Server-Side Database Programming

COM 3563: Database Implementation

Avraham Leff

Yeshiva University

avraham.leff@yu.edu

COM3563: Fall 2020

Today's Lecture: Overview

1. Introduction
2. Functions & Procedures
3. Triggers
4. Recursive Queries

Context

- Previous lectures introduced the issue of "database programming"
  - *"How to write applications that involve both a relational database component and a non-relational database component?"*
- "Pure SQL" approach can't do the entire job
- Traditional "host language" approach can't do the entire job
- Previous two lectures: extend the host language with the ability to interact with a relational database
- Today: extend SQL with "non-SQL" capabilities
  - I'll refer to this approach as server-side programming

Are There Limits To Server-Side Programming?

- I'd have thought that "server-side" programming is (only) about extending set-based relational operators with procedural constructs
- Examples
  - Conditional branching statements
  - Constructs for looping
  - Stored procedures
- But from a technical perspective, "code is code" ☺
- If you can send email from a "host language" application, there is no reason why you can't send email from an relational database
  - Once you've added the notion of a "stored procedure" to a relational database server, really, you're only limited by your imagination
  - See How to send email from SQL Server?
  - It may be a conceptual "abomination", but it can be done ☺

The Argument Is About "Business Logic"

- Every (non-trivial) application contains multiple components
  - "Application logic": how does the application interact with the end-user?
    - Example: web-pages & forms
    - Example: user login
  - "Business logic": the application's core computation
    - Example: create user accounts
    - Example: compute interest on a user's holdings
    - Example: moving funds between a user's accounts
  - "Data integrity": error-checking to prevent "garbage" from getting into the database
- Everyone agrees that "application logic" does not belong in the database
- Everyone agrees that "application data" does belong in the database
- The approaches differ about *"where to put an application's business logic?"*

Business Logic: The Case For “Application Side”

- View the DB as just “dumb storage”, and place even the data-integrity component in the application
- Scaling issues?
  - No problem: just add application servers (that’s the “mid-tier”) to handle the load
- Use the application to create a single “skin” that masks differences between different DB vendors
  - Avoid vendor “lock-in”!
- Larger pool of good software engineers than good SQL engineers
- Database team not interested in working with the application team ☹

Business Logic: The Case For "DB Side"

- The DB is already responsible for maintaining application data
  - Data integrity belongs in the DB side
  - Use stored procedures and triggers to do the "non-sql" work inside the DB itself
- Applications can still "extend" the available DB function
  - But now the application talks to a larger chunk of encapsulated server-side logic
  - Application programmers only need have minimal SQL skills
- Performance advantage(s)
  - Reduce client-server communication costs
  - Maximize query optimization
- Security advantage(s)
  - Expose minimum of DB data & structure
  - Protect data from incompetent application programmers ☺

Where Does "Error Checking" Go?

- *"I understand that application logic remains on the client-side of the fence ..."*
- *"I understand that reasonable people can differ as to where the business logic should go ..."*
- Q: but where should error checking logic go?
- A: everywhere!
- Application side:
  - Application is more responsive if it can avoid making a DB call with bad input (this is a big motivation for e.g., using Javascript in a web-page)
  - Application can usually provide better feedback to the user ("closer to the mistake")
- DB side:
  - Some errors are easier to detect when you have all the relevant data
  - DB is already checking for data and referential integrity
  - Never trust someone else to do your error checking ☺

“Application Side”: Efficiency & Expressiveness

- Before we leave this topic, note that even the “enhanced sql” approach can be much less suitable than application side approach in certain scenarios …
- Example: computation efficiency
  - Especially for parallel computation
- Example: object-oriented “expressiveness”
  - Store polygon or image data in the DB
  - But the function that computes whether polygons overlap or whether images are “similar” should (probably) reside in the application

