CS245: Databases

SQL

Vijaya saradhi

Department of Computer Science and Engineering
Indian Institute of Technology Guwahati

Active Databases

Constructs

- Triggers: a series of actions associated with INSERT, UPDATE or DELETE queries and performed whenever these queries are involved
- Assertions: a boolean valued SQL expression that must be true at all times
- Events: Time based actions as opposed to query based

Active Databases - Triggers

Triggers

- Triggers also known as event-condition-action rules or ECA rules
- Triggers are involved only when certain conditions specified by the database programmer occur
- Trigger tests a specified condition. If the condition does not hold then nothing else associated with the trigger happens
- If the condition is satisfied then associated action is performed

Active Databases - Triggers

Triggers

- Has all the power of assertions
- Easier to implement
- Programmer specifies when they should be invoked
- Every trigger must be associated with a table
- Triggers are invoked automatically
- Triggers cannot be called directly
- Are part of transactions and can ROLLBACK transactions

Active Databases - Triggers

Triggers

- Cascade changes through related tables in database
- Enforce complex data integrity than a CHECK constraint
- Define custom error messages
- Compare before and after states of data under modification
- Triggers can be
  - Created
  - Altered
  - Dropped

Active Databases - Triggers

Triggers

- The action may be executed either before or after the triggering event
- Action can refer to old and new values of tuples that were inserted, deleted or updated
- Condition may be specified using WHEN clause
- Programmer has an option of specifying that the action is performed either:
  - Once for each modified tuple OR
  - Once for all the tuples that are changed in the database operation

Active Databases - Triggers

Triggers

- Invoke certain operations upon specified action on a table
- Action could be: insert a tuple into a table
- Action could be: delete a row from a table
- Action could be: update a row from in a table
- Performed operation can be on the table itself
- Performed Operation can be on other tables and/or databases

Trigger - Example - 01

Totaling amount

account (acct_num INT, amount FLOAT)

Sum Keep track of how much amount is deposited (irrespective of account number)

Insert The above operation should be performed for deposites only (not withdraw)

Before Sum opertaion should be performed even before the tuple (acct_num, amount) is inserted into the `account` table

Trigger - Example - 01

```
CREATE TABLE account(acct_num INT, amount FLOAT);

-- Create a global variable @sum
SET @sum = 0;

CREATE TRIGGER insert_sum
BEFORE INSERT
ON account
FOR EACH ROW
    SET @sum = @sum + NEW.amount;
```

Trigger - Example - 01

Trigger Action

```
CREATE TRIGGER insert_sum
BEFORE INSERT
ON account
FOR EACH ROW
    SET @sum = @sum + NEW.amount;
```

Trigger Events

```
INSERT INTO account VALUES (137, 14.98);
INSERT INTO account VALUES (141,1937.50);
INSERT INTO account VALUES (97,-100.00);
SELECT @sum AS "Total␣amount␣inserted';
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
```

Example

Explanation

- CREATE TRIGGER will create a trigger with the name insert_sum
- The trigger will not get executed immediately
- Condition for invoking trigger is: When a INSERT operation is performed on table account
- Statements in trigger gets executed even before the row is written into the account table

Names and meanings

INSERT INTO Sailor (sid, sname, rating, age)
VALUES (99, 'Sailor 99', 9, 37);

NEW.sid	NEW.sname	NEW.rating	NEW.age
99	Sailor 99	9	37

Sailors

OLD.sid	OLD.sname	OLD.rating	OLD.age
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5
99	Sailor 99	9	37

Names and meanings

- Before the statement `INSERT INTO account VALUES (137, 14.98);` there are no rows in the table
- Attributes/colums in a new row to be inserted are referred with NEW
- NEW.acct_num refers to 137
- NEW.amount refers to 14.98
- Rows that are already present in the `account` table are referred with OLD
- The statement SET @sum = @sum + NEW.amount; gets executed before row is inserted into account table

