Week 04 Lectures

Problem Solving in SQL

1/42

Developing SQL queries ...

- relate required data to *attributes* in schema
- identify which *tables* contain these attributes
- combine data from relevant tables (*FROM*, *JOIN*)
- specify conditions to select relevant data (*WHERE*)
- [optional] define grouping attributes (*GROUP BY*)
- develop expressions to compute output values (*SELECT*)

Learn some query patterns and know when to apply them

... Problem Solving in SQL

2/42

Example: what is the most expensive beer?

- what is the highest price for a beer
- which beers are sold for this price

As SQL:

```
create view maxPrice as select max(price) from Sells;

create view highestPriceBeer as
select beer
from   Sells
where  price = (select * from maxPrice);
```

Pattern: what is the X of the Y with the maximum/minimum Z

E.g.   X = name,   Y = beer,   Z = price

... Problem Solving in SQL

3/42

Example: what is cheapest beer at each bar?

```
for each Bar {
   find cheapest price at the bar
   fine name of beer with this price
}
```

Needs *correlated subquery*

```
select s.bar, s.beer, s.price
from   Sells s
where  s.price = (select min(s1.price)
                  from   Sells s1
                  where  s1.bar = s.bar);
```

Views

4/42

A *view* associates a name with a query:

- **CREATE VIEW** *viewName* [**(** *attributes* **)**] **AS** *Query*

Each time the view is invoked (in a FROM clause):

- the *Query* is evaluated, yielding a set of tuples
- the set of tuples is used as the value of the view

A view can be treated as a "virtual table".

Views are useful for "packaging" a complex query to use in other queries.

cf. writing functions to package computations in programs

... Views

5/42

Previous example could have been solved as

```
create view Cheapest(bar,price)
as
select bar, min(price)  from Sells  group by bar
select s.bar, s.beer, s.price
from   Sells s
where  s.price
        = (select price from Cheapest where bar = s.bar);
```

Note:

- brown identifiers are local to subquery
- red identifiers are global to query

... Views

6/42

Views can be defined giving names to attributes

```
create view V(a, b, c)
as
select x, y, z from R where Condition
```

This is the same as

```
create view V
as
select x as a, y as b, z as c
from R where Condition
```

... Views

7/42

Views can be re-defined *in situ* using

```
create or replace view V(a,b,c) as Query
```

Restrictions:

- new view must have same number of attributes as old view
- attributes in new view must be same types as in old view

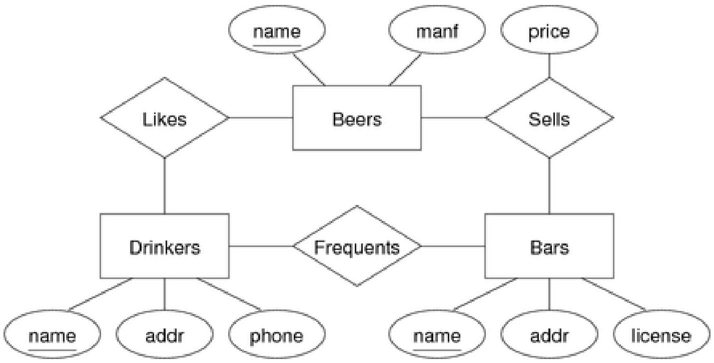
Otherwise

```
drop view V;
create view V(a,b,c) as Query
```

Exercise 1: More Queries on Beer Database

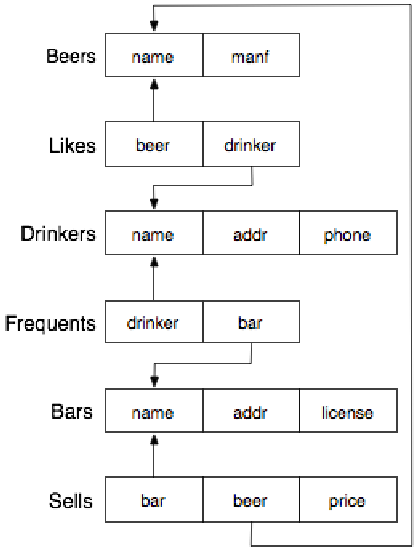
8/42

ER design for Beer database:



More queries on the Beer database:

15. Which bar is most popular? (Most drinkers)
16. Which bar is most expensive?
    (Maximum average price)
17. Which beers are sold at all bars?
18. Price of cheapest beer at each bar?
19. Name of cheapest beer at each bar?
20. How many drinkers are in each suburb?
21. How many bars in suburbs where drinkers live?
    (Must include suburbs with no bars)



Limitations of Basic SQL

10/42

What we have seen of SQL so far:

- data definition language (`create table(...)`)
- constraints (domain, key, referential integrity)
- query language (`select...from...where...`)
- views (give names to SQL queries)

This is not sufficient to write complete applications.

