the direction from which the next tuple be obtained
  - `FETCH NEXT` retrieve next tuple
  - `FETCH PRIOR` retrieve previous tuple
  - `FETCH FIRST` retrieve first tuple
  - `FETCH LAST` retrieve last tuple
  - `FETCH ABSOLUTE i` specifies the position of the tuple to be fetched from the top of the relation

Supervisor-supervisee

Manages Relation

Employee	Boss	Salary
Albert	$\perp$	1000.00
Bert	Albert	900.00
Chuck	Albert	900.00
Donna	Chuck	800.00
Eddie	Chuck	700.00
Fred	Chuck	600.00

Manages Relation



Supervisor-supervisee

Anomalies

INSERT	Can include cycles in the graph
UPDATE	
DELETE	
Structural	

Insertion Anomaly Example

Employee	Boss	Salary
Albert	$\perp$	1000.00
Albert	Fred	100.00
Bert	Albert	900.00
Chuck	Albert	900.00
Donna	Chuck	800.00
Eddie	Chuck	700.00
Fred	Chuck	600.00

Supervisor-supervisee

Anomalies

INSERT	Can include cycles in the graph
UPDATE	`UPDATE manager set Employee='Charles' where Employee = 'Chuck';`
DELETE	
Structural	

UPDATE Anomaly Example

Employee	Boss	Salary
Albert	$\perp$	1000.00
Bert	Albert	900.00
Charles	Albert	900.00
Donna	Chuck	800.00
Eddie	Chuck	700.00
Fred	Chuck	600.00

In atomic fashion

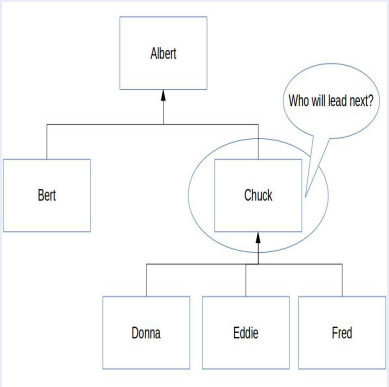
```
UPDATE manager set Employee='Charles' where Employee =
'Chuck';
UPDATE manager set Boss='Charles' where Boss = 'Chuck';
```

Supervisor-supervisee

Anomalies

INSERT	Can include cycles in the graph
UPDATE	`UPDATE manager set Employee='Charles' where Employee = 'Chuck';`
DELETE	Chuck left the organization. What should be the right way?
Structural	

DELETE Anomaly Example



Supervisor-supervisee

Structural Anomalies

- `INSERT INTO Manager (Employee, Boss) VALUES ('a', 'a');`
- Create simple cycles
- `INSERT INTO Manager (Employee, Boss) VALUES ('b', 'c');`
- `INSERT INTO Manager (Employee, Boss) VALUES ('c', 'b');`

Supervisor-supervisee: Solution - Part I

Modify relation

- Employee details and organization hierarchy must be separated
- Create table for `Employee(eid, ename, address)`
- Create table for hierarchy `Manages(role, eid, boss_eid)`
- role should be primary key
- (eid, boss_eid) should be unique
- eid should be foreign key referring Employee
- eid default value should be 0 to indicate vacant position
- eid should not be NULL

Supervisor-supervisee: Solution - Part II

Constraints

- Self boss is not allowed. `CHECK(eid <> boss_eid);`
- boss_eid and eid should not be 0; `CHECK( boss_eid != 0 AND eid != 0 )`
- Number of nodes in tree: `SELECT COUNT(*) FROM Manages`
- Number of edges in tree: `SELECT COUNT(boss_eid) FROM Manages`
- Number of edges = number of nodes - 1; `CHECK( (SELECT COUNT(*) FROM Manages) - 1 = (SELECT COUNT(boss_eid) FROM Manages) )`
- Only one root:
  `CHECK(SELECT COUNT(*) FROM Manages where ISNULL(boss_eid) = 1)`

Supervisor-supervisee: Solution - Part III

Constraints - Check for Cycles

```
1 CREATE FUNCTION TreeTest() RETURNS CHAR(6)
2 BEGIN ATOMIC
3 -- put a copy in a temporary table
4       INSERT INTO Tree SELECT eid, boss_id FROM Manages
5
6 -- prune the leaves
7         WHILE ((SELECT COUNT(*) FROM Tree) - 1) = (SELECT COUNT(boss_id) FROM Tree)
8         DO
9                 DELETE FROM Tree
10              -- Check employee is not the boss
11              WHERE Tree.eid
12              NOT IN (
13                      -- Select all the bosses
14                      SELECT T2.boss_id
15                      FROM Tree AS T2
16                      WHERE NOT ISNULL(T2.boss_id)
17              );
18
19    IF NOT EXISTS (SELECT * FROM Tree)
20    THEN
21      RETURN ('Tree');
20____ELSE
23______RETURN_('Cycles');
24    END IF;
25  END WHILE;
```

Supervisor-supervisee: Steps

Detailed Steps

```
Iteration #1
-------------------------------------------------------------------------------
Albert Not in {Albert, Albert, Chuck, Chuck, Chuck}? No;
Bert  Not in {Albert, Albert, Chuck, Chuck, Chuck}? Yes; Delete
Chuck Not in {Albert, Albert, Chuck, Chuck, Chuck}? No;
Donna Not in {Albert, Albert, Chuck, Chuck, Chuck}? Yes; Delete
Eddie Not in {Albert, Albert, Chuck, Chuck, Chuck}? Yes; Delete
Fred  Not in {Albert, Albert, Chuck, Chuck, Chuck}? Yes; Delete
-------------------------------------------------------------------------------
```

