1 Stored procedures and triggers

So far, we have been concerned with the storage of data in databases. However, modern DBMSs also allow code to be stored in the database.

Sequences of queries and updates can be bundled together into a stored procedure or function which is stored in the database and is available for subsequent execution.

To illustrate, suppose a relational database contains two relations

\[ \text{Products}(\text{ProductID}, \text{QuantityInStock}, \text{ReorderLevel}) \]
\[ \text{NeedToReorder}(\text{ProductID}, \text{ReorderQuantity}) \]

The following stored procedure (written in Oracle’s PL/SQL language, which we will be looking at in more detail in an upcoming lab session) takes a product ID p and a quantity q (both numbers), finds out whether p is already on order, and if not it inserts the tuple (p,q) into the NeedToReorder relation:

```sql
CREATE PROCEDURE ReorderProduct(p,q NUMBER) AS
    DECLARE AlreadyReordered NUMBER;
    BEGIN
        SELECT COUNT(*) INTO AlreadyReordered
        FROM NeedToReorder
        WHERE ProductID = p;

        IF AlreadyReordered = 0 THEN
            INSERT INTO NeedToReorder VALUES (p,q)
        ENDIF;
    END;
```

Such stored procedures or functions can be called from within an SQL statement or from within another function or procedure. Stored functions can be used anywhere that SQL built-in functions can be used.

In addition to providing procedural abstraction, stored procedures/functions allow a block of SQL statements to be grouped into one statement that is submitted by the client application to the database server.

Generally, every SQL statement is executed individually by the database server. So grouping several statements together into one reduces client/server communication overheads and also network overheads if the client application and the database server are running on different machines. This has the potential to give considerable performance gains for applications.

The SQL/PSM (Persistent Stored Modules) standard specifies a language for defining stored procedures and functions. Its major elements include CREATE PROCEDURE and CREATE FUNC-
TION statements. Within the definitions of such procedures/functions, one can declare variables and use branching statements (IF ... THEN ... ELSEIF ... ELSE ... ENDIF), SQL statements, loops (LOOP ... END LOOP, WHILE ... DO ... END WHILE, REPEAT ... UNTIL ... END REPEAT). Similar features are provided by all modern commercial DBMSs, but in general not conforming precisely with the SQL/PSM syntax e.g. Oracle's PL/SQL language.

In modern DBMSs it is also possible to define triggers which will automatically execute some code if a specified event occurs in the database.

For example, the following trigger (written in Oracle's trigger syntax) is executed after any update of the QuantityInStock or ReorderLevel attributes of the Products relation.

It invokes the procedure ReorderProduct above for each product where the quantity in stock is now less than the reorder level:

```
CREATE TRIGGER ReorderTrigger
AFTER UPDATE OF QuantityInStock OR ReorderLevel
ON Products
FOR EACH ROW
WHEN (new.QuantityInStock < new.ReorderLevel)
BEGIN
    CALL ReorderProduct(:new.ProductID,:new.ReorderLevel);
END;
```

(Variables declared in a PL/SQL host environment - such as SQL*Plus - and passed to PL/SQL as arguments must be prefixed with a colon.)

The following, alternative, trigger would be executed before any update of the QuantityInStock or ReorderLevel attributes of the Products relation. It would print out a warning message for any record for which the quantity in stock would have fallen below the reorder level and would reinstate the old values of these two attributes:

```
CREATE OR REPLACE TRIGGER ReorderTrigger
BEFORE UPDATE OF QuantityInStock OR ReorderLevel
ON Products
FOR EACH ROW
WHEN (new.QuantityInStock < new.ReorderLevel)
BEGIN
    DBMS_OUTPUT.PUT_LINE('quantity in stock below reorder level for product ' || :new.ProductID) ;
    DBMS_OUTPUT.PUT_LINE(' original values reinstated') ;
    :new.QuantityInStock := :old.QuantityInStock ;
END;
```
1.1 Active Databases

Support of triggers turns databases from being passive to being active:

Passive databases execute transactions and queries that are explicitly submitted by users or applications.

Active databases can automatically react to occurrences of events and carry out appropriate actions.

Users specify the events to react to and the actions to carry out, by defining a set of triggers. A trigger consists of:

- An event part — the event that activates the trigger.
- A condition part — which is evaluated when the triggered is activated.
- An action part — which is executed if the condition is true.