More *extensibility* and *programmability* are needed.

Extending SQL

11/42

Ways in which standard SQL might be extended:

- new data types (incl. constraints, I/O, indexes, ...)
- object-orientation
- more powerful constraint checking
- packaging/parameterizing queries
- more functions/aggregates for use in queries
- event-based triggered actions

All are required to assist in application development.

Programming with SQL

SQL as a Programming Language

13/42

SQL is a powerful language for manipulating relational data.

But it is *not* a powerful *programming language*.

At some point in developing complete database applications

- we need to implement user interactions
- we need to control sequences of database operations
- we need to process query results in complex ways

and SQL cannot do any of these.

SQL cannot even do something as simple as factorial!

Ok ... so PostgreSQL added a factorial operator ... but it's non-standard.

What's wrong with SQL?

14/42

Consider the problem of withdrawal from a bank account:

If a bank customer attempts to withdraw more funds than they have in their account, then indicate "Insufficient Funds", otherwise update the account

An attempt to implement this in SQL:

```
select 'Insufficient Funds'
from   Accounts
where  acctNo = AcctNum and balance < Amount;
update Accounts
set    balance = balance - Amount
where  acctNo = AcctNum and balance >= Amount;
select 'New balance: '||balance
from   Accounts
where  acctNo = AcctNum;
```

... What's wrong with SQL?

15/42

Two possible evaluation scenarios:

- displays "Insufficient Funds", `UPDATE` has no effect, displays unchanged balance
- `UPDATE` occurs as required, displays changed balance

Some problems:

- SQL doesn't allow parameterisation (e.g. *AcctNum*)
- always attempts `UPDATE`, even when it knows it's invalid
- need to evaluate `(balance <` *Amount*`)` test twice
- always displays balance, even when not changed

To accurately express the "business logic", we need facilities like conditional execution and parameter passing.

Database Programming

16/42

Database programming requires a combination of

- manipulation of data in DB (via SQL)
- conventional programming (via procedural code)

This combination is realised in a number of ways:

- passing SQL commands via a "call-level" interface
  (prog lang is decoupled from DBMS; most flexible; e.g. Java/JDBC, PHP)
- embedding SQL into augmented programming languages
  (requires pre-processor for language; typically DBMS-specific; e.g. SQL/C)
- special-purpose programming languages in the DBMS
  (closely integrated with DBMS; enable extensibility; e.g. PL/SQL, PLpgSQL)

... Database Programming

17/42

Combining SQL and procedural code solves the "withdrawal" problem:

```
create function
    withdraw(acctNum text, amount integer) returns text
declare bal integer;
begin
    set bal = (select balance
               from   Accounts
               where  acctNo = acctNum);
    if (bal < amount) then
        return 'Insufficient Funds';
    else
        update Accounts
        set    balance = balance - amount
        where  acctNo = acctNum;
        set bal = (select balance
                   from   Accounts
                   where  acctNo = acctNum);
        return 'New Balance: ' || bal;
    end if
end;
```

(This example is actually a stored procedure, using SQL/PSM syntax)

PostgreSQL Stored Procedures

18/42

PostgreSQL syntax for defining stored *functions*:

```
CREATE OR REPLACE FUNCTION
   funcName(arg1, arg2, ....) RETURNS retType
AS $$
```

```
String containing function definition
$$ LANGUAGE funcDefLanguage;
```

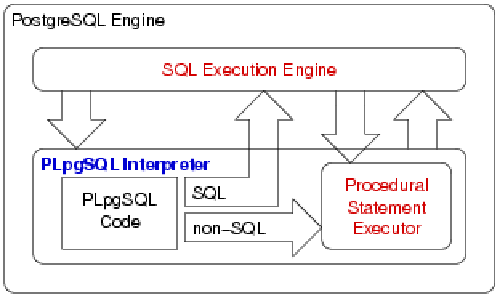
Notes:

- *arg$_i$* consists of *name type*
- `$$` ... `$$` are just another type of string quote
- function definition languages: SQL, PLpgSQL, Python, ...

... PostgreSQL Stored Procedures

The PLpgSQL interpreter

- executes procedural code and manages variables
- calls PostgreSQL engine to evaluate SQL statements



Function Return Types

A PostgreSQL function can return a value which is

- `void` (i.e. no return value)
- an atomic data type (e.g. `integer, text, ...`)
- a tuple (e.g. table record type or tuple type)
- a set of atomic values (like a table column)
- a set of tuples (i.e. a table)

A function returning a set of values is similar to a view.