Supervisor-supervisee: Steps

Detailed Steps

```
Iteration #2
-------------------------------------------------------------------------
Albert NULL
Chuck  Albert

Albert  Not in {Albert} No;
Chuck   Not in {Albert} Yes; Delete
-------------------------------------------------------------------------
```

Supervisor-supervisee: Steps

Detailed Steps

```
Iteration #3
-------------------------------------------------------------------------------
Albert NULL
Albert  Not in {} Yes; Delete
-------------------------------------------------------------------------------
```

Recursion

Use case

- A common kind of operation on data is that operators cannot be applied infinitely and recursively which are defined using sequence of similar expressions

Recursion

Use case

M1	
movie	sequel
Rocky	Rocky II
Rocky II	Rocky III
Rocky III	Rocky IV

Recursion

Use case - Sequel of Sequel

Select M1C1.movie, M1C2. sequel FROM M1 as M1C1 JOIN M1 as M1C2 ON M1C1.sequel = M1C2.movie

M1C1		M1C2		M1C1.sequel = M1C2.movie
movie	sequel	movie	sequal	Not in M1
Rocky	Rocky II	Rocky	Rocky II	
Rocky	Rocky II	Rocky II	Rocky III	(Rocky, Rocky III)
Rocky	Rocky II	Rocky III	Rocky IV	

Recursion

Use case - Sequel of Sequel

Select M1C1.movie, M1C2. sequel FROM M1 as M1C1 JOIN M1 as M1C2 ON M1C1.sequel = M1C2.movie

M1C1		M1C2		M1C1.sequel = M1C2.movie
movie	sequel	movie	sequal	Not in M1
Rocky	Rocky II	Rocky	Rocky II	
Rocky	Rocky II	Rocky II	Rocky III	(Rocky, Rocky III)
Rocky	Rocky II	Rocky III	Rocky IV	
Rocky II	Rocky III	Rocky	Rocky II	
Rocky II	Rocky III	Rocky II	Rocky III	
Rocky II	Rocky III	Rocky III	Rocky IV	(Rocky II, Rocky IV)

Recursion

Use case - Sequel of Sequel

Select M1C1.movie, M1C2. sequel FROM M1 as M1C1 JOIN M1 as M1C2 ON M1C1.sequel = M1C2.movie

M1C1		M1C2		M1C1.sequel = M1C2.movie
movie	sequel	movie	sequal	Not in M1
Rocky	Rocky II	Rocky	Rocky II	
Rocky	Rocky II	Rocky II	Rocky III	(Rocky, Rocky II)
Rocky	Rocky II	Rocky III	Rocky IV	
Rocky II	Rocky III	Rocky	Rocky II	
Rocky II	Rocky III	Rocky II	Rocky III	
Rocky II	Rocky III	Rocky III	Rocky IV	(Rocky II, Rocky IV)
Rocky III	Rocky IV	Rocky	Rocky II	
Rocky III	Rocky IV	Rocky II	Rocky III	
Rocky III	Rocky IV	Rocky III	Rocky IV	

Recursion

Perform Join

obtain records which are not in $M1$

M1C1-M1C2

movie	sequel
Rocky	Rocky III
Rocky II	Rocky IV

Perform Union

Add the new records to $M1$ to make it $M2$

M2

movie	sequel
Rocky	Rocky II
Rocky II	Rocky III
Rocky III	Rocky IV
Rocky	Rocky III
Rocky II	Rocky IV

Recursion

Use case - Sequel of Sequel of Sequel

Perform (M21 ← M2 Join M1) followed by (M2 Union M21)

M2

movie	sequel
Rocky	Rocky II
Rocky II	Rocky III
Rocky III	Rocky IV
Rocky	Rocky III
Rocky II	Rocky IV

M1

movie	sequel
Rocky	Rocky II
Rocky II	Rocky III
Rocky III	Rocky IV

Use case - Sequel of Sequel of Sequel

M1-M2				
movie	sequel	movie	sequal	Not in M2
Rocky	Rocky II	Rocky	Rocky II	
Rocky	Rocky II	Rocky II	Rocky III	(Rocky, Rocky III)
Rocky	Rocky II	Rocky III	Rocky IV	
Rocky II	Rocky III	Rocky	Rocky II	
Rocky II	Rocky III	Rocky II	Rocky III	
Rocky II	Rocky III	Rocky III	Rocky IV	(Rocky II, Rocky IV)
Rocky III	Rocky IV	Rocky	Rocky II	
Rocky III	Rocky IV	Rocky II	Rocky III	
Rocky III	Rocky IV	Rocky III	Rocky IV	(Rocky, Rocky IV)
Rocky	Rocky III	Rocky	Rocky II	
Rocky	Rocky III	Rocky II	Rocky III	
Rocky	Rocky III	Rocky III	Rocky IV	
Rocky II	Rocky IV	Rocky	Rocky II	
Rocky II	Rocky IV	Rocky II	Rocky III	
Rocky II	Rocky IV	Rocky III	Rocky IV	

Recursion

Use case - Sequel of Sequel of Sequel

Perform (M21 ← M2 Join M1) followed by (M2 Union M21)

M2

movie	sequel
Rocky	Rocky II
Rocky II	Rocky III
Rocky III	Rocky IV
Rocky	Rocky III
Rocky II	Rocky IV