Functions & Procedures

- SQL:1999 supports functions and procedures that are stored in the database and executed from SQL statements
- Functions and procedures can be written
  - Either: in SQL itself, using the procedural extensions to SQL
  - Or: in an language such as C and Java
- I'll quickly discuss the "C and Java" approach
  - It's pretty boring and straight-forward ☺
  - The key point is that the capability is available and has been standardized
  - Unfortunately, the standard is most definitely not supported uniformly by the various vendors ☹

Note: this capability shows that the "host language" issue is orthogonal to the code placement issue

```
create procedure dept_count_proc
(in dept_name varchar(20), out count integer)
language C
external name '/var/dba/bin/dept_count_proc'

create function dept_count
(dept_name varchar(20))
returns integer
language C
external name '/var/dba/bin/dept_count'
```

- These examples only show the declaration part of a stored function or procedure
  - You add local variable declarations and the function/procedure body as needed
  - Only difference between a “procedure” and “function” is that the latter declares a return type
- These examples are for an external language (“C”)
  - We therefore specify both the language and the file name where the program code is stored
  - Not necessary if you implement in SQL

Using External Language Routines

- Each parameter should have a parameter type that is one of the SQL data types
- Each parameter should also have a parameter mode: one of **IN, OUT, or INOUT**
- These correspond to parameters whose values are:
  - Input only
  - Output (“value will be set by the routine and returned to the client”) only
  - Both input and output
- Procedures and functions are stored persistently by the DBMS, and can therefore be invoked any of the interfaces that you already know about
- The SQL standard specifies that you invoke a routine through the CALL statement

```
CALL <procedure or function name> (<argument list>);
```

- Invoke
  - Either: through the command-line processor
  - Or: embedded SQL
  - Or: JDBC (*via* the *CallableStatement* class)

Evaluating Benefits of External Language Routines (I)

You already know these points (quick review)

- External language routines are usually more efficient than client-side SQL
  - Pre-compiled and bound to specific database configuration
  - No network traffic: code & data both reside in the database engine
  - Easier to enforce integrity rules & authorization constraints
  - Much easier to protect against SQL injection attacks
- External language routines are usually more expressive than "pure SQL"
- Probably more programmers available
- Biggest disadvantage: proprietary syntax ☹
- Also: having to get the DBA sign-off implies very slow development time ☹
- Finally: you have to consider the security risk (next slide)

Evaluating Benefits of External Language Routines (II)

- Part of the "efficiency advantage" comes from loading the routine into the database system so that it can execute in the DBMS's own address space
- Major risks:
  - Risk of accidental corruption of database structures
  - Security risk, allowing users access to unauthorized data
  - Basic problem: SQL code is (implicitly) sand-boxed, but we can't make "safety guarantees" for more general languages
- One approach: run the external language code outside the DBMS address space
  - Can't corrupt internal data-structures
  - If the process goes down, doesn't take the database with it
- Note: this solution gives up some of the server-side performance benefits
  - Although we still gain the "communication costs" benefits
- Note also: can use a safer (compared to C and C++) language such as Java ☺

SQL Functions: Example

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

```sql
create function dept_count (dept_name varchar(20))
-- type of return value
returns integer
-- compound statement, bracketed by
-- 'begin' and 'end'
begin
declare d_count  integer;
select count (*) into d_count
from instructor
where instructor.dept_name = dept_name
-- return the value
return d_count;
end
```

Use the dept_count function to find the department names and budget of all departments with more that 12 instructors

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

- You can think of an SQL function as a parameterized view
  - Generalizes the concept of a view by allowing parameters
- Table functions (SQL:2003) are an even better fit for SQL: these are functions that return a relation
  - Example: *"Return all instructors in a given department"*

```sql
-- definition
CREATE FUNCTION instructor_of (dept_name char(20))
  RETURNS TABLE (ID varchar(5), name varchar(20),
                 dept_name varchar(20), salary numeric(8,2))

     RETURN TABLE (SELECT ID, name, dept_name, salary
        FROM instructor
        WHERE instructor.dept_name = instructor_of.dept_name)

-- Invoked as
SELECT * FROM TABLE (instructor_of ('Music'))
```

