COSC210

Database Mangement Sytems

Lab Session 2 - Final Topics on SQL Using the PostgreSQL DBMS

By Mitchell Welch

University of New England

Reading

- Chapters 6 and 7 from ***Fundamentals of Database Systems*** by Elmazri and Navathe

Summary

- House Keeping
- String Comparison Operations
- Arithmetic Operations
- Grouping and Aggregate Functions
- Views(Virtual Tables)

Before we Get Started...

- In order to run the examples presented through this practical session, you should import the database you created last week into a new database for this week.
- Create a backup of your prac_01 database using the pg_dump utility:

```
[mwelch8@turing ~]$ pg_dump mwelch8_prac_01 > ~/prac_01.sql
```

- Now create your prac_03 database for this weeks exercise and import your week 1 database using the `\i` command from within the `psql` client:

```
[mwelch8@turing ~]$ createdb mwelch8_prac_02

[mwelch8@turing ~]$ psql mwelch8_prac_02 < ~/prac_01.sql

[mwelch8@turing ~]$ psql mwelch8_prac_02
psql (9.4.4)
Type "help" for help.

...

mwelch8_prac_02=> \dt
             List of relations
 Schema |      Name       | Type  |  Owner
--------+-----------------+-------+---------
 public | department      | table | mwelch8
 public | dependent       | table | mwelch8
 public | dept_locations  | table | mwelch8
 public | employee        | table | mwelch8
 public | project         | table | mwelch8
 public | works_on        | table | mwelch8
(6 rows)
```

- Throughout this practical session, you should run each of the example queries and review the results returned from the COMPANY database.
- Now you are ready for the prac...

String Comparison Operations

- So far through the practical exercises, we have simply compared string-typed attributes to determine equality.
- SQL and PostgreSQL provide facilities for complex pattern matching string-type attributes.
- *Pattern matching* is achieved through the use of the `LIKE` operator.

- The pattern matching facilities can be used for identifying partial matches to patterns.
- The string patterns are specified through the use of two reserved characters:
  - `%` is used to specify an arbitrary section of 0 or more characters
  - `_` is used to specify a single character.
- The reserved characters are simply inserted into the comparison literal on the right side of the `LIKE` operator.
- Try These:

```
-- Retrieve all employees with an address in Houston,TX

SELECT fname,lname
FROM employee
WHERE address LIKE '%Houston, TX%'

-- Find employees that were born in the 1950's

SELECT fname,lname
FROM employee
WHERE date_part('Year',bdate)::VARCHAR LIKE '__5_'
```

- If you wish to use the '%' and '_' characters as part of a string literal for matching, you will need to escape them in your string using a '\' character:
  - 'AB_CD\%EF' represents the string 'AB_CD%EF'
- A few more interesting examples (source: http://www.postgresql.org/docs/9.4/static/functions-matching.html):

```
'abc' LIKE 'abc'    -- true
'abc' LIKE 'a%'     -- true
'abc' LIKE '_b_'    -- true
'abc' LIKE 'c'      -- false
```

- You can mess around with these in postgres:

```
SELECT 'abc' LIKE 'abc' AS answer;
SELECT 'abc' LIKE 'a%' AS answer;
SELECT 'abc' LIKE '_b_' AS answer;
SELECT 'abc' LIKE 'c' AS answer;
```

- PostgreSQL provides POSIX regular expressions through the use of the `SIMILAR TO` clause.
- `SIMILAR TO` supports these pattern-matching meta-characters borrowed from POSIX regular expressions:
  - | denotes alternation (either of two alternatives).
  - * denotes repetition of the previous item zero or more times.
  - + denotes repetition of the previous item one or more times.
  - ? denotes repetition of the previous item zero or one time.
  - {m} denotes repetition of the previous item exactly m times.
  - {m,} denotes repetition of the previous item m or more times. {m,n} denotes repetition of the previous item at least m and not more than n times.
  - Parentheses () can be used to group items into a single logical item.
  - A bracket expression [...] specifies a character class, just as in POSIX regular expressions.
- A few Simple ones:

```
'abc' SIMILAR TO 'abc'      -- true
'abc' SIMILAR TO 'a'        -- false
'abc' SIMILAR TO '%(b|d)%'  -- true
'abc' SIMILAR TO '(b|c)%'   -- false
```

- String-type attributes can be concatenated using the `||` operator:

```
-- Get the full name in a single returned field (with a space between the names)

SELECT fname||' '||lname AS Full_name
FROM employee;
```

Arithmetic Operations

- The standard arithmetic operators (`+`,`-`,`*`,`/`) can be applied to numerical values and attributes with numeric domains.