... Function Return Types

Examples of different function return types:

```
create function factorial(integer) returns integer ...
create function EmployeeOfMonth(date) returns Employee ...
create function allSalaries() returns setof float ...
create function OlderEmployees() returns setof Employee ...
```

Different kinds of functions are invoked in different ways:

```
select factorial();  -- returns one integer
select EmployeeOfMonth('2008-04-01');  -- returns (x,y,z)
select * from EmployeeOfMonth('2008-04-01'); -- one-row table
select * from allSalaries();  -- single-column table
select * from OlderEmployees();  -- subset of Employees
```

SQL Functions

PostgreSQL allows functions to be defined in SQL

```
CREATE OR REPLACE
   funcName(arg1type, arg2type, ....)
   RETURNS rettype
AS $$
   SQL statements
$$ LANGUAGE sql;
```

Within the function, arguments are accessed as `$1`, `$2`, ...

Return value: result of the last SQL statement.

rettype can be any PostgreSQL data type (incl tuples,tables).

Function returning a table: `returns setof` *TupleType*

Details: PostgreSQL Documentation, Section 38.5

... SQL Functions

23/42

Examples:

```
-- max price of specified beer
create or replace function
    maxPrice(text) returns float
as $$
select max(price) from Sells where beer = $1;
$$ language sql;

-- usage examples
select maxPrice('New');
 maxprice
----------
      2.8

select bar,price from sells
where beer='New' and price=maxPrice('New');
    bar     | price
------------+-------
 Marble Bar |   2.8
```

... SQL Functions

24/42

Examples:

```
-- set of Bars from specified suburb
create or replace function
    hotelsIn(text) returns setof Bars
as $$
select * from Bars where addr = $1;
$$ language sql;

-- usage examples
select * from hotelsIn('The Rocks');
      name        |    addr    | license
-----------------+-----------+---------
 Australia Hotel | The Rocks |  123456
 Lord Nelson     | The Rocks |  123888
```

Exercise 2: SQL Functions

25/42

Recall Q19 on Beer db: Name of cheapest beer at each bar?

```
create view Cheapest(bar, price)
as
select bar, min(price) from Sells group by bar;

select s.*
from   Sells s
where  s.price = (select price from Cheapest where bar = s.bar);
```

Re-implement it by defining an SQL function `LowestPriceAt(bar)`

```
select * from Sells where price = LowestPriceAt(bar);
```

Functions vs Views

26/42

A parameterless function behaves similar to a view

E.g.

```
create or replace view EmpList(name, addr)
as
select family||' '||given,
       street||', '||town
from   Employees;
```

which is used as

```
mydb=# select * from EmpList;
```

... Functions vs Views

27/42

Compared to its implementation as a function:

```
create type EmpRecord as (name text, addr text);

create or replace function
   EmpList() returns setof EmpRecord
as $$
select family||' '||given as name,
       street||', '||town as addr
from   Employees
$$ language sql;
```

which is used as

```
mydb=# select * from EmpList();
```

PLpgSQL

28/42

PLpgSQL = **P**rocedural **L**anguage extensions to **P**ost**g**re**SQL**

A PostgreSQL-specific language integrating features of:

- procedural programming and SQL programming

Provides a means for *extending DBMS functionality*, e.g.

- implementing constraint checking (triggered functions)
- complex query evaluation (e.g. recursive)
- complex computation of column values
- detailed control of displayed results

Details: PostgreSQL Documentation, Chapter 43

Defining PLpgSQL Functions

29/42

PLpgSQL functions are created (and inserted into db) via:

```
CREATE OR REPLACE
   funcName(param1, param2, ....)
   RETURNS rettype
AS $$
DECLARE
   variable declarations
BEGIN
   code for function
END;
$$ LANGUAGE plpgsql;
```

Note: the entire function body is a single SQL string.

Simple PLpgSQL Example

30/42

A function to return *2n*, for a given *n*

```
create or replace
   double(n integer) returns integer
as $$
declare
   res integer;
begin
   res := 2*n;
   return res;
end;
$$ language plpgsql;
```

or, more simply

```
... $$  begin  return 2*n;  end;  $$ ...
```

PLpgSQL Function Parameters

31/42

Example: new-style function ("a","b") → "a'b"

```
CREATE OR REPLACE FUNCTION
    cat(x text, y text) RETURNS text
AS $add$
DECLARE
    result text;     -- local variable
```

```
BEGIN
    result := x||''''||y;
    return result;
END;
$add$ LANGUAGE 'plpgsql';
```

Beware: never give parameters the same names as attributes.

One strategy: start all parameter names with an underscore.