M1

movie	sequel
Rocky	Rocky II
Rocky II	Rocky III
Rocky III	Rocky IV

Use case - Sequel of Sequel of Sequel

M1-M2

movie	sequel	movie	sequal	Not in M2
Rocky	Rocky II	Rocky	Rocky II	
Rocky	Rocky II	Rocky II	Rocky III	(Rocky, Rocky III)
Rocky	Rocky II	Rocky III	Rocky IV	
Rocky	Rocky II	Rocky	Rocky III	
Rocky	Rocky II	Rocky II	Rocky IV	(Rocky, Rocky IV)

Recursion

Use case - Sequel of Sequel of Sequel

Perform (M21 ← M2 Join M1) followed by (M2 Union M21)

M2

movie	sequel
Rocky	Rocky II
Rocky II	Rocky III
Rocky III	Rocky IV
Rocky	Rocky III
Rocky II	Rocky IV

M1

movie	sequel
Rocky	Rocky II
Rocky II	Rocky III
Rocky III	Rocky IV

Use case - Sequel of Sequel of Sequel

M1-M2				
movie	sequel	movie	sequal	Not in M2
Rocky	Rocky II	Rocky	Rocky II	
Rocky	Rocky II	Rocky II	Rocky III	(Rocky, Rocky III)
Rocky	Rocky II	Rocky III	Rocky IV	
Rocky	Rocky II	Rocky	Rocky III	
Rocky	Rocky II	Rocky II	Rocky IV	(Rocky, Rocky IV)
Rocky II	Rocky III	Rocky	Rocky II	
Rocky II	Rocky III	Rocky II	Rocky III	
Rocky II	Rocky III	Rocky III	Rocky IV	(Rocky II, Rocky IV)
Rocky II	Rocky III	Rocky	Rocky III	
Rocky II	Rocky III	Rocky II	Rocky IV	

Recursion

Use case - Sequel of Sequel of Sequel

Perform (M21 ← M2 Join M1) followed by (M2 Union M21)

M2

movie	sequel
Rocky	Rocky II
Rocky II	Rocky III
Rocky III	Rocky IV
Rocky	Rocky III
Rocky II	Rocky IV

M1

movie	sequel
Rocky	Rocky II
Rocky II	Rocky III
Rocky III	Rocky IV

Use case - Sequel of Sequel of Sequel

M1-M2

movie	sequel	movie	sequal	Not in M2
Rocky	Rocky II	Rocky	Rocky II	
Rocky	Rocky II	Rocky II	Rocky III	(Rocky, Rocky III)
Rocky	Rocky II	Rocky III	Rocky IV	
Rocky	Rocky II	Rocky	Rocky III	
Rocky	Rocky II	Rocky II	Rocky IV	(Rocky, Rocky IV)
Rocky II	Rocky III	Rocky	Rocky II	
Rocky II	Rocky III	Rocky II	Rocky III	
Rocky II	Rocky III	Rocky III	Rocky IV	(Rocky II, Rocky IV)
Rocky II	Rocky III	Rocky	Rocky III	
Rocky II	Rocky III	Rocky II	Rocky IV	
Rocky III	Rocky IV	Rocky	Rocky II	
Rocky III	Rocky IV	Rocky II	Rocky III	
Rocky III	Rocky IV	Rocky III	Rocky IV	(Rocky III, Rocky IV
Rocky III	Rocky IV	Rocky	Rocky IV	
Rocky III	Rocky IV	Rocky II	Rocky IV	

Recursion: Sequel of Sequel of Sequel

Perform Join

obtain records which are not in $M2$

M1-M2

movie	sequel
Rocky	Rocky IV

Perform Union

Add the new records to $M2$ to make it $M3$

M3

movie	sequel
Rocky	Rocky II
Rocky II	Rocky III
Rocky III	Rocky III
Rocky	Rocky III
Rocky II	Rocky IV
Rocky	Rocky IV

Recursion: Sequel of Sequel of Sequel of Sequel?

Trilogy and more?

- Repeating above steps: Perform M3 Join M1
- Perform M3 Union (records (M3 Join M1) not in M3) to obtain M4
- These two steps yields no new records. The recursion terminates

Datalog

Formulation

- FollowOn(x, y) ← M1(x, y)
- FollowOn(x, y) ← M1(x, z) AND FollowOn(z, y)

Airlines Database

Graph & Table

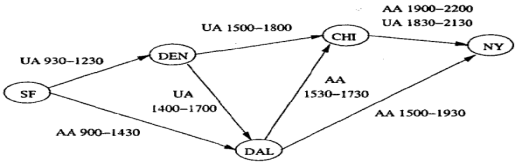


Figure 10.5: A map of some airline flights

airline	*from*	*to*	*departs*	*arrives*
UA	SF	DEN	930	1230
AA	SF	DAL	900	1430
UA	DEN	CHI	1500	1800
UA	DEN	DAL	1400	1700
AA	DAL	CHI	1530	1730
AA	DAL	NY	1500	1930
AA	CHI	NY	1900	2200
UA	CHI	NY	1830	2130

Datalog: Recursive Programming

Recursive Rules

1. `Reaches(x, y) ← Flights(a, x, y, d, r)`
2. `Reaches(x, y) ← Reaches(x, z) AND Reaches(z, y)`

Flights				
airline	from	to	departs	arrives
UA	SF	DEN	0930	1230
AA	SF	DAL	0900	1430
UA	DEN	CHI	1500	1800
UA	DEN	DAL	1400	1700
AA	DAL	CHI	1530	1730
AA	DAL	NY	1500	1930
AA	CHI	NY	1900	2200
UA	CHI	NY	1830	2130