SQL/PSM: Server-Side Hybrid Approach (I)

- The SQL functions that we've just discussed are really an encapsulation approach for SQL
- They don't fundamentally transform the set-based foundations of SQL
- We're now going to take a look at **SQL/PSM (SQL/Persistent Stored Modules)**
  - SQL/PSM *"is an ISO standard mainly defining an extension of SQL with a procedural language for use in stored procedures"*
- SQL/PSM is thus a true server-side hybrid approach to database programming
- It supports a rich set of imperative constructs, including loops, if-then-else, assignment, exception conditions, and exception handlers
- Key issue: should you be using these constructs at all?
- The following is my opinion only ...
  - Don't go there! ☺
  - Or: *"go there only when the bulk of the code is pure SQL"*

- One good reason to avoid SQL/PSM is that many databases only provide proprietary extensions to the standard ☹
- As usual, this course is interested in discussing examples that you can run on your computer: specifically POSTGRESQL
- POSTGRESQL provides PL/pgSQL (*Procedural Language/PostgreSQL*)
  - POSTGRESQL also provides language support for Tcl, Perl, and Python
  - Third-party support for Java and many other languages
- I'm going to begin with an example of the (IMNSHO) sort of code that should not be implemented on the server-side
  - Regardless of the language support!
- But: this should give you a sense of how much procedural code you can write in PL/pgSQL

“Wow! I Didn’t Know You Can Do This”

```sql
-- https://www.postgresqltutorial.com/plpgsql-loop-statements/
CREATE OR REPLACE FUNCTION fibonacci (n INTEGER)
        RETURNS INTEGER AS $$
DECLARE
   counter INTEGER := 0 ;
   i INTEGER := 0 ;
   j INTEGER := 1 ;
BEGIN

        IF (n < 1) THEN
                RETURN 0 ;
        END IF;

        LOOP
                EXIT WHEN counter = n ;
                counter := counter + 1 ;
                SELECT j, i + j INTO i, j ;
        END LOOP ;

        RETURN i;
END ;
$$ LANGUAGE plpgsql;
```

A More Reasonable Example

```sql
-- definition
CREATE FUNCTION Dept_size(IN deptno INTEGER)
  RETURNS VARCHAR [7]
  RETURNS NULL ON NULL INPUT
  As $$
    DECLARE No_of_emps INT;
    BEGIN
      SELECT COUNT(*) INTO No_of_emps
        FROM EMPLOYEE WHERE Dno = deptno ;
      IF No_of_emps > 100 THEN RETURN "HUGE";
      ELSEIF No_of_emps > 25 THEN RETURN "LARGE";
      ELSEIF No_of_emps > 10 THEN RETURN "MEDIUM";
      ELSE RETURN "SMALL";
      END IF;
    END $$
    LANGUAGE plpgsql
    IMMUTABLE;

-- Invoked as
SELECT Dept_size(42);
```

- Warning: much online POSTGRESQL documentation states that “procedures and functions” are identical
  - Only difference is that functions return results, procedures do not return results
    - This was true pre-POSTGRESQL v11
  - Now: different syntax, and invoked differently
    - SELECT for function, CALL for procedure

Even More Reasonable Example

```
-- Example taken from Example 43.2 in Postgres documentation
-- Chapter 43.6 "Control Structures"
CREATE TABLE db (a INT PRIMARY KEY, b TEXT);
CREATE FUNCTION merge_db(key INT, data TEXT) RETURNS VOID AS
$$
BEGIN
    LOOP
        -- first try to update the key
        UPDATE db SET b = data WHERE a = key;
        IF found THEN
            RETURN;
        END IF;
        -- not there, so try to insert the key
        -- if someone else inserts the same key concurrently,
        -- we could get a unique-key failure
        BEGIN
            INSERT INTO db(a,b) VALUES (key, data);
            RETURN;
        EXCEPTION WHEN unique_violation THEN
            -- Do nothing, and loop to try the UPDATE again.
        END;
    END LOOP;
END;
$$
LANGUAGE plpgsql;

-- Invoked as
SELECT merge_db(1, 'david');
SELECT merge_db(1, 'dennis');
```