So triggers are also known as event-condition-action rules (ECA rules).

The kinds of events that are detectable by typical RDBMSs are insertions, deletions, and updates on relations.

Some systems also support composite events, which are composed from primitive events using an ‘event language’ in which events can be combined by operators such as AND, OR, NOT, FOLLOWED BY.

Triggers can be executed BEFORE, AFTER or INSTEAD OF the event that activates them.

Triggers can be:

- statement-level — the action part executes just once, provided the condition is true; or
- row-level — the action part executes for each row that was inserted/deleted/updated, and for which the condition is true.

The two versions of ReorderTrigger in Section 1 above are row-level triggers.

Here is an example of a statement-level trigger (written in the syntax specified by the SQL standard), assuming a relation Students(studentId, name, address):

```sql
CREATE TRIGGER overRecruited
AFTER INSERT ON Students
REFERENCING NEW TABLE AS NewStudents
    OLD TABLE AS OldStudents
FOR EACH STATEMENT
WHEN (20000 < (SELECT COUNT(*)
    FROM NewStudents))
DELETE FROM Students
WHERE (studentId,name,address) IN NewStudents
    AND (studentId,name,address) NOT IN OldStudents;
```
INSERT INTO WaitingList (NewStudents EXCEPT OldStudents);

Note the use of NEW TABLE and OLD TABLE, and their aliases

The execution of one trigger can activate other triggers. This ‘cascade’ of activations continues until no more triggers are activated. In current commercial DBMSs there is a predefined limit on the number of such recursive activations, and if it is exceeded the current transaction is rolled back.

Much research has focussed on developing more sophisticated methods of detecting the termination properties of triggers, both statically (when triggers are defined) and dynamically (as triggers execute).

Applications of triggers include:

- detecting violation of complex integrity constraints and undertaking repair operations;
- enforcing complex authorisation restrictions;
- maintaining materialised views and replicas;
- logging particular events for auditing and security purposes;
- gathering statistics about database usage;
- notifying users or DBAs of the occurrence of particular database events.

2 Deductive Databases

Before SQL99, SQL did not support recursively defined relations.

For example, if we have a stored table assembly with attributes Part, Subpart and Quantity, recording the immediate subparts of parts, then it is not possible to write a (pre-SQL99) SQL query that returns all the components of a part unless the maximum depth of the part hierarchy is known, because we don’t know how many times to join the assembly table with itself.

This limitation of SQL has led since the 1980s to research into deductive databases. These are databases which do support recursively defined derived relations. This research in turn led to the addition of recursion into SQL99.

The WITH statement in SQL99 allows the definition of derived relations:

\[ \text{WITH } r \text{ AS } d \ q \]

where \( r \) is the scheme of the derived relation, \( d \) is an SQL query defining the contents of \( r \), and \( q \) is an SQL query using \( r \).

If \( r \) appears within \( d \) (i.e. \( r \) is derived recursively), then the keyword \texttt{RECURSIVE} is required after WITH.

To illustrate, given a stored table assembly(Part,Subpart,Quantity), we can find all the parts of which P1 is a component as follows:
WITH RECURSIVE component(Part,Comp) AS
  (SELECT assembly.Part, assembly.Subpart
   FROM assembly)
  UNION
  (SELECT assembly.Part component.Comp
   FROM assembly, component
   WHERE assembly.Subpart = component.Part)
SELECT * FROM component WHERE Comp='P1'

Note that SQL's WITH introduces a relation which is available for use only locally within this statement — component does not become part of the database schema.

Only stratified definitions (see Section 4.3) are permitted in SQL99, both with respect to negation (e.g. EXCEPT, NOT IN, NOT EXISTS) and with respect to aggregation functions.

3 More on Deductive Databases

The bulk of research into deductive databases has used a language called Datalog to specify inference rules — Datalog is a subset of Prolog.

Returning to the example of the assembly(Part,Subpart,Quantity) table, we can specify as follows in Datalog a relation component that contains all pairs (P,S) such that S is a component of P, at any level in the parts hierarchy:

component(P,S) :- assembly(P,S,Q)
component(P,S) :- assembly(P,S1,Q), component(S1,S)

A Datalog rule has its antecedents on its right-hand side (RHS), and its consequent on its left-hand side (LHS). Any variable that appears in the consequent must also appear in the argument list of a (non-negated) predicate in the antecedent.