Datalog: Recursive Programming

Recursive Rules

1. `Reaches(x, y) ← Flights(a, x, y, d, r)`
2. `Reaches(x, y) ← Reaches(x, z) AND Reaches(z, y)`

Flights				
airline	from	to	departs	arrives
UA	SF	DEN	0930	1230
AA	SF	DAL	0900	1430
UA	DEN	CHI	1500	1800
UA	DEN	DAL	1400	1700
AA	DAL	CHI	1530	1730
AA	DAL	NY	1500	1930
AA	CHI	NY	1900	2200
UA	CHI	NY	1830	2130

Round # 1

Reaches	
x	y
SF	DEN
SF	DAL
DEN	CHI
DEN	DAL
DAL	CHI
DAL	NY
CHI	NY

Datalog: Recursive Programming

Recursive Rules

1. `Reaches(x, y) ← Reaches(x, z) AND Reaches(z, y)`

Reaches R1		Reaches R2	
R1.x	R1.z	R2.z	R2.y
SF	DEN	SF	DEN
SF	DEN	SF	DAL
SF	DEN	DEN	CHI
SF	DEN	DEN	DAL
SF	DEN	DAL	CHI
SF	DEN	DAL	NY
SF	DEN	CHI	NY

Round # 1

Reaches	
x	y
SF	DEN
SF	DAL
DEN	CHI
DEN	DAL
DAL	CHI
DAL	NY
CHI	NY

Datalog: Recursive Programming

Result

1. `Reaches(x, y) ← Reaches(x, z) AND Reaches(z, y)`

Reaches R1		Reaches R2	
R1.x	R1.z	R2.z	R2.y
SF	DEN	SF	DEN
SF	DEN	SF	DAL
SF	DEN	DEN	CHI
SF	DEN	DEN	DAL
SF	DEN	DAL	CHI
SF	DEN	DAL	NY
SF	DEN	CHI	NY

Round # 1	
Reaches	
x	y
SF	DEN
SF	DAL
DEN	CHI
DEN	DAL
DAL	CHI
DAL	NY
CHI	NY
Round #2	
SF	CHI

Datalog: Recursive Programming

Recursive Rules

1. `Reaches(x, y) ← Reaches(x, z) AND Reaches(z, y)`

Reaches R1		Reaches R2	
R1.x	R1.z	R2.z	R2.y
SF	DAL	SF	DEN
SF	DAL	SF	DAL
SF	DAL	DEN	CHI
SF	DAL	DEN	DAL
SF	DAL	DAL	CHI
SF	DAL	DAL	NY
SF	DAL	CHI	NY

Round # 1	
Reaches	
x	y
SF	DEN
SF	DAL
DEN	CHI
DEN	DAL
DAL	CHI
DAL	NY
CHI	NY
Round #2	
SF	CHI
SF	NY

Datalog: Recursive Programming

Recursive Result

1. `Reaches(x, y) ← Reaches(x, z) AND Reaches(z, y)`

Reaches R1		Reaches R2	
R1.x	R1.z	R2.z	R2.y
DEN	CHI	SF	DEN
DEN	CHI	SF	DAL
DEN	CHI	DEN	CHI
DEN	CHI	DEN	DAL
DEN	CHI	DAL	CHI
DEN	CHI	DAL	NY
DEN	CHI	CHI	NY

Round # 1	
Reaches	
x	y
SF	DEN
SF	DAL
DEN	CHI
DEN	DAL
DAL	CHI
DAL	NY
CHI	NY
Round #2	
SF	CHI
SF	NY
DEN	NY

Datalog: Recursive Programming

Recursive Result

1. `Reaches(x, y) ← Reaches(x, z) AND Reaches(z, y)`

Reaches R1		Reaches R2	
R1.x	R1.z	R2.z	R2.y
DEN	DAL	SF	DEN
DEN	DAL	SF	DAL
DEN	DAL	DEN	CHI
DEN	DAL	DEN	DAL
DEN	DAL	DAL	CHI
DEN	DAL	DAL	NY
DEN	DAL	CHI	NY

Round # 1	
Reaches	
x	y
SF	DEN
SF	DAL
DEN	CHI
DEN	DAL
DAL	CHI
DAL	NY
CHI	NY
Round #2	
SF	CHI
SF	NY
DEN	NY
DEN	CHI

Datalog: Recursive Programming

Recursive Result

1. `Reaches(x, y) ← Reaches(x, z) AND Reaches(z, y)`

Reaches R1		Reaches R2	
R1.x	R1.z	R2.z	R2.y
DEN	DAL	SF	DEN
DEN	DAL	SF	DAL
DEN	DAL	DEN	CHI
DEN	DAL	DEN	DAL
DEN	DAL	DAL	CHI
DEN	DAL	DAL	NY
DEN	DAL	CHI	NY

Round # 1	
Reaches	
x	y
SF	DEN
SF	DAL
DEN	CHI
DEN	DAL
DAL	CHI
DAL	NY
CHI	NY
Round #2	
SF	CHI
SF	NY
DEN	NY

Datalog: Recursive Programming

Recursive Result

1. `Reaches(x, y) ← Reaches(x, z) AND Reaches(z, y)`

Reaches R1		Reaches R2	
R1.x	R1.z	R2.z	R2.y
DAL	CHI	SF	DEN
DAL	CHI	SF	DAL
DAL	CHI	DEN	CHI
DAL	CHI	DEN	DAL
DAL	CHI	DAL	CHI
DAL	CHI	DAL	NY
DAL	CHI	CHI	NY