... PLpgSQL Function Parameters

32/42

Example: old-style function ("a","b") → "a'b"

```
CREATE OR REPLACE FUNCTION
    cat(text, text) RETURNS text
AS '
DECLARE
    x alias for $1;  -- alias for parameter
    y alias for $2;  -- alias for parameter
    result text;     -- local variable
BEGIN
    result := x||''''''''||y;
    return result;
END;
' LANGUAGE 'plpgsql';
```

Example PLpgSQL function

33/42

Function which handles withdrawl of money from account and returns status message:

```
create function
    withdraw(acctNum text, amount integer) returns text
as $$
declare bal integer;
begin
    select balance into bal
    from   Accounts
    where  acctNo = acctNum;
    if (bal < amount) then
        return 'Insufficient Funds';
    else
        update Accounts
        set    balance = balance - amount
        where  acctNo = acctNum;
        select balance into bal
        from   Accounts
        where  acctNo = acctNum;
        return 'New Balance: ' || bal;
    end if;
end;
$$ language plpgsql;
```

PLpgSQL Gotchas

34/42

Some things to beware of:

- doesn't provide any i/o facilities (except `RAISE NOTICE`)
  - the aim is to build computations on tables that SQL alone can't do
- functions are not syntax-checked when loaded into DB
  - you don't find out about the syntax error until "run-time"
- error messages are sometimes not particularly helpful
- functions are defined as strings
  - change of "lexical scope" can sometimes be confusing
- giving params/variables the same names as attributes

Summary: debugging PLpgSQL can sometimes be tricky.

Data Types

35/42

PLpgSQL constants and variables can be defined using:

- standard SQL data types (`CHAR, DATE, NUMBER, ...`)
- user-defined PostgreSQL data types (e.g. `Point`)
- a special structured record type (`RECORD`)
- table-row types (e.g. `Branches%ROWTYPE`)
- types of existing variables (e.g. `Branches.location%TYPE`)

There is also a `CURSOR` type for interacting with SQL.

... Data Types

36/42

Variables can also be defined in terms of:

- the type of an existing variable or table column
- the type of an existing table row (implict `RECORD` type)

Examples:

```
quantity    INTEGER;
start_qty   quantity%TYPE;

employee    Employees%ROWTYPE;

name        Employees.name%TYPE;
```

Syntax/Control Structures

37/42

A standard assignment operator is available:

Assignment	*var* `:=` *expr* `SELECT` *expr* `INTO` *var*

Selection	`IF` C_1 `THEN` S_1
	`ELSIF` C_2 `THEN` S_2 `...`
	`ELSE` S `END IF`

Iteration	`LOOP` S `END LOOP`
	`WHILE` C `LOOP` S `END LOOP`
	`FOR` *rec_var* `IN` *Query* `LOOP ...`
	`FOR` *int_var* `IN` *lo..hi* `LOOP ...`

SELECT...INTO

38/42

Can capture query results via:

`SELECT`	$Exp_1, Exp_2, ..., Exp_n$
`INTO`	$Var_1, Var_2, ..., Var_n$
`FROM`	*TableList*
`WHERE`	*Condition* `...`

The semantics:

- execute the query as usual
- return "projection list" (Exp_1,Exp_2,...) as usual
- assign each Exp_i to corresponding Var_i

... SELECT...INTO

39/42

Assigning a simple value via `SELECT...INTO`:

```
-- cost is local var, price is attr
SELECT price INTO cost
FROM   StockList
WHERE  item = 'Cricket Bat';
cost := cost * (1+tax_rate);
total := total + cost;
```

The current PostgreSQL parser also allows this syntax:

```
SELECT INTO cost price
FROM   StockList
WHERE  item = 'Cricket Bat';
```

... SELECT...INTO

40/42

Assigning whole rows via `SELECT...INTO`:

```
DECLARE
   emp    Employees%ROWTYPE;
   eName  text;
   pay    real;
```

```
BEGIN
   SELECT * INTO emp
   FROM Employees WHERE id = 966543;
   eName := emp.name;
   ...
   SELECT name,salary INTO eName,pay
   FROM Employees WHERE id = 966543;
END;
```

... SELECT...INTO 41/42

In the case of a PLpgSQL statement like

```
select a into b from R where ...
```

If the selection returns no tuples

- the variable `b` gets the value `NULL`

If the selection returns multiple tuples

- the variable `b` gets the value from the first tuple

... SELECT...INTO 42/42

An alternative to check for "no data found"

Use the special variable `FOUND` ...

- local to each function, set false at start of function
- set true if a `SELECT` finds at least one tuple
- set true if `INSERT/DELETE/UPDATE` affects at least one tuple
- otherwise, remains as `FALSE`

Example of use:

```
select a into b from R where ...
if (not found) then
    -- handle case where no matching tuples b
```

Produced: 2 Mar 2020