Odds and Ends of SQL

CPS 216
Advanced Database Systems

Outline

- Recursion
- Triggers
- Application programming

Recursion

- Example: find Bart’s ancestors
- “Ancestor” has a recursive definition
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)
    UNION
    (SELECT a1.ancestor, a2.descendent
    FROM Ancestor AS a1, Ancestor AS a2
    WHERE a1.descendent = a2.ancestor)
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)
    UNION
    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 $f(x) = x / 2$?

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 $f(x) = x / 2$.

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)
```

```
<table>
<thead>
<tr>
<th>ancestor</th>
<th>descendent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Homer</td>
<td>Bart</td>
</tr>
<tr>
<td>Homer</td>
<td>Lisa</td>
</tr>
<tr>
<td>Marge</td>
<td>Lisa</td>
</tr>
<tr>
<td>Marge</td>
<td>Bart</td>
</tr>
<tr>
<td>Abe</td>
<td>Homer</td>
</tr>
<tr>
<td>Abe</td>
<td>Lisa</td>
</tr>
<tr>
<td>Abe</td>
<td>Bart</td>
</tr>
<tr>
<td>Abe</td>
<td>Lisa</td>
</tr>
<tr>
<td>Abe</td>
<td>Bart</td>
</tr>
<tr>
<td>Ape</td>
<td>Homer</td>
</tr>
<tr>
<td>Ape</td>
<td>Lisa</td>
</tr>
<tr>
<td>Ape</td>
<td>Bart</td>
</tr>
<tr>
<td>Ape</td>
<td>Lisa</td>
</tr>
</tbody>
</table>
```
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

- 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
  – BEFORE DELETE row triggers
  – 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 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 ( )
))
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

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

- 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 ...;
  float thisGPA;
  EXEC SQL FETCH ...;
  printf("%f", thisGPA);
EXEC SQL END DECLARE ...;
```

Host program with special SQL directives and commands

Preprocess using the preprocessor provided by DBMS

```
float thisGPA;
sql("SELECT ...;
  printf("%f", thisGPA);
```

Host program with special DBMS API calls

Compile and link with libraries provided by DBMS

Binary executable

Client

DBMS

Server
### 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

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

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

EXEC SQL OPEN CPS216Student;
EXEC WHENEVER NOT FOUND DO break;
while (1) {
  EXEC SQL FETCH CPS216Student INTO :thisSID, :thisGPA;
  printf("SID %d: current GPA is %f\n", thisSID, thisGPA);
  printf("Enter new GPA: ");
  scanf("%f", &thisGPA);
  EXEC SQL UPDATE Student SET GPA = :thisGPA
  WHERE CURRENT OF CPS216Student;
}
EXEC SQL CLOSE CPS216Student;
```

---

### More embedded SQL

```sql
EXEC SQL OPEN CPS216Student;
EXEC WHENEVER NOT FOUND DO break;
while (1) {
  EXEC SQL FETCH CPS216Student INTO :thisSID, :thisGPA;
  printf("SID %d: current GPA is %f\n", thisSID, thisGPA);
  printf("Enter new GPA: ");
  scanf("%f", &thisGPA);
  EXEC SQL UPDATE Student SET GPA = :thisGPA
  WHERE CURRENT OF CPS216Student;
}
EXEC SQL CLOSE CPS216Student;
```
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;
while (1) {
    /* issue SQL> prompt */
    /* read user input into query */
    EXEC SQL PREPARE q FROM :query;
    EXEC SQL EXECUTE q;
}
```

Limitations of embedded SQL

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

JDBC

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

```
  Application
     /\    /
    /  \  /   
   /    \|
  Driver Driver
  Oracle Oracle
  Oracle DB2
```
JDBC example

```java
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();
```

More JDBC example

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

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.execute();
pstmt.close();
conn.commit();
```

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