Round # 1	
Reaches	
x	y
SF	DEN
SF	DAL
DEN	CHI
DEN	DAL
DAL	CHI
DAL	NY
CHI	NY
Round #2	
SF	CHI
SF	NY
DEN	NY

Datalog: Recursive Programming

Recursive Result

1. `Reaches(x, y) ← Reaches(x, z) AND Reaches(z, y)`

Reaches R1		Reaches R2	
R1.x	R1.z	R2.z	R2.y
DAL	NY	SF	DEN
DAL	NY	SF	DAL
DAL	NY	DEN	CHI
DAL	NY	DEN	DAL
DAL	NY	DAL	CHI
DAL	NY	DAL	NY
DAL	NY	CHI	NY

Round # 1	
Reaches	
x	y
SF	DEN
SF	DAL
DEN	CHI
DEN	DAL
DAL	CHI
DAL	NY
CHI	NY
Round #2	
SF	CHI
SF	NY
DEN	NY

Datalog: Recursive Programming

Recursive Result

1. `Reaches(x, y) ← Reaches(x, z) AND Reaches(z, y)`

Reaches R1		Reaches R2	
R1.x	R1.z	R2.z	R2.y
CHI	NY	SF	DEN
CHI	NY	SF	DAL
CHI	NY	DEN	CHI
CHI	NY	DEN	DAL
CHI	NY	DAL	CHI
CHI	NY	DAL	NY
CHI	NY	CHI	NY

Round # 1	
Reaches	
x	y
SF	DEN
SF	DAL
DEN	CHI
DEN	DAL
DAL	CHI
DAL	NY
CHI	NY
Round #2	
SF	CHI
SF	NY
DEN	NY

Datalog: Recursive Programming

Recursive Result

1. `Reaches(x, y) ← Reaches(x, z) AND Reaches(z, y)`

Reaches R1		Reaches R2	
R1.x	R1.z	R2.z	R2.y
SF	DEN	SF	DEN
SF	DEN	SF	DAL
SF	DEN	DEN	CHI
SF	DEN	DEN	DAL
SF	DEN	DAL	CHI
SF	DEN	DAL	NY
SF	DEN	CHI	NY
SF	DEN	DAL	CHI
SF	DEN	DAL	NY
SF	DEN	CHI	NY

Round # 2	
Reaches	
x	y
SF	DEN
SF	DAL
DEN	CHI
DEN	DAL
DAL	CHI
DAL	NY
CHI	NY
SF	CHI
SF	NY
DEN	NY

Datalog: Recursive Programming

Recursive Result

1. `Reaches(x, y) ← Reaches(x, z) AND Reaches(z, y)`

Reaches R1		Reaches R2	
R1.x	R1.z	R2.z	R2.y
SF	DAL	SF	DEN
SF	DAL	SF	DAL
SF	DAL	DEN	CHI
SF	DAL	DEN	DAL
SF	DAL	DAL	CHI
SF	DAL	DAL	NY
SF	DAL	CHI	NY
SF	DAL	DAL	CHI
SF	DAL	DAL	NY
SF	DAL	CHI	NY

Round # 2

Reaches	
x	y
SF	DEN
SF	DAL
DEN	CHI
DEN	DAL
DAL	CHI
DAL	NY
CHI	NY
SF	CHI
SF	NY
DEN	NY

Datalog: Recursive Programming

Recursive Result

1. `Reaches(x, y) ← Reaches(x, z) AND Reaches(z, y)`

Reaches R1		Reaches R2	
R1.x	R1.z	R2.z	R2.y
DEN	CHI	SF	DEN
DEN	CHI	SF	DAL
DEN	CHI	DEN	CHI
DEN	CHI	DEN	DAL
DEN	CHI	DAL	CHI
DEN	CHI	DAL	NY
DEN	CHI	CHI	NY
DEN	CHI	DAL	CHI
DEN	CHI	DAL	NY
DEN	CHI	CHI	NY

Round # 2

Reaches	
x	y
SF	DEN
SF	DAL
DEN	CHI
DEN	DAL
DAL	CHI
DAL	NY
CHI	NY
SF	CHI
SF	NY
DEN	NY

Datalog: Recursive Programming

Recursive Result

1. `Reaches(x, y) ← Reaches(x, z) AND Reaches(z, y)`

Reaches R1		Reaches R2	
R1.x	R1.z	R2.z	R2.y
DEN	DAL	SF	DEN
DEN	DAL	SF	DAL
DEN	DAL	DEN	CHI
DEN	DAL	DEN	DAL
DEN	DAL	DAL	CHI
DEN	DAL	DAL	NY
DEN	DAL	CHI	NY
DEN	DAL	DAL	CHI
DEN	DAL	DAL	NY
DEN	DAL	CHI	NY

Round # 2

Reaches	
x	y
SF	DEN
SF	DAL
DEN	CHI
DEN	DAL
DAL	CHI
DAL	NY
CHI	NY
SF	CHI
SF	NY
DEN	NY

Datalog: Recursive Programming

Recursive Result

1. `Reaches(x, y) ← Reaches(x, z) AND Reaches(z, y)`

Reaches R1		Reaches R2	
R1.x	R1.z	R2.z	R2.y
DAL	CHI	SF	DEN
DAL	CHI	SF	DAL
DAL	CHI	DEN	CHI
DAL	CHI	DEN	DAL
DAL	CHI	DAL	CHI
DAL	CHI	DAL	NY
DAL	CHI	CHI	NY
DAL	CHI	DAL	CHI
DAL	CHI	DAL	NY
DAL	CHI	CHI	NY