Trigger - Example - 02

Assumption

- Assume existance of table: `account(acc_num, amount)`
- updated amount must alway be between 0 and 100
- If the updated amount is more than 100, clamp to 100
- It the updated amount is less than 100, clamp to 0

```
DELIMITER //
CREATE TRIGGER update_check
BEFORE UPDATE ON account
FOR EACH ROW
BEGIN
  IF NEW.amount < 0 THEN
    SET NEW.amount = 0;
  ELSEIF NEW.amount > 100 THEN
    SET NEW.amount = 100;
  END IF
END; //
DELIMITER ;
```

Trigger - Example - 03

```
CREATE TABLE test1(a1 INT);
CREATE TABLE test2(a2 INT);
CREATE TABLE test3(a3 INT NOT NULL PRIMARY KEY(a3));
CREATE TABLE test4(a4 INT NOT NULL PRIMARY KEY(a4), b4 INT DEFAULT 0);
```

Trigger - Example - 03

```
DELIMITER |

CREATE TRIGGER testref BEFORE INSERT ON test1
  FOR EACH ROW
  BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = NEW.a1;
  END;
|

DELIMITER ;
```

Trigger - Example - 03

```
INSERT INTO test3 values (1), (2), (3), (4), (5), (6), (7), (8), (9), (10);
INSERT INTO test4 values (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0), (7,
                0), (8, 0), (9, 0), (10, 0);
```

Database state

test1	test2	test3	test4	
a1	a2	a3	a4	b4
		1	1	0
		2	2	0
		3	3	0
		4	4	0
		5	5	0
		6	6	0
		7	7	0
		8	8	0
		9	9	0
		10	10	0

Trigger - Example - 03 (a)

```
INSERT INTO test1 VALUES (1);
```

```
DELIMITER |

CREATE TRIGGER testref BEFORE INSERT ON test1
  FOR EACH ROW
  BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = N
  END;
|

DELIMITER ;
```

Database state

Trigger - Example - 03 (a)

```
INSERT INTO test1 VALUES (1);
```

```
DELIMITER |

CREATE TRIGGER testref BEFORE INSERT ON test1
  FOR EACH ROW
  BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = N
  END;
|

DELIMITER ;
```

Database state

test1	test2	test3	test4	
a1	a2	a3	a4	b4
1	1	~~1~~	1	1
		2	2	0
		3	3	0
		4	4	0
		5	5	0
		6	6	0
		7	7	0
		8	8	0
		9	9	0
		10	10	0

Trigger - Example - 03 (a)

```
INSERT INTO test1 VALUES (3);
```

```
DELIMITER |

CREATE TRIGGER testref BEFORE INSERT ON test1
  FOR EACH ROW
  BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = N
  END;
|

DELIMITER ;
```

Database state

Trigger - Example - 03 (a)

```
INSERT INTO test1 VALUES (3);
```

```
DELIMITER |

CREATE TRIGGER testref BEFORE INSERT ON test1
  FOR EACH ROW
  BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = N
  END;
|

DELIMITER ;
```

Database state

test1	test2	test3	test4	
a1	a2	a3	a4	b4
1	1	~~1~~	1	1
3	3	2	2	0
		~~3~~	3	1
		4	4	0
		5	5	0
		6	6	0
		7	7	0
		8	8	0
		9	9	0
		10	10	0

Trigger - Example - 03 (a)

```
INSERT INTO test1 VALUES (1);
```

```
DELIMITER |

CREATE TRIGGER testref BEFORE INSERT ON test1
  FOR EACH ROW
  BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = N
  END;
|

DELIMITER ;
```

Database state

Trigger - Example - 03 (a)

```
INSERT INTO test1 VALUES (1);
```

```
DELIMITER |

CREATE TRIGGER testref BEFORE INSERT ON test1
  FOR EACH ROW
  BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = N
  END;
|

DELIMITER ;
```

Database state

test1	test2	test3	test4	
a1	a2	a3	a4	b4
1	1	~~1~~	1	2
3	3	2	2	0
1	1	~~3~~	3	1
		4	4	0
		5	5	0
		6	6	0
		7	7	0
		8	8	0
		9	9	0
		10	10	0

Trigger - Example - 03 (a)

```
INSERT INTO test1 VALUES (7);
```

```
DELIMITER |

CREATE TRIGGER testref BEFORE INSERT ON test1
  FOR EACH ROW
  BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = N
  END;
|

DELIMITER ;
```

