Outline

- Recursion
- Triggers
- Application programming

Recursion

Example: find Bart’s ancestors

- “Ancestor” has a recursive definition
  - X is Y’s ancestor if
    - X is Y’s parent, or
    - X is Z’s ancestor and Z is Y’s ancestor

Recursion in SQL

- SQL2 has no recursion
  - You can find Bart’s parents, grandparents, great grandparents, etc.
  - But you cannot find all his ancestors in a single query
- SQL3 proposal has recursion
  - WITH RECURSIVE statements
  - Implemented by DB2

Ancestor query in SQL3

WITH

RECURSIVE Ancestor(ancestor, descendent) AS
(SELECT * FROM ParentChild) + Base case
UNION
(SELECT a1.ancestor, a2.descendent FROM Ancestor AS a1, Ancestor AS a2 WHERE a1.descendent = a2.ancestor) + Recursion

SELECT ancestor FROM Ancestor
WHERE descendent = 'Bart';

Linear recursion

- Technically, SQL3 only requires support of linear recursion: each RECURSIVE definition has at most one reference to a recursively-defined table
- Can we make the ancestor query linear?

WITH

RECURSIVE Ancestor(ancestor, descendent) AS
(SELECT * FROM ParentChild) + Base case
UNION
(SELECT parent, descendent FROM ParentChild, Ancestor WHERE child = ancestor)

SELECT ancestor FROM Ancestor
WHERE descendent = 'Bart';
Fixed point of a function

- If \( f : T \rightarrow T \) is a function from a type \( T \) to itself, a fixed point of \( f \) is a value \( x \) such that \( f(x) = x \).
- Example: What is the fixed point of \( f(x) = x / 2 \)?
  - 0, because \( f(0) = 0 / 2 = 0 \).
- To compute a fixed point of \( f \):
  - Start with a "seed": \( x \leftarrow x_0 \).
  - Compute \( f(x) \):
    - If \( f(x) = x \), stop; \( x \) is fixed point of \( f \).
    - Otherwise, \( x \leftarrow f(x) \); repeat.
- Example: compute the fixed point of \( f(x) = x / 2 \):
  - With seed 1: 1, 1/2, 1/4, 1/8, 1/16, … → 0.

Fixed point of a query

- A query \( q \) is just a function that maps an input table to an output table, so a fixed point of \( q \) is a table \( T \) such that \( q(T) = T \).
- To compute fixed point of \( q \):
  - Start with an empty table: \( T \leftarrow \emptyset \).
  - Evaluate \( q \) over \( T \):
    - If the result is identical to \( T \), stop; \( T \) is a fixed point.
    - Otherwise, let \( T \) be the new result; repeat.
  - Starting from \( \emptyset \) produces the unique minimal fixed point (assuming \( q \) is monotonic).

Finding ancestors

**RECURSIVE** Ancestor(ancestor, descendent) AS
(SELECT * FROM ParentChild)
UNION
(SELECT parent, descendent
FROM ParentChild, Ancestor
WHERE child = ancestor)
WHERE child = ancestor

```
Ancestor | Descendent
--------- | -----------
Homer     | Bart       
Homer     | Lisa       
Marge     | Bart       
Marge     | Lisa       
Abe       | Homer      
Abe       | Lisa       
Abe       | Homer      
```

Intuition behind fixed-point iteration

- Initially, we know nothing about ancestor-descendent relationships.
- In the first step, we deduce that parents and children form ancestor-descendent relationships.
- In each subsequent step, we use the facts deduced in previous steps to get more ancestor-descendent relationships.
- We stop when no new facts can be proven.

Mixing negation with recursion

- If \( q \) is non-monotonic:
  - The fixed-point iteration may flip-flop and never converge.
  - There could be multiple minimal fixed points—so which one is the right answer?
- Want to know more?
  - Maybe another, more theoretical database course.
  - Or take an AI course.
  - Or read the two-volume Ullman book, Database and Knowledge-Base Systems.

Trigger

- A trigger is an event-condition-action rule:
  - When event occurs, test condition; if condition is satisfied, execute action.
  - An "active database" feature.
- Example:
  - Event: whenever there comes a new student…
  - Condition: with GPA higher than 3.0…
  - Action: then make him/her take CPS 216!
Trigger example

CREATE TRIGGER CPS216AutoRecruit
AFTER INSERT ON Student
REFERENCING NEW AS newStudent
FOR EACH ROW
WHEN (newStudent.GPA > 3.0)
INSERT INTO Enroll
VALUES(newStudent.SID, 'CPS 216');

Trigger options

- Possible events include:
  - INSERT ON table
  - DELETE ON table
  - UPDATE [OF column] ON table
- Trigger can be activated:
  - FOR EACH ROW modified
  - FOR EACH STATEMENT that performs modification
- Action can be executed:
  - AFTER or BEFORE the triggering event

Transition variables

- OLD: the modified row before the triggering event
- NEW: the modified row after the triggering event
- OLD_TABLE: a hypothetical read-only table containing all modified rows before the triggering event
- NEW_TABLE: a hypothetical table containing all modified rows after the triggering event
- Not all of them make sense all the time, e.g.
  - AFTER INSERT statement triggers
  - Can use only NEW_TABLE
  - BEFORE DELETE row triggers
  - Can use only OLD
  - etc.