Round # 2

Reaches	
x	y
SF	DEN
SF	DAL
DEN	CHI
DEN	DAL
DAL	CHI
DAL	NY
CHI	NY
SF	CHI
SF	NY
DEN	NY

Datalog: Recursive Programming

Recursive Result

1. `Reaches(x, y) ← Reaches(x, z) AND Reaches(z, y)`

Reaches R1		Reaches R2	
R1.x	R1.z	R2.z	R2.y
DAL	NY	SF	DEN
DAL	NY	SF	DAL
DAL	NY	DEN	CHI
DAL	NY	DEN	DAL
DAL	NY	DAL	CHI
DAL	NY	DAL	NY
DAL	NY	CHI	NY
DAL	NY	DAL	CHI
DAL	NY	DAL	NY
DAL	NY	CHI	NY

Round # 2

Reaches	
x	y
SF	DEN
SF	DAL
DEN	CHI
DEN	DAL
DAL	CHI
DAL	NY
CHI	NY
SF	CHI
SF	NY
DEN	NY

Datalog: Recursive Programming

Recursive Result

1. `Reaches(x, y) ← Reaches(x, z) AND Reaches(z, y)`

Reaches R1		Reaches R2	
R1.x	R1.z	R2.z	R2.y
CHI	NY	SF	DEN
CHI	NY	SF	DAL
CHI	NY	DEN	CHI
CHI	NY	DEN	DAL
CHI	NY	DAL	CHI
CHI	NY	DAL	NY
CHI	NY	CHI	NY
CHI	NY	DAL	CHI
CHI	NY	DAL	NY
CHI	NY	CHI	NY

Round # 2

Reaches	
x	y
SF	DEN
SF	DAL
DEN	CHI
DEN	DAL
DAL	CHI
DAL	NY
CHI	NY
SF	CHI
SF	NY
DEN	NY

Datalog: Recursive Programming

Recursive Result

1. `Reaches(x, y) ← Reaches(x, z) AND Reaches(z, y)`

Reaches R1		Reaches R2	
R1.x	R1.z	R2.z	R2.y
SF	CHI	SF	DEN
SF	CHI	SF	DAL
SF	CHI	DEN	CHI
SF	CHI	DEN	DAL
SF	CHI	DAL	CHI
SF	CHI	DAL	NY
SF	CHI	CHI	NY
SF	CHI	DAL	CHI
SF	CHI	DAL	NY
SF	CHI	CHI	NY

Round # 2

Reaches	
x	y
SF	DEN
SF	DAL
DEN	CHI
DEN	DAL
DAL	CHI
DAL	NY
CHI	NY
SF	CHI
SF	NY
DEN	NY

Datalog: Recursive Programming

Recursive Result

1. `Reaches(x, y) ← Reaches(x, z) AND Reaches(z, y)`

Reaches R1		Reaches R2	
R1.x	R1.z	R2.z	R2.y
SF	NY	SF	DEN
SF	NY	SF	DAL
SF	NY	DEN	CHI
SF	NY	DEN	DAL
SF	NY	DAL	CHI
SF	NY	DAL	NY
SF	NY	CHI	NY
SF	NY	DAL	CHI
SF	NY	DAL	NY
SF	NY	CHI	NY

Round # 2

Reaches	
x	y
SF	DEN
SF	DAL
DEN	CHI
DEN	DAL
DAL	CHI
DAL	NY
CHI	NY
SF	CHI
SF	NY
DEN	NY

Datalog: Recursive Programming

Recursive Result

1. `Reaches(x, y) ← Reaches(x, z) AND Reaches(z, y)`

Reaches R1		Reaches R2	
R1.x	R1.z	R2.z	R2.y
DEN	NY	SF	DEN
DEN	NY	SF	DAL
DEN	NY	DEN	CHI
DEN	NY	DEN	DAL
DEN	NY	DAL	CHI
DEN	NY	DAL	NY
DEN	NY	CHI	NY
DEN	NY	DAL	CHI
DEN	NY	DAL	NY
DEN	NY	CHI	NY

Round # 2

Reaches	
x	y
SF	DEN
SF	DAL
DEN	CHI
DEN	DAL
DAL	CHI
DAL	NY
CHI	NY
SF	CHI
SF	NY
DEN	NY

Datalog: Recursive Programming

Recursive Rules

1. Difference between Departure time at next hop and arrival time should be at least 100 minutes
2. `Connects(x, y, d, r)` ← `Flights(a, x, y, d, r)`
3. `Connects(x, y, d, r)` ← `Connects(a, x, z, d, t1)` AND `Connects(a, z, y, t2, r)` AND `t1` ≤ `t2 - 100`

Flights				
airline	from	to	departs	arrives
UA	SF	DEN	0930	1230
AA	SF	DAL	0900	1430
UA	DEN	CHI	1500	1800
UA	DEN	DAL	1400	1700
AA	DAL	CHI	1530	1730
AA	DAL	NY	1500	1930
AA	CHI	NY	1900	2200
UA	CHI	NY	1830	2130