The first rule above states that, for every tuple (P,S,Q) in the assembly relation, we can infer that there is a tuple (P,S) in the component relation.

The second rule above states that, for every pair of tuples (P,S1,Q) in the assembly relation and (S1,S) in the component relation, we can infer that there is a tuple (P,S) in the component table.

Note that the above is a recursive definition of the derived relation component since the identifier component appears both on the LHS and the RHS of the second rule.

Other terms used for a stored relation are base relation or extensional relation; a derived relation is also known as an intentional relation.

A general method for evaluating derived relations in Datalog is as follows:

1. set the derived relation to be the empty set;
2. evaluate the rule(s) defining the relation using the current value of the derived relation in the RHS of the rule(s)
3. If there is any change to the derived relation return to step 2, else stop.

For recursive definitions, this is known as computing the least fixpoint.

To illustrate, suppose the assembly base relation is as follows:

<table>
<thead>
<tr>
<th>assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>P5 P2 3</td>
</tr>
<tr>
<td>P5 P7 2</td>
</tr>
<tr>
<td>P2 P3 1</td>
</tr>
<tr>
<td>P2 P4 2</td>
</tr>
<tr>
<td>P7 P1 4</td>
</tr>
<tr>
<td>P1 P6 3</td>
</tr>
<tr>
<td>P1 P8 2</td>
</tr>
<tr>
<td>P1 P9 7</td>
</tr>
</tbody>
</table>

Then the first iteration of the evaluation procedure gives this as the definition of the component relation:

<table>
<thead>
<tr>
<th>component_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>P5 P2</td>
</tr>
<tr>
<td>P5 P7</td>
</tr>
<tr>
<td>P2 P3</td>
</tr>
<tr>
<td>P2 P4</td>
</tr>
<tr>
<td>P7 P1</td>
</tr>
<tr>
<td>P1 P6</td>
</tr>
<tr>
<td>P1 P8</td>
</tr>
<tr>
<td>P1 P9</td>
</tr>
</tbody>
</table>

The second iteration gives:

<table>
<thead>
<tr>
<th>component_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>P5 P2</td>
</tr>
<tr>
<td>P5 P7</td>
</tr>
<tr>
<td>P2 P3</td>
</tr>
<tr>
<td>P2 P4</td>
</tr>
<tr>
<td>P7 P1</td>
</tr>
<tr>
<td>P1 P6</td>
</tr>
<tr>
<td>P1 P8</td>
</tr>
<tr>
<td>P1 P9</td>
</tr>
</tbody>
</table>

The third iteration gives:
The fourth iteration gives the same relation, and the evaluation stops.

3.1 Optimisation

The above evaluation method is rather naive as it performs redundant computations, and improvements are possible. One optimisation is to use semi-naive evaluation where only the new tuples inferred in each iteration are combined with the existing tuples.

Consider again these rules:

\[
\text{component}(P,S) :\text{- assembly}(P,S,Q)
\]
\[
\text{component}(P,S) :\text{- assembly}(P,S1,Q), \text{component}(S1,S)
\]

and the assembly base table as given earlier.

Initially, \text{component} is empty, so we apply just the first rule to compute the first increment to \text{component}, \text{delta}_1:

\[
\text{delta}_1(P,S) :\text{- assembly}(P,S,Q)
\]

Giving:
Thereafter, we use the second rule to compute successive increments to component, as follows:

\[
\text{delta}_i+1(P,S) := \text{assembly}(P,S_1,Q), \text{delta}_i(S_1,S)
\]

Applying this rule for the first time, gives

\[
\begin{array}{|c|c|}
\hline
\text{delta}_2 & \\
\hline
P5 & P3 \\
P5 & P4 \\
P5 & P1 \\
P7 & P6 \\
P7 & P8 \\
P7 & P9 \\
\hline
\end{array}
\]

Applying this rule for the second time, gives

\[
\begin{array}{|c|c|}
\hline
\text{delta}_3 & \\
\hline
P5 & P6 \\
P5 & P8 \\
P5 & P9 \\
\hline
\end{array}
\]

Applying this rule for the third time gives \(\text{delta}_4 = \{\}\).

The evaluation therefore ends, giving \(\text{component} = \text{delta}_1 \cup \text{delta}_2 \cup \text{delta}_3\), which equals:
3.2 Magic Sets (optional)

Another optimisation is known as **Magic Sets** and is useful for pushing selection conditions into a recursive definition to reduce the amount of computation — see Ramakrishnan and Gerhke 24.5.2 - 25.5.3 (optional reading).