- Biggest difference now: you cannot run a transaction in a function, only in a procedure (stay tuned)

Running Example For Remainder Of Lecture

- "Functions and procedures" are straightforward
- Only issue: getting the syntax correct ☹
  - PL/pgSQL is effectively a new language for you to learn
- However: the concept of triggers (last portion of today's lecture) is much less straightforward
  - Especially in the way that PL/pgSQL supports the concept
- I'll present an end-to-end example, beginning with a vanilla `accounts` table

```sql
CREATE TABLE accounts (
  id INT GENERATED BY DEFAULT AS IDENTITY,
  first_name VARCHAR(100) NOT NULL,
  last_name VARCHAR(100) NOT NULL,
  balance DEC(15,2) NOT NULL,
  PRIMARY KEY(id)
);

INSERT INTO accounts(first_name, last_name, balance) VALUES('Bob', 'Jones', 10000);
INSERT INTO accounts(first_name, last_name, balance) VALUES('Alice', 'Brown', 50000);
SELECT * FROM accounts ORDER BY ID ASC;
```

```
 id | first_name | last_name | balance
----+------------+-----------+----------
  1 | Bob        | Jones     | 10000.00
  2 | Alice      | Brown     | 50000.00
(2 rows)
```

Encapsulate A `transfer` Between Accounts

```
CREATE OR REPLACE PROCEDURE transfer(sourceAccount INT, destinationAccount INT, amount DEC)
LANGUAGE plpgsql
AS $$
BEGIN
   RAISE NOTICE 'Transferring % Dollars from account % to account %', amount, sourceAccount, destinationAccount;
    -- subtracting the amount from the sender's account
    UPDATE accounts
    SET balance = balance - amount
    WHERE id = sourceAccount;

    -- adding the amount to the receiver's account
    UPDATE accounts
    SET balance = balance + amount
    WHERE id = destinationAccount;

    IF amount < 5000 THEN
        RAISE NOTICE 'Funds transfer succeeded';
       COMMIT;
    ELSE
      -- doesn't seem to echo back to user
      RAISE WARNING 'Invalid funds transfer: amount % is too large, speak to customer service!', amount;
       ROLLBACK;
    END IF;

END;
$$;
```

- Note the use of `raise`
- Note the use of transactions

Invoking `transfer` (I)

```sql
-- Transfer 1000 from an account with id 1 (Bob) to the account with id 2
-- (Alice), should succeed

CALL transfer(1,2,1000);
SELECT * FROM accounts ORDER BY ID ASC;
```

```
psql:SimplePostgresServerSide.sql:81: NOTICE:  Transferring 1000 Dollars from account
      1 to account 2
psql:SimplePostgresServerSide.sql:81: NOTICE:  Funds transfer succeeded

SELECT * FROM accounts ORDER BY ID ASC;
 id | first_name | last_name | balance
----+------------+-----------+---------
  1 | Bob        | Jones     |  9000.00
  2 | Alice      | Brown     | 51000.00
(2 rows)
```

- As expected, Bob's balance has decreased, because `transfer` moved the $1,000 from his account to Alice's account

Invoking `transfer` (II)

```sql
-- Transfer 5050 from an account with id 1 (Bob) to the account with id 2[s
-- (Alice), should not succeed

CALL transfer(1,2, 5500);
SELECT * FROM accounts ORDER BY ID ASC;
```

```
psql:SimplePostgresServerSide.sql:87: NOTICE:  Transferring 5500 Dollars from account 1 to account 2
psql:SimplePostgresServerSide.sql:87: WARNING:  Invalid funds transfer: amount 5500 is too large, speak to customer service!
CALL
SELECT * FROM accounts ORDER BY ID ASC;
 id | first_name | last_name | balance
----+------------+-----------+----------
  1 | Bob        | Jones     |  9000.00
  2 | Alice      | Brown     | 51000.00
(2 rows
```

