WEEK04 LECTURE

EXTENDING SQL

LIMITATIONS OF BASIC SQL

What we have seen of SQL so far:

- Dara definition language (create table (…))
- Constraints (domain, key, referential integrity)
- Query language (select … from … where)
- Views (give names to SQL queries)

NEW DATA TYPES

SQL data definition language provides:

- Atomic types: integer, float, character, Boolean
- Ability to define tuple types (create table)
    - Create domain Positive as integer check (value>0);
    - Create type Rating as enum ('poor', 'ok', 'excellent');
        - An ENUM is a string object with a value chosen from a list of permitted values that are enumerated explicitly in the column specification at table creation time.
        - An enumeration value must be a quoted string literal
    - Create type Pair as (x integer, y integer);

NEW FUNCTIONS

SQL provides for new functions via stored procedures

```
Create function f (arg1 type1, arg2 type2, …) returns type
As $$          function body          $$ language  language [mode]
```

Possible modes:

1. Immutable : does not access database (fast)
2. Stable : does not modify the database
3. Volatile : may change the database (slow, default)

EXERCISE: FUNCTIONS ON (SETOF) INTEGERS

QUERIES

ADVANCED QUERY TYPES

- Many specialized types of query have been identified
- We have seen: select/project/join, aggregation, grouping
- Many modern queries (e.g. skyline) come from OLAP
- Two important standard query types:
    1. Recursive: e.g. to manage hierarchies graphs
    2. Window: e.g. to spread group by summaries

WINDOW FUNCTIONS

- Group-by allows us to
    - Summarize a set of tuples
    - That have common values for a set of attributes

- E.g. average mark for each student
  Select student, avg(mark) from CourseEnrolments
  Group by students;
- Produces a single summary tuple for each group.

- Window functions allow us to
  - Compute summary values for a group
  - Append the summary value to each tuple in the group

Window functions operate on asset or rows and return a single value for each row from the underlying query. The term window describes the set of rows on which the function operates. A window function uses values from the rows in a window to calculate the returned values.
*** the OVER() clause has the following capabilities:*
1- defines window partitions to form groups or rows (partition by)
2- orders rows with a partition (order by)

- E.g. attach student's average mark to each enrolment
  Select *, avg(mark)
  Over (partition by student) from CourseEnrolments;

```
select student,avg(mark) ... group by student

 student  |  avg
----------+-------
 46000936 | 64.75
 46001128 | 73.50
```

```
select *,avg(mark) over (partition by student) ...

 student  | course | mark | grade | stueval |  avg
----------+--------+------+-------+---------+-------
 46000936 |  11971 |   68 | CR    |       3 | 64.75
 46000936 |  12937 |   63 | PS    |       3 | 64.75
 46000936 |  12045 |   71 | CR    |       4 | 64.75
 46000936 |  11507 |   57 | PS    |       2 | 64.75
 46001128 |  12932 |   73 | CR    |       3 | 73.50
 46001128 |  13498 |   74 | CR    |       5 | 73.50
 46001128 |  11909 |   79 | DN    |       4 | 73.50
 46001128 |  12118 |   68 | CR    |       4 | 73.50
```

WITH QUERIES

- We often break a complex query up into views: e.g.
  Create view V as select a,b,c from … where …;
- WITH allows scoped/temporary views
- View v and w
  - Only exist while this query is evaluated
  - Are not accessible in any other context

... WITH Queries

WITH allows scoped/temporary views, e.g.

```
with V as (select a,b,c from ... where ...),
     W as (select d,e from ... where ...)
select V.a as x, V.b as y, W.e as z
from   V join W on (v.c = W.d);
```

The views V and W

- only exist while this query is evaluated
- are not accessible in any other context

V and W are also called "common table expressions" (CTEs)

... WITH Queries

Note that named subqueries achieve the same effect:

```
select V.a as x, V.b as y, W.e as z
from   (select a,b,c from ... where ...) as V,
       (select d,e from ... where ...) as W
where  V.c = W.d;
```

For this purpose, WITH is a syntactic convenience.

However, WITH also provides recursive queries.

