CSE 462 Homework #7 (Optional): Stored Proceduers and Triggers

Name: _____ Date: April 28, 2011

***** Submit electronically on May 06, 2011 no later than 11:59pm. *****

This problem set is worth 250 points. You **must** type your answers.

1. (50pts) Read sections 9.2, 9.3, and 9.4 of the textbook and skim through the following sections of the PostgreSQL documentation for an overview of some features supported by this DBMS. Then, provide short answers for the given questions.

`http://www.postgresql.org/docs/current/static/xfunc-sql.html`
`http://www.postgresql.org/docs/current/static/xfunc-overload.html`
`http://www.postgresql.org/docs/current/static/xfunc-volatility.html`
`http://www.postgresql.org/docs/current/static/plpgsql.html`
`http://www.postgresql.org/docs/current/static/sql-createfunction.html`
`http://www.postgresql.org/docs/current/static/triggers.html`
`http://www.postgresql.org/docs/current/static/sql-createtrigger.html`

a) What is a stored procedure (function)?
b) Explain the difference between the volatility categories of PostgreSQL functions.
c) Give examples of volatile, stable, and immutable functions in PostgreSQL.
d) What is a cursor?
e) What is a trigger?
f) Explain the difference between `BEFORE` and `AFTER` triggers.
g) Explain the difference between row-level and statement-level triggers (`FOR EACH ROW` and `FOR EACH STATEMENT` syntax, respectively).
h) What is the relationship between triggers and functions?

2. (100pts) For each statement below, say whether it is `true` or `false`. Then, justify your answer with either an example or a short explanation.

a) Stored procedures can be invoked from user programs, e.g., using JDBC in Java.
b) Multiple SQL statements can be encapsulated in a single stored procedure.
c) Stored procedures can be used to implement complex business logic.
d) Stored procedures and triggers can be combined to keep a complex report table up-to-date.
e) Using a volatile function in a query is always more efficient than using a stable function.
f) A cursor provides an efficient way to traverse a large set of data, avoiding certain memory problems.
g) The use of a function in a query does not affect the way the query engine selects a plan for executing the query.
h) If a table has a table-level a complex invariant, that cannot be enforced through `CHECK`, `PRIMARY KEY`, or `FOREIGN KEY` constraints, it may be possible to implement the invariant check via a trigger on the table.
i) PostgreSQL allows triggers to be fired based on a pre-defined condition.
j) A trigger may cause itself to fire recursively.

3. (100pts) The goal of this exercise is to implement a trigger/function based solution to maintain a couple derived tables in a database. First, run the script below, which will create the "usual" tables as well as the ones where the derived data will reside.

```
CREATE TABLE customers (
  cId             SERIAL NOT NULL PRIMARY KEY,
  name            VARCHAR(100) NOT NULL
);

CREATE TABLE products (
  sku             INT NOT NULL PRIMARY KEY,
  product         VARCHAR(100) NOT NULL,
  unitPrice       NUMERIC(10,2) NOT NULL CHECK(unitPrice > 0),
  unitWeight      NUMERIC(10,2) NULL CHECK(unitWeight > 0)
);

CREATE TABLE orders (
  oId             SERIAL NOT NULL PRIMARY KEY,
  cId             INT NOT NULL REFERENCES customers(cId),
  orderDate       DATE NOT NULL CHECK(orderDate > '2010-01-01')
);

CREATE TABLE orderEntries (
  oId             INT NOT NULL REFERENCES orders(oId),
  sku             INT NOT NULL REFERENCES products(sku),
  qty             INT NOT NULL CHECK(qty > 0),
  CONSTRAINT pkOrderEntries PRIMARY KEY (oId, sku)
);

-- statistics for each customer
CREATE TABLE summary_customers (
  cId             INT NOT NULL,
  ordersPlaced    INT,            -- count of orders placed
  grossValue      NUMERIC(10,2),  -- sum of gross value over all order entries
  averageValue    NUMERIC(10,2),  -- average order gross value over all orders
  lastOrder       DATE            -- date of the last order
);

-- statistics for each product
CREATE TABLE summary_products (
  sku             INT NOT NULL,
  ordersPlaced    INT,            -- count of orders
  unitsOrdered    INT,            -- sum of qty over all orders
  grossValue      NUMERIC(10,2),  -- sum of gross value over all orders
  averageValue    NUMERIC(10,2),  -- average of gross value over all orders
  lastOrder       DATE            -- date of the last order
);

-- statistics for each order
CREATE TABLE summary_orders (
  oId             INT NOT NULL,
  entryCount      INT,            -- count of entries
  totalItems      INT,            -- sum of qty over all entries
  totalWeight     NUMERIC(10,2),  -- sum of qty*unitWeight over all entries
  grossValue      NUMERIC(10,2)   -- sum of qty*unitPrice over all entries
);
```

Create the necessary trigger functions and trigger definitions in order to maintain all summary tables up-to-date. The computations should be incremental. If you cannot compute something incrementally, please justify. Note that you will need to create triggers on both `orders` and `orderEntries`, as some summary information depends on `orders`, and some on `orderEntries`. You will get 30% of the points for `INSERT` triggers, 40% for `UPDATE` triggers, and 30% for `DELETE` triggers.