Database state

Trigger - Example - 03 (a)

```
INSERT INTO test1 VALUES (7);
```

```
DELIMITER |

CREATE TRIGGER testref BEFORE INSERT ON test1
  FOR EACH ROW
  BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = N
  END;
|

DELIMITER ;
```

Database state

test1	test2	test3	test4	
a1	a2	a3	a4	b4
1	1	~~1~~	1	2
3	3	2	2	0
1	1	~~3~~	3	1
7	7	4	4	0
		5	5	0
		6	6	0
		~~7~~	7	1
		8	8	0
		9	9	0
		10	10	0

Trigger - Example - 03 (a)

```
INSERT INTO test1 VALUES (1);
```

```
DELIMITER |

CREATE TRIGGER testref BEFORE INSERT ON test1
  FOR EACH ROW
  BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = N
  END;
|

DELIMITER ;
```

Database state

Trigger - Example - 03 (a)

```
INSERT INTO test1 VALUES (1);
```

```
DELIMITER |

CREATE TRIGGER testref BEFORE INSERT ON test1
  FOR EACH ROW
  BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = N
  END;
|

DELIMITER ;
```

Database state

test1	test2	test3	test4	
a1	a2	a3	a4	b4
1	1	~~1~~	1	3
3	3	2	2	0
1	1	~~3~~	3	1
7	7	4	4	0
1	1	5	5	0
		6	6	0
		7	7	1
		8	8	0
		9	9	0
		10	10	0

Trigger - Example - 03 (a)

```
INSERT INTO test1 VALUES (8);
```

```
DELIMITER |

CREATE TRIGGER testref BEFORE INSERT ON test1
  FOR EACH ROW
  BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = N
  END;
|

DELIMITER ;
```

Database state

Trigger - Example - 03 (a)

```
INSERT INTO test1 VALUES (8);
```

```
DELIMITER |

CREATE TRIGGER testref BEFORE INSERT ON test1
  FOR EACH ROW
  BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = N
  END;
|

DELIMITER ;
```

Database state

test1	test2	test3	test4	
a1	a2	a3	a4	b4
1	1	~~1~~	1	3
3	3	2	2	0
1	1	~~3~~	3	1
7	7	4	4	0
1	1	5	5	0
8	8	6	6	0
		7	7	1
		~~8~~	8	1
		9	9	0
		10	10	0

Trigger - Example - 03 (a)

```
INSERT INTO test1 VALUES (4);
```

```
DELIMITER |

CREATE TRIGGER testref BEFORE INSERT ON test1
  FOR EACH ROW
  BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = N
  END;
|

DELIMITER ;
```

Database state

Trigger - Example - 03 (a)

```
INSERT INTO test1 VALUES (4);
```

```
DELIMITER |

CREATE TRIGGER testref BEFORE INSERT ON test1
  FOR EACH ROW
  BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = N
  END;
|

DELIMITER ;
```

Database state

test1	test2	test3	test4	
a1	a2	a3	a4	b4
1	1	~~1~~	1	3
3	3	2	2	0
1	1	~~3~~	3	1
7	7	4	4	1
1	1	5	5	0
8	8	6	6	0
4	4	7	7	1
		8	8	1
		9	9	0
		10	10	0

Trigger - Example - 03 (a)

```
INSERT INTO test1 VALUES (4);
```

```
DELIMITER |

CREATE TRIGGER testref BEFORE INSERT ON test1
  FOR EACH ROW
  BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = N
  END;
|

DELIMITER ;
```

Database state

Trigger - Example - 03 (a)

```
INSERT INTO test1 VALUES (4);
```

```
DELIMITER |

CREATE TRIGGER testref BEFORE INSERT ON test1
  FOR EACH ROW
  BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = N
  END;
|

DELIMITER ;
```

Database state

test1	test2	test3	test4	
a1	a2	a3	a4	b4
1	1	~~1~~	1	3
3	3	2	2	0
1	1	~~3~~	3	1
7	7	~~4~~	4	2
1	1	5	5	0
8	8	6	6	0
4	4	7	7	1
4	4	8	8	1
		9	9	0
		10	10	0