- Note: Bob's balance has not changed, because the transaction was rolled back

Triggers

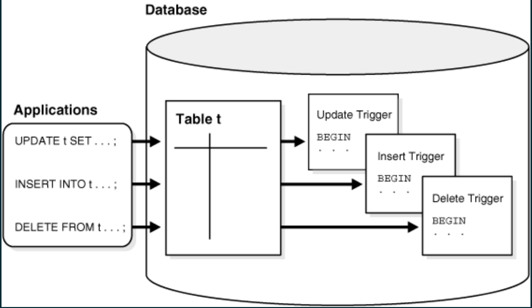
Database Triggers

- A trigger is a statement that is executed automatically by the system as a side effect of a modification to the database
- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed
  - Specify the actions to be taken when the trigger executes
- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases

> Trigger syntax is part of the SQL standard. That said, specific database systems often support triggers with syntax that differs from the textbook. As usual, we'll focus on understanding the concepts, some initial examples from the standard, then illustrate with PL/pgSQL example

Triggers Are Not Executed Explicitly

- Think of a trigger as a procedure that is stored in the database
  1. Database monitors "change events"
  2. Database determines that a change event corresponds to a trigger condition
  3. Database then executes the procedure
- Differs from stored procedures (discussed earlier) because they are not under client control
  - You do not explicitly invoke trigger execution
  - Instead: triggers are under the database's control



Triggers: Non-Motivating Scenarios

- Don't abuse triggers capability!
  - Often used when referential integrity constraints can do the job
  - Example: using a trigger to enforce requirement that value isn't NULL
  - Example: using a trigger to enforce foreign key constraint
    - Reminder: value appearing in $relation_A$ also appears as the primary key in a $relation_B$ tuple
  - Attribute based CHECK constraints
- The temptation comes from the fact that a trigger is a hook for invoking arbitrary code
  - Most of us reach instinctively for the "programming solution"
  - But: always better to use declarative SQL features when possible
- Now I'll present some motivating scenarios …

Complex Auditing

- Organizations often require an audit trail that enable auditors to track a sequence of changes made to an important database table or row
- Database logs & journals are fine …but hard to manifest at "user level"
- You want changes made to $table_A$ to result in a new row added to $table_B$ that records
  - Who made the change
  - Primary key of that tuple
  - Pre-change state
  - Post-change state
- This scenario uses the AFTER trigger (discussed later)

Business Rules

- Business categorizes customers into *Platinum*, *Gold*, *Silver*
  - Based on value of purchases in a given lagged time-frame
- Define a trigger that recomputes customer status each time that a customer record is added or modified
  - Need to sum all of the customer's purchases over the specified time-frame

Triggers: Scenario 3

Derived Attribute Values

- Business wants to maintain an up-to-date *TotalSales* attribute
- Must be modified every time customer makes a purchase
- And every time that customer returns an item!

Note: same effect can be achieved if database system supports materialized views ☺

Triggering the Trigger

You specify both:

- a trigger event (e.g., INSERT, DELETE, UPDATE)
- a trigger condition that must be satisfied before the trigger executes (e.g., BEFORE, AFTER)

```sql
-- Triggers on update can be restricted to
-- specific attributes
AFTER UPDATE OF takes ON GRADE
```

- You can reference either the pre-operation attribute values or the post-operation values
- DELETE & UPDATE

```sql
        REFERENCING OLD ROW AS old_row
```

- INSERT & UPDATE

```sql
        REFERENCING NEW ROW AS new_row
```

Time For An Example

- Can use triggers to fix a problem before the system aborts the transaction
- Another motivating scenario for triggers
  - Referential integrity constraints stops the operation
  - Trigger fixes the problem, allows operation to continue
- Example: *referential integrity does not allow blank grades*
  - Requires either a “letter grade” or NULL

```sql
CREATE TRIGGER setnull_trigger BEFORE UPDATE OF takes
REFERENCING NEW ROW as NROW
FOR EACH ROW
WHEN (nrow.grade = '')
BEGIN atomic
SET nrow.grade = null;
END;
```