RECURSIVE QUERIES

Recursive queries are defined as:

```
with recursive T(a1, a2, ...) as
(
    non-recursive select
  union
    recursive select involving T
)
select ... from T where ...
```

T(a_1, a_2, ...) is a recursively-defined view.

```
-- res, work, tmp are all temporary tables
res = result of non-recursive query
work = res
while (work is not empty) {
   -- using work as the value for T ...
   tmp = result of recursive query
   res = res + tmp
   work = tmp
}
return res
```

*** example: generate sum of first 100 integers:

```
with recursive nums(n) as (
    select 1
  union
    select n+1 from nums where n < 100
)
select sum(n) from nums;
```

AGREEGATES

- Aggregates reduce a collection of values into a single result.
- Examples: count(Tuples), sum(Numbers)...
- The action of an aggregate function can be viewed as:

```
AggState = initial state
for each item V {

    AggState = newState(AggState,
V)
}
return final(AggState)
```

- Aggregates are commonly used with GROUP BY
- In the context they "summarize" each group

Example:

```
R
 a | b | c
---+---+---
 1 | 2 | x
 1 | 3 | y
 2 | 2 | z
 2 | 1 | a
 2 | 3 | b
```

```
select a,sum(b),count(*)
from R group by a

 a | sum | count
---+-----+-------
 1 |   5 |     2
 2 |   6 |     3
```

USER-DEFINED AGGREGATES

- SQL standard does not specify user-defined aggregates.
- But PostgreSQL provides a mechanism for defining them.
- To define a new aggregate, first need to apply:
  - BaseType: type of input values
  - StateType: type of intermediate states
  - State mapping function: sfunc(state, value) -> newState
  - (optionally) an initial state of value (defaults to null)
  - (optionally) final function: ffunc(star)->result

```
CREATE AGGREGATE AggName(BaseType) (
    sfunc     = NewStateFunction,
    stype     = StateType,
    initcond  = InitialValue,
    finalfunc = FinalResFunction,
    sortop    = OrderingOperator
);
```

Example: defining the count aggregate (roughly)

```
create aggregate myCount(anyelement) (
    stype    = int,    -- the accumulator type
    initcond = 0,      -- initial accumulator value
    sfunc    = oneMore -- increment function
);

create function
    oneMore(sum int, x anyelement) returns int
as $$
begin return sum + 1; end;
$$ language plpgsql;
```

Example: sum2 sums two columns of integers

```
create type IntPair as (x int, y int);

create function
    AddPair(sum int, p IntPair) returns int
as $$
begin return p.x + p.y + sum; end;
$$ language plpgsql;

create aggregate sum2(IntPair) (
    stype     = int,
    initcond  = 0,
    sfunc     = AddPair
);
```

CONSTRAINTS

- Column and table constraints ensure validity of one table
- RI constraints ensure connections between tables are valid
- However, specifying validity of entire database often requires constraints involving multiple tables.

```
for all Branches b
   b.assets == (select sum(acct.balance)
                from   Accounts acct
                where  acct.branch = b.location)
```

ASSERTIONS

- Assertions are schema-level constraints
  - Typically involving multiple tables
  - Expressing a condition that must hold at all times
  - Need to be checked on each change to relevant tables
  - If change would cause check to fail, reject change

```
CREATE ASSERTION name CHECK (condition)
```

- Example: #students in any UNSW course must be < 10000

```
Create assertion ClassSizeConstraint check (
        Not exists (
                Select c.id
                from Courses c, CourseEnrolments e
                where c.id = e.course
                group by c.id having count(e.student) > 9999
        )
)
```

TRIGGERS

- Triggers are
  - Procedures stored in the database
  - **Activated response to database event** (e.g. updates, inserted, deleted)
- Examples of uses for triggers:
  - Maintaining summary data
    - When table a maintain table b, when table b got updated, table a will be updated automatically
  - Checking schema-level constraints (assertions) on update
    - More efficiently than assertion
  - Performing multi-table updates (to maintain assertions)
    - Instead of checking assertion, maintain other than just check
- Triggers provide event-condition-action(ECA) programming:
  - An event activates the trigger
  - On activation the trigger checks a condition
  - If the condition holds, a procedure is executed (the action)
- Some typical variations on it:
  - Execute the action before, after or instead of the triggering event
    - Before you withdraw the money, you do something, afterwards you do something
  - Can refer to both old and new values of updated tuples
  - Can limit updates to a particular set of attributes
  - Perform action: for each modified tuple, once for all modified tuples
    - E.g. one update, can update all the tuples