Statement trigger example

CREATE TRIGGER CPS216AutoRecruit
AFTER INSERT ON Student
REFERENCING NEW_TABLE AS newStudents
FOR EACH STATEMENT
INSERT INTO Enroll
SELECT SID, 'CPS 216'
FROM newStudents
WHERE GPA > 3.0
AND SID NOT IN
(SELECT SID FROM Enroll
WHERE CID = 'CPS 216');

Another trigger example

Give faculty a raise if all GPAs increase (in one update)
CREATE TRIGGER AutoRaise
AFTER UPDATE OF GPA ON Student
REFERENCING OLD_TABLE AS o
NEW_TABLE AS n
FOR EACH STATEMENT
WHEN (NOT EXISTS(SELECT * FROM o, n
WHERE o.SID = n.SID
AND o.GPA >= n.GPA))
UPDATE Faculty SET salary = salary + 1000;
- A row trigger would be hard to write and inefficient

Yet another trigger example

Never give faculty more than 50% raise in one update
CREATE TRIGGER NotTooGreedy
BEFORE UPDATE OF salary ON Faculty
REFERENCING OLD AS o NEW AS n
FOR EACH ROW
WHEN (n.salary > 1.5 * o.salary)
SET n.salary = 1.5 * o.salary;
- BEFORE triggers are often used to “condition” data
Implementation issues

- Recursive firing of triggers
  - Action of one trigger causes another trigger to fire
  - Can get into an infinite loop
    - Some DBMS restrict trigger actions
    - Most DBMS set a maximum level of recursion (16 in DB2)
- Interaction with constraints (very tricky to get right!)
  - When do we check if a triggering event violates constraints?
    - After a BEFORE trigger (so the trigger can fix a potential violation)
    - Before an AFTER trigger
  - AFTER triggers also see the effects of, say, cascaded deletes caused by referential integrity constraint violations
    (Based on DB2; no two DBMS implement the same policy!)

Programming in SQL

- Idea: Instead of making SQL do more, just use it together with a general-purpose programming language
  - Embedded SQL
  - JDBC (and ODBC, Perl DBI, etc.)

Embedded SQL

- EXEC SQL BEGIN DECLARE SECTION;
  - float thisGPA;
  - printf("%f", thisGPA);
- EXEC SQL END DECLARE SECTION;

- Declaration of a cursor, used to loop over a set of results
  - EXEC SQL DECLARE CPS216Student CURSOR FOR
    SELECT SID, GPA FROM Student
    WHERE SID IN (SELECT SID FROM Enroll
    WHERE CID = 'CPS 216')
    FOR UPDATE;

- You can update through the cursor if it is clear how to map a result row to a real row in a database table

Issues when embedding SQL

- Which statements are SQL?
  - A special preprocessor directive EXEC SQL
- How are the values passed from the host program into SQL commands?
  - Explicitly declared shared variables that are accessible to both SQL and the host program (preprocessor will insert conversion code if necessary)
- How are the results of SQL queries returned into program variables?
  - For a query returns a scalar, use SELECT INTO
  - For a query returns a set, use a cursor

Embedded SQL example

EXEC SQL BEGIN DECLARE SECTION;
    int thisSID; float thisGPA;
EXEC SQL END DECLARE SECTION;

Shared variable declarations

EXEC SQL DECLARE CPS216Student CURSOR FOR
    SELECT SID, GPA FROM Student
    WHERE SID IN (SELECT SID FROM Enroll
    WHERE CID = 'CPS 216')
    FOR UPDATE;

- You can update through the cursor if it is clear how to map a result row to a real row in a database table

More embedded SQL

EXEC SQL OPEN CPS216Student; // Open result set
EXEC WHENEVER NOT FOUND DO break; // Exit condition
while (1) {
    Read through the cursor
    EXEC SQL FETCH CPS216Student INTO :thisSID, :thisGPA;
    printf("SID %d: current GPA is %f\n", thisSID, thisGPA);
    scanf("%f", &thisGPA);
}

- Update through the cursor
EXEC SQL UPDATE Student SET GPA = :thisGPA
    WHERE CURRENT OF CPS216Student;
EXEC SQL CLOSE CPS216Student; // Close result set
Dynamic SQL

- Embedded SQL is fine for “canned” queries, but how do we write a generic query interface?
- Two special statements to make it dynamic

```sql
EXEC SQL BEGIN DECLARE SECTION;
char query[MAX_Q_LEN];
EXEC SQL END DECLARE SECTION;
```

```sql
while (1) {
    /* issue SQL> prompt */
    /* read user input into query */
    EXEC SQL PREPARE q FROM :query;
    EXEC SQL EXECUTE q;
}
```

Ship query to DBMS and get it compiled; return a handle
Use handle to execute query

Limitations of embedded SQL

- Not very portable
- Cannot talk to different DBMS at the same time

Need to compile the application for each DBMS because different DBMS use different preprocessors and different libraries

JDBC

- Solution: one more level of indirection through drivers
  - Same idea as ODBC, Perl DBI, etc.

```
Connection conn = DriverManager.getConnection(url, uid, password);
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM Student");
while (rs.next()) {
    int sid = rs.getInt(1);
    String name = rs.getString(2);
    System.out.println("SID: " + sid + " name: " + name);
}
stmt.close();
```

Connecting to the database can easily take the longest time!
Connection pooling!

More JDBC example

```java
conn.setTransactionIsolation(TRANSACTION_SERIALIZABLE);
conn.setAutoCommit(false);
```

Performing an insert
```
PreparedStatement pstmt = conn.prepareStatement("INSERT INTO Student(SID, name) VALUES(?, ?)");
// read sid and name from input
pstmt.setInt(1, sid);
pstmt.setString(2, name);
pstmt.executeUpdate();
pstmt.close();
```

Getting ready for transactions

Review of introductory materials

- Relational model and relational algebra
- Relational design theory
  - FD, MVD, BCNF...
- SQL
  - Query: SFWGHO, subqueries, NULL, recursion
  - Constraints and triggers
- Transaction processing
  - Concurrency control and recovery
- Programming with SQL
  - Embedded SQL
  - JDBC