For example, suppose we have `component` relation defined as follows:

\[
\text{component}(P, S) :- \text{assembly}(P, S, Q) \\
\text{component}(P, S) :- \text{assembly}(P, S_1, Q), \text{component}(S_1, S)
\]

The following Datalog queries respectively return: the whole component relation; the parts of which P3 is a component; the components of P1; and whether P3 is a component of P1:

\[
\text{component}(P, S); \\
\text{component}(P, 'P3'); \\
\text{component}('P1', S); \\
\text{component}('P1', 'P3');
\]

The Magic Sets optimisation approach would generate 4 different versions of the definition of `component`, each version optimised for a different combination of known/unknown information about the super-component and the sub-component.

The terms **bound** and **free** are used to indicate a known or unknown argument to a derived relation, respectively.

Here are the four definitions (which can be automatically generated from the original definition above):
Known super-component:

\[
\text{component\_bf}(P, S) \leftarrow \text{assembly}(P, S, Q) \\
\text{component\_bf}(P, S) \leftarrow \text{assembly}(P, S_1, Q), \text{component\_bf}(S_1, S)
\]

Known sub-component:

\[
\text{component\_fb}(P, S) \leftarrow \text{assembly}(P, S, Q) \\
\text{component\_fb}(P, S) \leftarrow \text{assembly}(P, S_1, Q), \text{component\_bb}(S_1, S)
\]

Known super-component and sub-component:

\[
\text{component\_bb}(P, S) \leftarrow \text{assembly}(P, S, Q) \\
\text{component\_bb}(P, S) \leftarrow \text{assembly}(P, S_1, Q), \text{component\_bb}(S_1, S)
\]

Unknown super-component and sub-component:

\[
\text{component\_ff}(P, S) \leftarrow \text{assembly}(P, S, Q) \\
\text{component\_ff}(P, S) \leftarrow \text{assembly}(P, S_1, Q), \text{component\_bf}(S_1, S)
\]

The 4 queries above can then be optimised by using the appropriate definition:

\[
\text{component\_ff}(P, S); \\
\text{component\_fb}(P, 'P3'); \\
\text{component\_bf}('P1', S); \\
\text{component\_bb}('P1', 'P3');
\]

### 3.3 Left, right and nonlinear recursion (optional)

The above definition of the derived relation \text{component} is called \textbf{right-recursive} since the base relation \text{assembly} appears first within the second rule:

\[
\text{component}(P, S) \leftarrow \text{assembly}(P, S, Q) \\
\text{component}(P, S) \leftarrow \text{component}(P, S_1), \text{assembly}(S_1, S, Q), \text{component}(S_1, S)
\]

The following \textbf{left-recursive} definition would give the same answer [exercise for the reader]:

\[
\text{component}(P, S) \leftarrow \text{assembly}(P, S, Q) \\
\text{component}(P, S) \leftarrow \text{component}(P, S_1), \text{assembly}(S_1, S, Q)
\]

Right- and left-recursion are both examples of \textbf{linear} recursion, where the derived relation appears only once in the RHS of the definition. \textbf{Nonlinear} recursion is also possible. For example, this definition gives the same answer as the two above definitions [exercise for the reader]:

\[
\text{component}(P, S) \leftarrow \text{assembly}(P, S, Q) \\
\text{component}(P, S) \leftarrow \text{component}(P, S_1), \text{component}(S_1, S)
\]
3.4 Negation in Recursive Definitions (optional)

Introducing negation into recursive definitions can cause problems. For example, consider the following definitions:

\[
\text{big}(P) \ :- \ \text{assembly}(P,S,Q), \ \text{NOT} \ \text{small}(P)
\]
\[
\text{small}(P) \ :- \ \text{assembly}(P,S,Q), \ \text{NOT} \ \text{big} \ (P)
\]

What is the value of big and small ?

Well, following the evaluation method described above,

\[
\text{big}_0 = \text{small}_0 = \{\}
\]
\[
\text{big}_1 = \text{small}_1 = \text{assembly}
\]
\[
\text{big}_2 = \text{small}_2 = \{\}
\]
\[
\text{big}_3 = \text{small}_3 = \text{assembly}
\]

etc. and the computation fails to terminate. This is because there is no single least fixpoint answer for the above definitions.