The event can be insert, delete or update

FOR EACH ROW: after you do the updates, you do this

```
CREATE TRIGGER TriggerName
{AFTER|BEFORE}  Event1 [   OR    Event2 ….]
[ for each row ]
ON TableName
[ WHEN ( Condition )]
Block of procedural/SQL code
```

- ***riggers can be activated BEFORE or AFTER the event
- OLD doesn't exist for insertion, NEW doesn't exist for deletion
- If activated BEFORE, can affect the change that occurs:
    - NEW contains "proposed" value of changed tuple
    - Modifying NEW causes a different value to be placed in DB
- If activated AFTER, the effects of the event are visible
    - NEW contains the current value of the changed tuple
    - OLD contains the previous value of the changed tuple
    - Constraint-checking has been down for NEW
- Note!
    - OLD doesn't exist for INSERTION
    - NEW doesn't exist for DELETION

- Consider two triggers and an **INSERT** statement

```
Create trigger X before insert on T code1;
Create trigger Y after insert on T code2;
Insert into T values (a, b, c, …);
```

(when you run the last line, insert statement)

1. Execute Code1 for trigger X
    a. Because of this trigger is to check something before insert T to code1 so we need to access code1 table
2. Code has access to (a, b, c, …) via NEW (contains all the new values)
    a. NEW, which is a RECORD datatype, which stores the information of each row of the whole table NEW now stores a, b, c, … from T
3. Code typically checks the values of a, b, c, …
    a. The trigger contains constraints, it checks every row in the old table to see if they can do the next action, INSERT
4. Code can modify values a, b, c .. in NEW (change the value/ update)
    a. After checking all the values via NEW, to make sure it reach all the requirement/constraints in the trigger X, we can insert the values, which is like modifying NEW, so old NEW becomes new NEW
5. DBMS does constraint checking as if NEW is inserted (db assume that it's the new db then check)
6. If failed any checking, abort insertion and rollback
7. Execute code2 for trigger Y
    a. Active trigger Y after the INSERT on code2 table
8. Code has access to final version of tuple via NEW
    a. So the NEW stores all the updated/inserted information from the code2 table, which contains the record of each row
9. Code typically does final checking or modifies other tables in database to ensure constraints are satisfied

- Consider two triggers and an **UPDATE** statement

```
Create trigger X before update on T code1;
Create trigger Y after update on T code2;
Update T set b=j, c=k where a=m;
```

1. Execute code1 for trigger X
    a. Trigger X need to be active before the update of code1, after everything reach the constraint, it can run the UPDATE
2. Code has access to the current version of tuple via OLD
    a. Now all the information/rows from the code1 are stores in the OLD record
3. Code has access to updated version of tuple via NEW
    a. The record or the set that we want to change old to update
4. Code typically checks new values of b, c, ..

5. Code can modify values of a, b, c, … in NEW
6. Do constraint checking as if NEW has replaced OLD
7. If fails any checking, abort update and rollback
8. Execute Code2 for trigger Y
    a. After UPDATE checking
9. Code has access to final version of tuple via NEW
10. Code typically does final checking or modifies other tables in database to ensure constraints are satisfied.

- Consider two triggers and **DELETE** statement

```
Create trigger X before delete on T code1;
Create trigger Y after delete on T code2;
Delete T where a=m;
```

1. Execute code1 for trigger X
2. Code has access to (a, b, c, …) via OLD
3. Code typically checks the values of a, b, c, …
4. DBMS does constraint checking as if OLD is removed
5. If fails any checking, abort deletion (restore OLD)
6. Execute Code2 for trigger Y
7. Code has access to ABOUT TO BE DELETED tuple via OLD
8. Code typically does final checking or modifies other tables in database to ensure constraints are satisfied.

TRIGGERS IN POSTGRESQL

- PostgreSQL triggers provide a mechanism for
    - INSERT, DELETE or UPDATE events
    - To automatically activate PlpgSQL functions
- Syntax for PostgreSQL trigger definition

```
CREATE TRIGGER triggername
(BEFORE or AFTER) EVENT1 (or EVENT2 ...)
ON tablename
[WHEN (CONDITION)]
FOR EACH (ROW  or STATEMENT)
EXECUTE PROCEDURE functionname(...)
```

- There is no restriction on what code can go I function
- A BEFORE function must contain one of
    - RETURN OLD or RETURN NEW
- Depending on which version of the tuple is to be used
- If BEFORE trigger returns OLD, no change occurs
- If exception is raised in trigger function, no change occurs.