Connects			
x	y	d	r
Round #1			
SF	DEN	0930	1230
SF	DAL	0900	1430
DEN	CHI	1500	1800
DEN	DAL	1400	1700
DAL	CHI	1530	1730
DAL	NY	1500	1930
CHI	NY	1900	2200
Round #2			
SF	CHI	0900	1730
SF	CHI	0930	1800
SF	DAL	0930	1700
DEN	NY	1500	2200
DAL	NY	1530	2130
DAL	NY	1530	2200
Round #3			
SF	NY	0900	2130

Datalog: Recursive Programming

Recursive Rules

1. Find those pairs of cities (x, y) in which only UA operates but not AA
2. `UAReaches(x, y) ← Flights(a, x, y, d, r)`
3. `UAReaches(x, y) ← UAReaches(x, z) AND UAReaches(z, y)`
4. `AAReaches(x, y) ← Flights(a, x, y, d, r)`
5. `AAReaches(x, y) ← AAReaches(x, z) AND AAReaches(z, y)`
6. `UAOnly(x, y) ← UAReaches(x, y) AND NOT AAReaches(x, y)`

Flights				
airline	from	to	departs	arrives
UA	SF	DEN	0930	1230
AA	SF	DAL	0900	1430
UA	DEN	CHI	1500	1800
UA	DEN	DAL	1400	1700
AA	DAL	CHI	1530	1730
AA	DAL	NY	1500	1930
AA	CHI	NY	1900	2200
UA	CHI	NY	1830	2130

UAReaches	
x	y
SF	DEN
SF	DAL
SF	CHI
SF	NY
DEN	DAL
DEN	CHI
DEN	NY
CHI	NY

AAReaches	
SF	DAL
SF	CHI
SF	NY
DAL	CHI

UAOnly	
x	y
SF	DEN
DEN	DAL
DEN	CHI
DEN	NY

SQL: Recursive Programming

Introduction

- SQL has grown to be an expressive data-oriented language
- Intentionally it has not been designed as a general-purpose programming language
- SQL does not loop forever.
- Any SQL query is expected to terminate, regardless of size/contents of the input tables
- SQL queries are evaluated efficiently

SQL: Addition of recursion

Expressive

SQL becomes a Turing-complete language thus a general-purpose programming lanugage

Efficiency

No longer queries are guaranteed to terminate.

SQL: WITH RECURSIVE

Recursive common table expression (CTE)

```
WITH RECURSIVE T(c1, c2, ..., ck)  -- common schema of q0 and q.(.)
AS(
   q0      -- base case query, evaluated once

   UNION [ALL]

   q0(T) -- recursive query refers to T itself
)          -- evaluated repeatedly

q(T)       -- final post processing query
```

SQL: WITH RECURSIVE

Initial query

Forms the base result set of the CTE structure. The initial query part is referred to as an anchor member.

Recursive query

References to the CTE name, therefore, it is called a recursive member. The recursive member is joined with the anchor member by UNION or UNION ALL

Termination

A termination condition that ensures the recursion stops when the recursive member returns no row

SQL: WITH RECURSIVE

Step 1

Separate the members into anchor and recursive members

Step 2

Execute the anchor member to form the base result set R0. Use this base result set for next iteration

Step 3

Execute the recursive member with Ri result set as input and make Ri+1 as an output

Step 4

Repeat step 3 till recursive member returns an empty result set

Step 5

Combine result sets from R0 to Rn using UNION [ALL] operator

Example - 01

Recursive Rules

1. `Reaches(x, y) ← Flights(a, x, y, d, r)`
2. `Reaches(x, y) ← Reaches(x, z) AND Reaches(z, y)`

Recursive SQL Query

```
WITH RECURSIVE Reaches(frm, to)  -- T(c1, c2)
AS(
    (SELECT frm, to FROM Flights) -- q0

      UNION

    ( SELECT R1.frm, R2.to
      FROM   Reaches AS R1, Reaches AS R2
      WHERE  R1.to = R2.frm
    )
)
    SELECT * FROM Reaches;
```

Embedded SQL - 01

Introduction

- Customize input handling - with MySQL works with SQL statements. However, application programs customize input methods for the users.
- Validate input provided by the user - input sanity check is important which result in a cleaner SQL statements.
- Generate inputs automatically - some application might not even involve human users example when an input to MySQL is generated from a program
- Customize output - MySQL output is unformatted. Depending on the application and the context, output customization is necessary for various use cases

Embedded SQL - 02

Introduction

- Work around constraints imposed by the nature of SQL itself - SQL scripts consist of a set of statements executed one at a time from beginning to end with minimal error checking.
  In general the requirement for tasks that involve master-detail relationships and have complex output-formatting requirements.
- Integrate MySQL into any application - client programming interface gives the means to benefit the capability of a database to provide information.

Embedded SQL - 03

Introduction

- Use a web server to provide enhanced access to MySQL
- Use MySQL to enhance the capabilities of web server

Embedded SQL - 04

Available APIs for MySQL

- MySQL server has low-level native client-server protocol that defines how client programs establish connections and how to communicate using the established connections
- Clients can use this protocol at various levels of abstraction

Embedded SQL - 05

Levels of Abstraction

- MySQL provides a client library written in the C programming language
- This enables to access MySQL databases from withint any C program
- MySQL interfaces for other languages can link the C client library into the language processor.
- The client library thus provides the means whereby MySQL bindings for other languages can be built on top of the C API.
- This type of interfaces exists for Perl, PHP, Python, Ruby, C++, Tcl and others

Embedded SQL - 06

APIs

- The C client library API is the primary programming interface to MySQL. It is used to implement the standard clients in the MySQL distribution such as `mysql`, `mysqladmin` and `mysqldump`
- The PHP API is a server-side scripting language that provides convenient way of embedding programs in `Web pages`.

