Lecture IX
DATABASES and POSTGRES

We saw that the TIGER dataset is too large for in-core manipulation and visualization. We need to use some general database tools for this. This lecture gives a general introduction to relational databases and to using Java to access a Postgresql Server. There are four parts to this lecture:
(A) General introduction to the theory of relational databases.
(B) We look at a particular implementation, POSTGRES, and how to interactively use POSTGRES.
(C) We look at the SQL, a widely used query language for relational databases. Postgresql is the postgres version of SQL.
(D) We look at JDBC, the Java Database Connection Package, that allows you to access any database server (in particular Postresql). This will allow us to programmatically interact with a database.

§1. Relational Database Concepts

Let A, B be sets and $A \times B$ denote their Cartesian product. Any set $r \subseteq A \times B$ is called a **binary relation** over A, B. In general, if $A_1, \ldots, A_k$ are $k \geq 1$ sets, then a subset $r \subseteq \prod_{i=1}^{k} A_i$ is called a k-ary relation over $A_1, \ldots, A_k$. An element $t \in r$ is called a **tuple**. We may speak of the i-th **component** of t in the natural way ($i = 1, \ldots, k$).

First a clarification about the difference between "relationship" and "relation". If $(a, b) \in r$, we say that a and b "has the **relationship** r". Thus "relationship" is used on individuals. The collection of all relationships (of a fixed type) among individuals is called a relation.

There are several kinds of binary relations: one-one (husband-wife), many-one (child-parent), one-many (parent-child) and many-many (uncle-nephew). These concepts generalize to any k-ary relation ($k \geq 2$).

Schema. We can view each relation $r \subseteq \prod_{i=1}^{k} A_i$ as an instance of a **relation scheme** where the relation scheme just specifies a name R for the relation, and lists the domain sets $A_1, \ldots, A_k$. We may denote such a relation scheme as "$R(A_1, \ldots, A_k)$". We say r is an **instance** of the scheme $R(A_1, \ldots, A_k)$. A variant definition is to treat the A_i's as abstract domain names, in which case an instance r is a subset of $\prod_{i=1}^{k} D(A_i)$, relative to an assignment of each A_i to some set $D(A_i)$.

A **database scheme** is just a collection of relation schemes together with some set of **constraints**.

An example of a constraint is: $A_1 \rightarrow A_2, \ldots, A_k$. Suppose there is a relation scheme $R_1(A_1, \ldots, A_k)$. Then this constraint says that the attribute A_1 "determines" the attributes $A_2, \ldots, A_k$ in R_1. But more complicated constraints may involve two or more relation schemes.

Entity-Relation Analysis. The Object-Oriented (OO) view of the world is very natural for database theory. In the OO-view, all objects are classified into "classes", with the classes exhibiting a hierarchical structure. In database, we do not much emphasize this hierarchical structure, but are interested in a different extension of these ideas. This extension proposes to categorize all classes into one of three categories: **entities classes** (e.g., employees, companies), **relationships classes** (e.g., x is-employee-of y) and **properties classes** (e.g., n is-salary-of y).

We can such classes and their inter-relationship via a digraph: let each node represents a class. Under

the above categorization, we denote an entity class by a rectangle, a relationship class by a diamond, and an attribute class by an oval. Directed edges among these nodes can be used to represent their inter-relationships.

This is often called the ER-model of database modeling, and it is enough to capture many semantic aspects of a database. See Figure XXX for an example involving an airline company's database.

Formulas. We consider formulas involving

- Constants for numbers (e.g., $20,000$), strings (e.g., 'Jack') and boolean values.
- Variables (e.g., $A, B, Salary, Name$) and component numbers ($\#1, \#2$)
- Comparisons (e.g., $=, \neq, <, \leq$, etc)
- Logical connectives (e.g., $\wedge, \vee, \neg$)

These are put together using rules leading to formulas such as $F_1 : \#2 > \#3$, F_2: (Salary $= 20,000$)$\vee$(Name != 'Jack'), $F : F_1 \vee F_2$, etc. We then define the relationship of **satisfaction** between tuples and formulas: r **satisfies** F (denoted $r \vdash F$) if F is true under the intended r-interpretation of F.

Relational Calculus. These are operations involving relations. Since relations are sets, all the set theoretic operations are applicable, subject to the requirement that the underlying domains are "compatible". E.g., we can form the union of two relations only when they are instances of the same domain. Two interesting operations are **join** and **selection**. The selection operator on a relations r is written

$$\sigma_F(r)$$

where F is a formula as defined above. This amounts to selecting those tuples in r that satisfies F.