For example:

```
SELECT e.fname, e.lname, 1.1 * e.salary as Increased_salary
FROM employee AS e, works_on AS w , project AS p
WHERE e.ssn = w.essn AND
    w.pno=p.pnumber AND
    p.pname = 'ProductX';
```

- In addition to the basic operators, PostgreSQL provides a full range of builtin functions for performing common operations:

http://www.postgresql.org/docs/8.1/static/functions-math.html

Aggregate Functions

- Aggregate functions summarise information from multiple tuples into a single-tuple summary.
- The functions supplied include `COUNT`, `SUM`, `MAX`, `MIN` and `AVERAGE`
- A full list of the functions provided in PostgreSQL is available from: http://www.postgresql.org/docs/9.1/static/functions-aggregate.html
- These aggregate functions are applied to the individual attributes specified within a `SELECT` clause.
- An example for you to try:

```
-- Find the sum of all salaries, the max of all salaries and the min of all salaries.

SELECT SUM(salary), MAX(salary), MIN(salary)
FROM employee;
```

- We can add a``WHERE` clause to restrict the records that the aggregate functions are applied to:
- Some examples for you to try:

```
-- Find the sum of all salaries, the max of all salaries and the min of all salaries form the 'Research' department

SELECT SUM(salary), MAX(salary), MIN(salary)
FROM employee JOIN department on dno=dnumber
WHERE dname='Research';

-- A Count of all employees

SELECT COUNT(*) AS "Count of Employees"
FROM employee;

-- Count the number of distinct salary values in the database

SELECT COUNT(DISTINCT salary)
FROM employee;
```

Grouping

- In many situations we will need to apply aggregate functions to subgroups of tuples within a relation.
- The `GROUP BY` operations allows us to partition a relation into a set of non-overlapping groups.
- The `GROUP BY` operation specifies the attribute(s) to use for creating the subgroups.
  - Tuples with the same value for this specific attribute will be grouped together.
  - Any aggregate functions specified in the query will be carried out across each group. A single tuple will be returned for each of the groups partitioned.

```
-- Retrieve the number of employees and the average salary for each department.

SELECT dno, COUNT(*), AVG(salary)
FROM employee
GROUP BY dno;
```

- We can restrict rows returned (i.e. groups listed) in the aggregate results through the use of the `HAVING` clause.
- The `HAVING` clause puts a condition on the summary information
  - Only groups satisfying the condition will be returned.
- Examples for you to try:

```
-- For each project that has more than 2 employees, list the project number, name and number of employees working on the prc

SELECT pnumber, pname, COUNT(*)
FROM project, works_on
WHERE pnumber=pno
GROUP BY pnumber, pname
HAVING COUNT(*) > 2;
```

Views(Virtual Tables)

- A *View* is a virtual table derived from the base relations present within the database.
- Views are used to provide an additional layer of abstraction between the implementation of the database and user applications.
  - Changes can be made to the underlaying data structure (such as table/column names foreign keys etc.), without changes to the applications
  - Views can simply be updated to return the format expected by the application.
- Views can be created by using the `CREATE VIEW ... AS ...` clause
- Examples for you:

```
-- This view directly inherits the names of the SELECTed attributes from the
-- base tables

CREATE VIEW works_on1
AS SELECT fname,lname,pname,hours
FROM employee, project, works_on
WHERE ssn=essn AND pno=pnumber;

-- Look at the contents of your view...

SELECT * FROM works_on1;


-- This view renames the attributes and makes use of a GROUP BY clause to
-- return the number of employees and the total salary for each department

CREATE VIEW dept_info(dept_name, no_of_emps,total_sal)
AS SELECT dname, COUNT(*),SUM(salary)
FROM department, employee
WHERE dnumber = dno
GROUP BY dname;

-- Look at the contents of your view...

SELECT * FROM dept_info;
```

- To remove the view, use the `DROP VIEW ...` clause:

```
DROP VIEW dept_info;
```

- The data presented within a view is always up-to-date: If we modify the data in the tables on which a view is based, the data returned in queries to the view will be updated.
- Views data returned by a view can either be retrieved or calculated from the base table on-demand or they can be *Materialised* which involves physically creating a temporary table for the view on the assumption that other queries to the view will follow.
- This is managed within the DBMS.
- Update operations on views are possible if the view directly maps to a base relation.
  - Views that involve aggregate functions con't be updated as there relationship with the base tables can be ambiguous.
  - Views that involve table joins can be ambiguous, as they can map to the underlaying tables in multiple ways.
  - Updating a base relation via a view is usually not a good idea as the update may map to multiple update operations on the underlaying base relations.
  - These update operations can be unpredictable.

Exercises For You

Question 1

Specify the following queries on the COMPANY database in SQL. Execute your query and review the results.

a) For each department whose average employee salary is more than $30,000, retrieve the department name and the number of employees working for that department.

b) Retrieve all employees with a last name that starts the the character 'S'.

c) Find all employees that work on a project with the the word 'Product' in its name.

Question 2

Specify the following views in SQL on the COMPANY database. Execute your query and review the results.