Embedded SQL - PHP - 01

Required package for ubuntu

To use MySQL with PHP, install php-mysql package in ubuntu as `sudo apt-get install php-mysql`

Embedded SQL - PHP - 02

General structure

- Establish a database connection
- Embedded SQL statements
- Fire SQL queries (to database through established connection)
- Close connection

Primary key vs temporal key

Example Schema

- `eid` and `pcn` stand for primary key
- Only in the absence of timed attributes
- `start_date` and `end_date` are included in the relation
- No employee can have a particular position twice at the same time.
- `eid, pcn, start_date, end_date` not a primary key

eid	pcn	start_date	end_date
123	900225	01-Jan-1996	01-June-1996
123	900225	01-Apr-1996	01-Oct-1996

Primary key vs temporal key

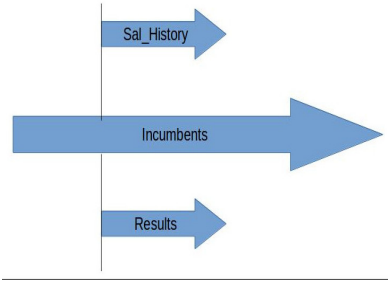
```
CREATE TABLE Incumbents( eid INT, pcn INT, start_date date,
        end_date date,
        CHECK(
                NOT EXISTS (
                        SELECT  *
                        FROM    Incumbents as I1
                        WHERE   1 <
                        (SELECT COUNT(eid)
                         FROM   Incumbents as I2
                         WHERE I1.eid = I2.eid
                         AND    I1.pcn = I2.pcn
                         AND    I1.start_date < I2.end_date
                         AND    I2.start_date < I1.end_date)
                        )
                AND NOT EXISTS (
                        SELECT *
                        FROM    Incumbents AS I1
                        WHERE   I1.eid is null OR I1.pcn is null
                        )
        )
```

Example queries involving time

Query involving time

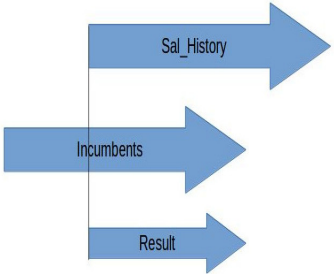
Provide the salary and position history of all employees
Salary history is in Sal_History table.
Position is in Incumbents table.

Join Query - Case_1



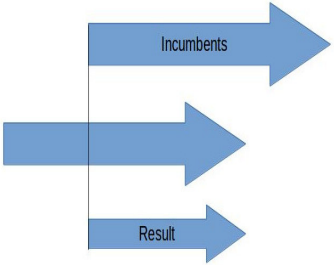

```
SELECT  eid, amount, pcn, SH.start_date, SH.end_date
FROM    Sal_History AS SH
JOIN    Incumbents AS I
ON      SH.eid = I.eid
WHERE   I.start_date  <= SH.start_date
AND     SH.end_date <= I.end_date
```

Join Query - Case_2



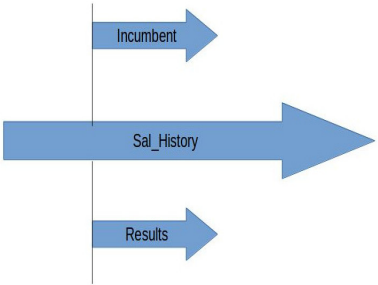

```
SELECT   eid, amount, pcn, SH.start_date, I.end_date
FROM     Sal_History AS SH
JOIN     Incumbents AS I
ON       SH.eid = I.eid
WHERE    SH.start_date  >= I.start_date
AND      I.end_date <= SH.end_date
AND      SH.start_date <= I.end_date
```

Join Query - Case_3




```
SELECT   eid, amount, pcn, I.start_date, SH.end_date
FROM     Sal_History AS SH
JOIN     Incumbents AS I
ON       SH.eid = I.eid
WHERE    I.start_date  > SH.start_date
AND      SH.end_date <= I.end_date
```

Join Query - Case_4




```
SELECT   eid, amount, pcn, I.start_date, I.end_date
FROM     Sal_History AS SH
JOIN     Incumbents AS I
ON       SH.eid = I.eid
WHERE    I.start_date  > SH.start_date
AND      I.end_date <= SH.end_date
```

Full query

Provide the salary and position history of all employees

```
SELECT  eid, amount, pcn, SH.start_date, SH.end_date
FROM    Sal_History AS SH
JOIN    Incumbents AS I
ON      SH.eid = I.eid
WHERE   I.start_date  <= SH.start_date
AND     SH.end_date <= I.end_date
    UNION
SELECT  eid, amount, pcn, SH.start_date, I.end_date
FROM    Sal_History AS SH
JOIN    Incumbents AS I
ON      SH.eid = I.eid
WHERE   SH.start_date  >= I.start_date
AND     I.end_date <= SH.end_date
AND     SH.start_date <= I.end_date
    UNION
SELECT  eid, amount, pcn, I.start_date, SH.end_date
FROM    Sal_History AS SH
JOIN    Incumbents AS I
ON      SH.eid = I.eid
WHERE   I.start_date  > SH.start_date
AND     SH.end_date <= I.end_date
AND     I.start_date < SH.end_date
    UNION
SELECT  eid, amount, pcn, I.start_date, I.end_date
FROM    Sal_History AS SH
JOIN    Incumbents AS I
ON      SH.eid = I.eid
WHERE   I.start_date  > SH.start_date
```