Multiple Triggers On Same Table

Multiple Triggers

- Multiple triggers can be placed on a single table
- Source of multiple triggers are due to the way a trigger is created

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | DELETE | UPDATE }
ON table_name
{FOLLOWS | PRECEDES}
```

- When multiple triggers exists on same table, they must be ordered
- The ordering is specified at the time of creation
- $Trigger_1 \rightarrow Trigger_2 \rightarrow Trigger_3 \cdots$
- $Trigger_2$ follows $Trigger_1$
- $Trigger_3$ follows $Trigger_2$ and so on

Multiple Triggers On Same Table

Example

```
CREATE TABLE T2 (
    id INT,
    productCode VARCHAR(15) NOT NULL,
    price DECIMAL(10,2) NOT NULL,
    updated_at TIMESTAMP NOT NULL
           DEFAULT CURRENT_TIMESTAMP
           ON UPDATE CURRENT_TIMESTAMP,
    PRIMARY KEY (id),
    FOREIGN KEY (productCode)
        REFERENCES T1 (productCode)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);
```

Multiple Triggers On Same Table

Example

```
DELIMITER |

CREATE TRIGGER before_products_update
BEFORE UPDATE ON T1
FOR EACH ROW
BEGIN
   IF OLD.msrp <> NEW.msrp THEN
      INSERT INTO T2(product_code, price)
      VALUES(old.productCode, old.msrp);
   END IF;
END|

DELIMITER ;
```

Multiple Triggers On Same Table

Example

```
SELECT
    productCode,
    msrp
FROM
    T1
WHERE
    productCode = 'S12_1099';
```

productCode	msrp
S12_1099	194.57

Multiple Triggers On Same Table

Example

```
UPDATE T1
SET msrp = 200
WHERE productCode = 'S12_1099';
```

T2

id	productCode	price	updated_at
1	S12_1099	194.57	2019-09-08 09:07:02

Multiple Triggers On Same Table

Example

```
CREATE TABLE T3 (
      id INT,
      productCode VARCHAR(15) DEFAULT NULL,
      updatedAt TIMESTAMP NOT NULL
                       DEFAULT CURRENT_TIMESTAMP
           ON UPDATE CURRENT_TIMESTAMP,
      updatedBy VARCHAR(30) NOT NULL,
     PRIMARY KEY (id),
     FOREIGN KEY (productCode)
          REFERENCES T1 (productCode)
          ON DELETE CASCADE
          ON UPDATE CASCADE
);
```

Multiple Triggers On Same Table

Example

- Table T1 has one trigger on BEFORE UPDATE to insert some content into T2
- We now set another trigger on BEFORE UPDATE on T1 to insert some content into T3

Multiple Triggers On Same Table

Example

```
DELIMITER |

CREATE TRIGGER before_products_update_log_user
   BEFORE UPDATE ON T1
   FOR EACH ROW
   FOLLOWS before_products_update
BEGIN
    IF OLD.msrp <> NEW.msrp THEN
    INSERT INTO
            T3(productCode,updatedBy)
        VALUES
            (OLD.productCode,USER());
    END IF;
END|

DELIMITER ;
```

Multiple Triggers On Same Table

Example

```
UPDATE
    T1
SET
    msrp = 220
WHERE
    productCode = 'S12_1099';
```

T2			
id	productCode	price	updated_at
1	S12_1099	194.57	2019-09-08 09:07:02
2	S12_1099	200.00	2019-09-08 09:10:32

Multiple Triggers On Same Table

Example

```
UPDATE
    T1
SET
    msrp = 220
WHERE
    productCode = 'S12_1099';
```

T3		
productCode	UpdatedAt	UpdatedBy
S12_1099	2019-09-08 09:10:32	root@localhost

System Information

Obtaining All Triggers

```
SHOW TRIGGERS
FROM classicmodels
WHERE ‘table‘ = ’T1’;
```

TRIGGERS

Trigger	Event	Table	Statement	Timing
before_products_update	UPDATE	T1	BEGIN IF old.msrp ...	BEFORE
before_products_update_log_user	UPDATE	T1	BEGIN IF OLD.msrp ...	BEFORE

System Information

Action Order