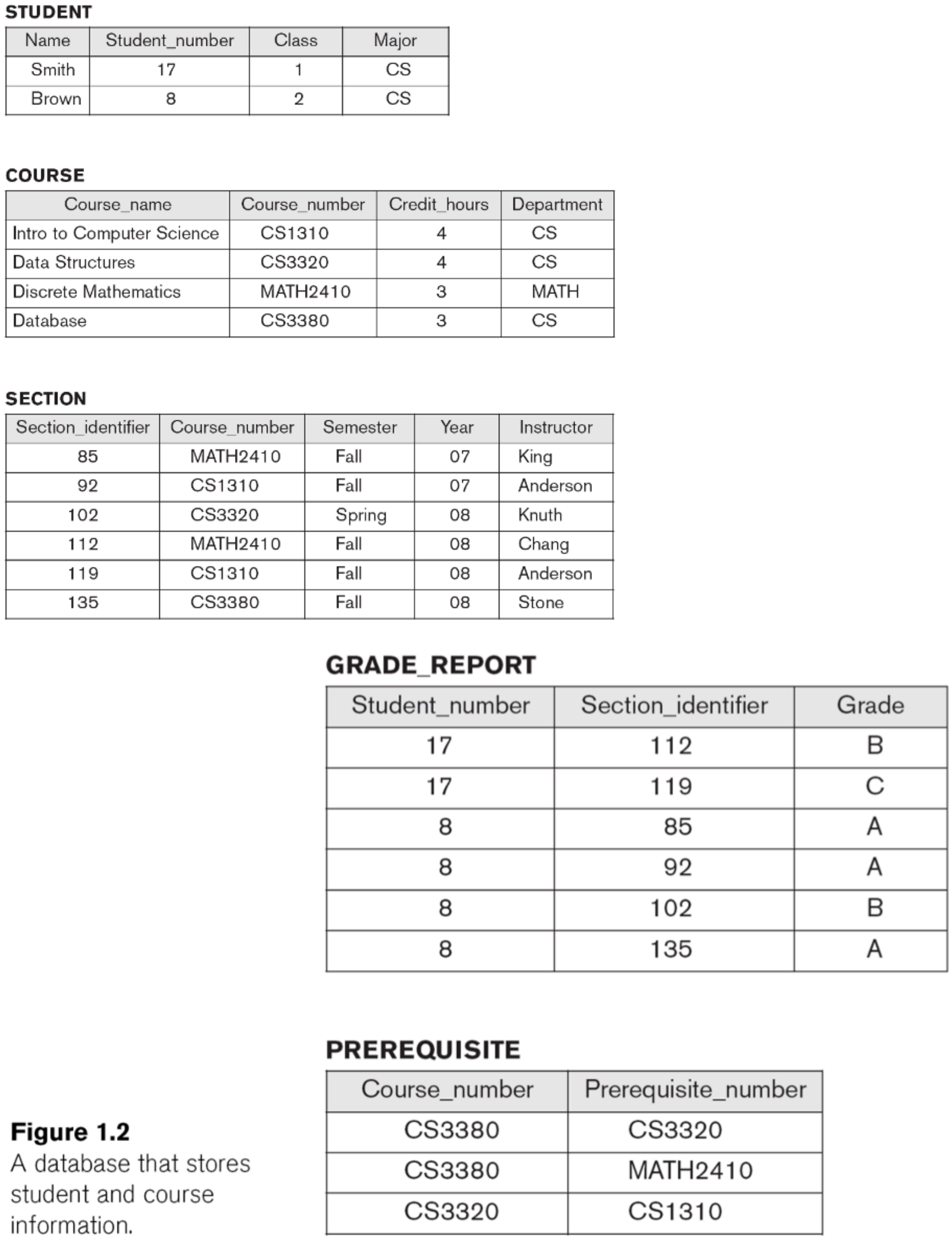
a) A view that has the department name, manager name, and manager salary for every department.

b) A view that has the employee name, supervisor name, and employee salary for each employee who works in the 'Research' department.

c) A view that has the project name, controlling department name, number of employees, and total hours worked per week on the project for each project.

Exercise 3

Using appropriate SQL DDL statements, create an SQL script (i.e. a file with the .sql extension) to implement that following schema:



STUDENT

Name	Student_number	Class	Major
Smith	17	1	CS
Brown	8	2	CS

COURSE

Course_name	Course_number	Credit_hours	Department
Intro to Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Database	CS3380	3	CS

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	07	King
92	CS1310	Fall	07	Anderson
102	CS3320	Spring	08	Knuth
112	MATH2410	Fall	08	Chang
119	CS1310	Fall	08	Anderson
135	CS3380	Fall	08	Stone

GRADE_REPORT

Student_number	Section_identifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

Figure 1.2
A database that stores student and course information.

Identify and implement the primary and foreign key values from the data supplied.

Finish your script by implementing a series of `INSERT` statements to load this data into the database. Run your script within your prac_02 database using the `psql` client's `\i` command.