The solution to this problem is to disallow ambiguous definitions such as that above by requiring that definitions are stratified:

Construct a graph (known as the dependency graph) whose nodes are the derived relations. Draw an arc from R to S if S appears in the definition of R. Label this arc with a '+' if S appears non-negated and label it with a '-' if S appears negated.

The set of definitions is stratified if there no cycles labelled with '-' on any of their arcs.

The relations defined by a stratified set of definitions can be divided into strata 0, 1, 2 ... such that the relations in stratum $i$ depend positively only on relations in strata $j \leq i$ and depend negatively only on relations in strata $j < i$.

A stratified set of definitions can be evaluated stratum by stratum to give an unambiguous result.

Aggregation functions can pose similar problems to negation in recursive definitions, because the incremental computation of an aggregation function invalidates the previous iteration’s computation.

For example, suppose we have a single-column base table p, containing three tuples 1, 2, 3. And a new derived relation, r, is defined by the following (non-stratified) rules:

\[
\text{r}(X) \ :- \ \text{p}(X)
\]
\[
\text{r}(X) \ :- \ X \ \text{is sum}(r)
\]

Then: $r_0 = \{\}
\]
\[
\text{r}_1 = \{1, 2, 3, 0\}
\]
\[
\text{r}_2 = \{1, 2, 3, 6\}
\]
\[
\text{r}_3 = \{1, 2, 3, 12\}
\]

etc., and the computation never reaches a fixpoint.

In general, derived relations need to be monotonic i.e. only new tuples can be added at each round of their evaluation and tuples computed in a previous round cannot be made invalid.
3.5 Influence

The theoretical foundations of Datalog, together with the semi-native and Magic Sets techniques pioneered in the context of that language, underlie the evaluation of recursively-defined relations and queries in modern-day DBMSs, as well as in specialised rule-based systems for reasoning with data in a variety of settings e.g. systems that combine RDFS/OWL reasoning with relational data management or with graph data.

Homework 3 (optional)

Consider a database for the London underground and bus network, consisting of two relations:

\[
\text{tube}(\text{Station,NextStation,TubeLine}) \\
\text{bus}(\text{Station,NextStation,BusLine})
\]

With contents:

<table>
<thead>
<tr>
<th>tube</th>
</tr>
</thead>
<tbody>
<tr>
<td>finsburyPark</td>
</tr>
<tr>
<td>manorHouse</td>
</tr>
<tr>
<td>manorHouse</td>
</tr>
<tr>
<td>turnpikeLane</td>
</tr>
<tr>
<td>turnpikeLane</td>
</tr>
<tr>
<td>woodGreen</td>
</tr>
<tr>
<td>leytonstone</td>
</tr>
<tr>
<td>snaresbrook</td>
</tr>
<tr>
<td>snaresbrook</td>
</tr>
<tr>
<td>southWoodford</td>
</tr>
<tr>
<td>southWoodford</td>
</tr>
<tr>
<td>woodford</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bus</th>
</tr>
</thead>
<tbody>
<tr>
<td>manorHouse</td>
</tr>
<tr>
<td>turnpikeLane</td>
</tr>
<tr>
<td>turnpikeLane</td>
</tr>
<tr>
<td>harringay</td>
</tr>
<tr>
<td>harringay</td>
</tr>
<tr>
<td>tottenham</td>
</tr>
<tr>
<td>tottenham</td>
</tr>
<tr>
<td>walthamstow</td>
</tr>
<tr>
<td>walthamstow</td>
</tr>
<tr>
<td>snaresbrook</td>
</tr>
</tbody>
</table>

Diagrammatically, we can present this route map as shown in the figure.
finsburyPark
manorHouse
turnpikeLane
woodGreen
leytonstone
snaresbrook
southWoodford
woodford
harringay
tottenham
snaresbrook
leytonstone
Write derived relation definitions in Datalog that give the following:

1. The pairs of stations A,B such that B is reachable from A by tube.
2. The pairs of stations A,B such that B is reachable from A by bus.
3. The pairs of stations A,B such that B is reachable from A by tube but not by bus.
4. The pairs of stations A,B such that B is reachable from A by some combination of tube or bus.
5. The pairs of stations A,B such that B is reachable from A by some combination of tube or bus, but not by tube or bus alone.