What Does This Trigger Do?

```sql
CREATE TRIGGER credits_earned
   AFTER UPDATE OF takes ON (grade)
REFERENCING NEW ROW as nrow
REFERENCING OLD ROW as orow
FOR EACH ROW
WHEN nrow.grade <> 'F' and nrow.grade IS NOT NULL
    AND (orow.grade = 'F' or orow.grade IS NULL)
BEGIN atomic
     UPDATE student
     SET tot_cred= tot_cred +
           (SELECT credits
            from COURSE
            WHERE course.course_id= nrow.course_id)
     WHERE student.id = nrow.id;
END;
```

- Student was previously assigned a NULL grade or an **F** ☹
- Whenever that grade is “up-graded”, we want to update the *total number of credits* associated with that student
  - Result: student has now successfully completed the course ☺

Statement Level Triggers: I

- Previous examples were row-level
  - Executed an action for each affected row
- Statement-level triggers execute an action for the entire SQL statement that caused the “insert”, “delete” or “update”
  - Fired once per statement, regardless of the number of rows that were affected
  - Even if no rows are affected
- Scenarios
  - Perform a complex security check on the current time or user
  - Generate a single audit record

Use the FOR EACH STATEMENT syntax instead of the FOR EACH ROW syntax

```
        -- temporary relation containing all the old tuples
        referencing old table as old_table

        -- temporary relation containing all the new tuples
        referencing new table as new_table
```

Example taken from here

Create a trigger that ensures that whenever a parts record is updated, the following check and (if necessary) action is taken: If the on-hand quantity is less than 10% of the maximum stocked quantity, then issue a shipping request ordering the number of items for the affected part to be equal to the maximum stocked quantity minus the on-hand quantity.

```
CREATE TRIGGER REORDER
   AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
   REFERENCING NEW TABLE AS NTABLE
   FOR EACH STATEMENT
      BEGIN ATOMIC
         SELECT ISSUE_SHIP_REQUEST(MAX_STOCKED - ON_HAND,
             PARTNO)
            FROM NTABLE
         WHERE (ON_HAND < 0.10 * MAX_STOCKED);
      END
```

Using Triggers Involves Risks

- We've said that triggers are executed implicitly
  - Very easy to introduce bugs ☹
  - Very hard to track down because of the *"I didn't do anything new"* problem
  - And the *"But my code doesn't even hit that table!"* problem
- A single insert to $table_1$ can trigger a change in $table_2$
- So far so good ...
- Now the change in $table_2$ triggers a change in $table_3$
- And so on: this is the problem of cascading trigger execution
- Can you imagine the problem if you end up with a *circular cascade*?
- Basic problem
  - Like the dreaded goto, code semantics can no longer be easily understood by reading the code
  - But triggers (can be) *even worse*
- See discussion here (pro) and here (con)

Time For PL/pgSQL Example

- The bank maintaining the `accounts` table must allow for certain "identity" changes
  - They've already applied the "Bill Hahm" axiom, so they define identity in terms of an auto-generated key
- But: people have "life-altering" events (e.g., they get married), and the bank wants to accommodate changing either a first or a last name
- But: such important changes need to be audited
  - When was the change made, and what change was made
- Because such audit code is conceptually decoupled from the "main" (update) code, it's a good candidate for a trigger-based implementation
- In PL/pgSQL, we must follow these two steps:
  1. Create a trigger function via a `CREATE FUNCTION` statement
  2. Bind the trigger function to a table via a `CREATE TRIGGER` statement