```
SELECT
    trigger_name,
    action_order
FROM
    information_schema.triggers
WHERE
    trigger_schema = 'classicmodels'
ORDER BY
    event_object_table ,
    action_timing ,
    event_manipulation;
```

information_schema	
TRIGGER_NAME	ACTION_ORDER
before_products_update	1
before_products_update_log_user	2

Nested Triggers

Example

1. Place a trigger on table T1 with some action (say when a row gets inserted)
2. Place a trigger on table T2 with action that on insert, invoke a trigger to update table T3
3. When a row gets inserted into T1, it invokes first trigger
4. Invocation of first triggers causes invocation of second trigger

Recursive Triggers

Types

Direct recursion: Occurs when a trigger fires and performs an action that causes the same trigger to fire again

Indirect recursion: Occurs when a trigger fires and performs an action that causes a trigger on another table to fire... that causes original trigger to fire again

Considerations for Using Triggers

Considerations

- Constraints are proactive
- Triggers are reactive
- Constraints are checked before triggers
- Multiple triggers can be placed for an action
- Each trigger must be sequenced

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

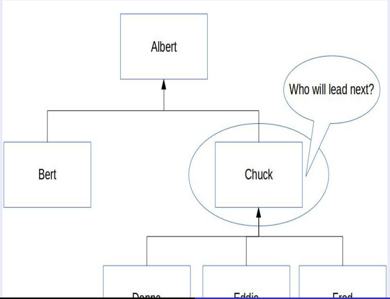
- Instruct the cursor to open in `SCROLL` model before the keyword `CURSOR`
- `EXEC SQL DECLARE name SCROLL CURSOR FOR MovieExec;`
- This will tell SQL that cursor may be used in a manner other than moving in forward direction alone
- The FETCH is responsible for specifying the direction from which the 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	⊥	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

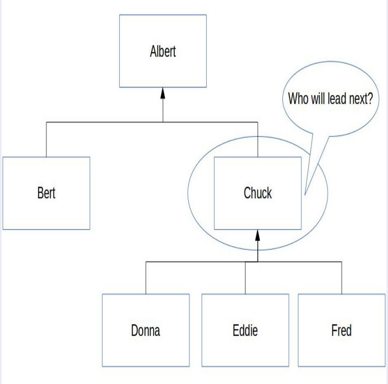
INSERT Can include cycles in the graph

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

```
CREATE FUNCTION TreeTest() RETURNS CHAR(6)
BEGIN ATOMIC
-- put a copy in a temporary table
      INSERT INTO Tree SELECT eid, boss_id FROM Manages

-- prune the leaves
        WHILE ((SELECT COUNT(*) FROM Tree) - 1) = (SELECT COUNT(boss_id) FROM Tree)
        DO
                DELETE FROM Tree
              -- Check employee is not the boss
              WHERE Tree.eid
              NOT IN (
                      -- Select all the bosses
                      SELECT T2.boss_id
                      FROM Tree AS T2
                      WHERE NOT ISNULL(T2.boss_id)
               );

    IF NOT EXISTS (SELECT * FROM Tree)
    THEN
      RETURN ('Tree');
␣␣␣␣ELSE
␣␣␣␣␣␣RETURN␣('Cycles');
    END IF;
  END WHILE;
END;
```

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
------------------------------------------------------------------------
```

Exceptions

SQL exception - 01

- An SQL system indicates error conditions by setting non-zero sequence of digits in SQLSTATE
- Example 02000 `no tuple found`
- Example 21000 `single row select has returned more than one row`
- We can declare user defined exceptions called exception handler
- Invoked whenever one of a list of these error codes appear in SQLSTATE during execution of a statement
- Each exception handler is associated with a block of code
- delineated by `BEGIN` ... `END`

Exceptions

SQL exception - 02

- The form of a handler declaration is
- DECLARE [where to go] HANDLER FOR [condition list] [statement]
- where to go:

CONTINUE means that after executing the statement in the handler declaration, we execute the statement after the one raised the exception

EXIT after executing the handler's statement, control leaves `BEGIN` … `END` block in which the handler is declared

UNDO Same as EXIT which a difference that any changes to the database or local variables that were made by the block executed so far are undone