PL/pgSQL: Create The Trigger Function

```
CREATE OR REPLACE FUNCTION log_name_changes()
  RETURNS trigger AS $$
DECLARE
BEGIN
        IF NEW.first_name <> OLD.first_name THEN
                 INSERT INTO account_audits(account_id, first_name, last_name, field_changed, new_value, changed_on)
                 VALUES(OLD.id, OLD.first_name, OLD.last_name, 'first name', NEW.first_name, now());
        END IF;

        IF NEW.last_name <> OLD.last_name THEN
                 INSERT INTO account_audits(account_id, first_name, last_name, field_changed, new_value, changed_on)
                 VALUES(OLD.id, OLD.first_name, OLD.last_name, 'last name', NEW.last_name, now());
        END IF;

        RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

PL/pgSQL: Bind The Trigger Function To `accounts` Table (I)

```sql
CREATE TRIGGER name_changes
  BEFORE UPDATE
  ON accounts
  FOR EACH ROW
  EXECUTE PROCEDURE log_name_changes();

-- Expect to see a changed-last-name record
UPDATE accounts SET last_name = 'Jones' WHERE id = 2;

-- Expect to see a change-first-name record
UPDATE accounts SET first_name = 'Robert' WHERE id = 1;
```

PL/pgSQL: Bind The Trigger Function To accounts Table (II)

```
SELECT * FROM accounts ORDER BY ID ASC;
SELECT * FROM account_audits;
```

```
-- verified that the name changes occurred
SELECT * FROM accounts ORDER BY ID ASC;
 id | first_name | last_name | balance
----+------------+-----------+----------
  1 | Robert     | Jones     |  9000.00
  2 | Alice      | Jones     | 51000.00
(2 rows)
```

```
-- verified that the name changes were
-- logged appropriately

SELECT * FROM account_audits;
 id | account_id | first_name | last_name | field_changed | new_value |
     changed_on
----+------------+------------+-----------+---------------+-----------+----------------------

  1 |          2 | Alice      | Brown     | last name     | Jones     | 2020-07-10
      14:22:35.112179
  2 |          1 | Bob        | Jones     | first name    | Robert    | 2020-07-10
      14:22:35.113545
(2 rows)
```

Introduction

- Earlier lecture claimed that SQL isn't Turing-complete
- Example: given a relation *Birth(Parent,Child)*
  - We can calculate "who is grand-parent of whom?"
  - We can calculate "who is great-grand-parent of whom?"
  - We cannot calculate "who is an ancestor of whom?" using relational algebra
- However: SQL has been "extended" (SQL:1999) to give it the power to calculate the "ancestor" (and related queries)
- Such queries are called recursive queries

Further Motivation

- The “ancestor” example is not uncommon
- Similar examples:
  - Employees have managers, managers are managed: return the “org-chart” of a given employee
  - Machines are composed of parts, parts are composed of sub-parts: return the set of parts needed to build a given machine
- These examples share the semantics of hierarchical relationships
  - And we want to formulate queries that are the transitive closure of these hierarchies

Why Are Transitive Closure Queries Difficult to Write

- Intuition: without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of the relation with itself
  - Example: fixed number of *grand*^{x}-parents
  - Example: fixed number of levels of managers
- Note: iterative solutions are possible
  - See Figure 5.13 from Textbook which solves the "find all course prerequisites" problem
- Recursive solutions are more elegant
  - So, we'll focus on that approach
- I hope that you realize that this discussion is straightforward application of your "Intro to Algorithms" course ☺

How to find all cities in U.S.A?

```
CREATE TABLE Area (
   area_id INT,
   area_name TEXT,
   area_type TEXT,
   area_parent INT,
   PRIMARY KEY (area_id),
   FOREIGN KEY (area_parent) REFERENCES Area (area_id));
```

```
INSERT INTO Area
VALUES
   (1, 'Earth', 'planet', NULL),
   (2, 'U.S.A.', 'country', 1),
   (3, 'Washington, D.C.', 'city', 2),
   (4, 'Texas', 'state', 2),
   (5, 'Harris', 'county', 4),
   (6, 'Houston', 'city', 5),
   (7, 'France', 'country', 1),
   (8, 'Ile-de-France', 'region', 7),
   (9, 'Paris', 'department', 8),
   (10, 'Paris', 'city', 9);
```

Hierarchy distance between country and city varies.

Solution outline

```
WITH USAArea AS
      (All areas within U.S.A.)

SELECT cities in USAArea;
```

Areas computed recursively – recursive CTE

```sql
WITH RECURSIVE USAArea (area_id, area_name, area_type, area_parent) AS
        (SELECT area_id, area_name, area_type, area_parent
         FROM Area
         WHERE area_name = 'U.S.A.'

         UNION

         SELECT a.area_id, a.area_name, a.area_type, a.area_parent
         FROM Area a
         INNER JOIN USAArea ua ON a.area_parent = ua.area_id)

SELECT area_id, area_name
FROM USAArea
WHERE area_type = 'city';
```

Base case

Must combine with **UNION [ALL]**

Inductive case

Evaluation traverses hierarchy breadth-first

WITH:

1. Base case selects 'U.S.A.' – id 2.
2. Union with inductive case selects children of just-added data – ids 3,4.
3. Union with inductive case selects children of just-added data – id 5.
4. Union with inductive case selects children of just-added data – id 6.
5. Union with inductive case selects children of just-added data – empty. **Induction terminates!**

SELECT:

1. Selects results with type 'city' – ids 3,6.

```
INSERT INTO Area
VALUES
  (1, 'Earth', 'planet', NULL),
  (2, 'U.S.A.', 'country', 1),
  (3, 'Washington, D.C.', 'city', 2),
  (4, 'Texas', 'state', 2),
  (5, 'Harris', 'county', 4),
  (6, 'Houston', 'city', 5),
  (7, 'France', 'country', 1),
  (8, 'Ile-de-France', 'region', 7),
  (9, 'Paris', 'department', 8),
  (10, 'Paris', 'city', 9);
```

Fibonnaci example

```
WITH RECURSIVE fib (num, f1, f2) AS
     (SELECT 0, 0, 1

      UNION ALL

      SELECT num+1, f2, f1+f2 FROM fib
      WHERE num < 20)

SELECT num, f1 AS fib_num
FROM fib;
```

Some details

- Common usage cases?

- Recursive CTEs restrict allowable queries. Restrictions vary.
  - PostgreSQL: One or more base cases. Listed before the one inductive case.
  - PostgreSQL: One use of recursion within the CTE.

- Cost of removing duplicates (UNION) might be higher than duplicating work (UNION ALL).

Find which courses are a prerequisite, whether directly or indirectly, for a specific course

```
with recursive rec_prereq(course_id, prereq_id) as (
        select course_id, prereq_id
        from prereq
    union
        select rec_prereq.course_id, prereq.prereq_id,
        from rec_rereq, prereq
        where rec_prereq.prereq_id = prereq.course_id
    )
select *
from rec_prereq;
```

The view, *rec_prereq*, is called the transitive closure of the *prereq* relation

Because You've All Taken COM 2545

```
create table arc(
  tail number,
  head number,
  primary key(tail , head)
);
insert into arc values(1,3);
insert into arc values(2,4);
insert into arc values(3,4);
insert into arc values(4,6);
insert into arc values(5,8);
insert into arc values(8,5);
insert into arc values(8,9);
with recursive route(tail , head) as
   (select a.tail , a.head
      from arc a

      union all

    select  a.tail , r.head
       from route r
       join arc a
       on a.head = r.tail
)
select * from route;
```

Example from Professor Zvi Kedem

```
ERROR: ORA-32044: cycle detected
while executing recursive WITH query
```

That is: Execution failed because the graph was not acyclic (it had a cycle)

Today's Lecture: Wrapping it Up

Readings

- Textbook discusses *server-side database programming* in Chapter 5.2
- Textbook discuss *triggers* in Chapter 5.3
- Skim Chapter 5.5: am not planning lecture on this material (no time ☹)