§2. Query Language

We need database languages to define and to query a database. The most widely used language in relational databases is called SQL. SQL is often pronounced "sequel", for the simple reason that it is derived from the language SEQUEL which was first developed at IBM. But it is also an acronym for "Structured Query Language". We give a brief introduction to SQL.

All database languages have two distinct sublanguages: the first is called the Data Definition Language (DDL) and it is the language for setting up the database scheme. The second sublanguage is called Data Manipulation Langauge (DML). It more important to the users, and is used for manipulating and querying instances of the database scheme. In the following, we follow the convention that SQL keywords are in caps, even though SQL keywords and identifiers are case insensitive (unless the identifier is quoted).

Basic DDL In SQL, we have the following basic forms of DDL statements:

```
> CREATE [TABLE | VIEW | INDEX ] name ...
          -- the other arguments depend on the 3 cases
> ALTER TABLE ...                  -- e.g., add column
> DROP [ TABLE | VIEW | INDEX ]
```

Tables is just SQL's term for relations. Notice that we have the concept of a "view", which is basically another table. This can be explained if we discuss the distinction between primary tables and derived tables. Indexes are search structures built over tables to facilitate fast queries. The following are some concrete examples:

```
> CREATE TABLE employeeTable
        (Ssno INTEGER UNIQUE,          -- social security number
         Name VARCHAR(30) NOT NULL,
         Street_Address TEXT,
         City VARCHAR(20) DEFAULT 'New York',
         Sex CHAR(1),
         Salary CASH DEFAULT '$0',
         PRIMARY KEY (Ssno));
> CREATE INDEX myIndex
         ON employeeTable(Name);         -- allow fast search on names
> CREATE VIEW myView AS
         SELECT *
         FROM employeeTable
         WHERE Salary > '$20,000';
```

Note that each command is ";" terminated. End-of-line comments (starting with "–") can be interspersed with a command. The data types include `INTEGER` (or `INT`), `CHAR`, `VARCHAR`, `CASH`, etc. Note that `TEXT` is just `VARCHAR` with no limit on the length. Constants are singly quoted as in '$0' and 'New York'. Furthermore, fields can be specified to be "NOT NULL", given DEFAULT values, empty string is an invalid entry.

Postgres supports a rich set of datatypes including: `int, real, timestamp, interval, date, point`. Clearly `timestamp, date` are useful. For example, '2003-03-24' is a date and '(1.23, 45.6)' can represent an interval as well as a point. The datatype of `point` is particularly interesting for us, as we can represent all Longitude/Latitude pairs as a point.

Basic DML First consider a group of 4 basic statements: Insert, Select, Delete and Update.

```
> INSERT INTO myTable(Name, Ssno, Street, City)
        VALUES ('Jack', '232323232', '21, Hill Road', 'New York');

> SELECT Name, Ssno
        FROM myTable
        WHERE City = 'New York';
>>   Name      | SSNO
     ----------+----------
     Chee      | 123456789
     Patricia  | 111111111
     Jill      | 212121212
```

We can also delete and update:

```
> UPDATE myTable
```

```
        SET Salary = Salary + '$10000'
        WHERE City = 'New York';
> DELETE FROM myTable
        WHERE City = 'Newark';
```

The simplest form of Selection is

```
> SELECT *
        FROM myTable;
> SELECT Name, Salary            -- makes a 2 column result
        FROM myTable
        WHERE Salary > '$20,000';
```

The Select statement can be quite complex: (1) the FROM-clause can involve several tables, (2) the WHERE-clause can be a complex formula, (3) the output table can have its attributes reordered and given new names, and output rows can be ordered by the values in any attribute, etc. Note that we can rearrange the order of the attributes in myTable during insertion.

§3. Postgres Server

To let you gain some experience with relational databases, it is useful to learn some particular system. We will use the freely available Postgres DBMS which was originally developed at UC Berkeley. It has an SQL language interface. Since SQL is the de facto standard for relational query languages, you will have no problem moving from this particular DBMS to another other relational DBMS. Indeed, you can begin to use the various Ken Been (been@cs.nyu.edu) has kindly set up a postgres server for this class:

```
host: slinky                -- full name is slinky.nyu.edu
username: visclass          -- you will need the password from me
database name: visdata
port number: 5432           -- this is the default for postgres,
                                so no need to specify

To connect from anywhere, you need to have the
postgres client application installed on your computer,
then run:

> psql -h slinky -U visclass visdata
```

You will be prompted for the password of user "visclass". and when this is entered, you will get a welcome message followed by the prompt "visdata=¿". For our illustrations below, we simply indicate the prompt by "=¿".

CLASS NOTE: Unless you install your own Postgresql program on your machine, you must use the CS class account that I have obtained you. Your files are on the machine named sparky.cs.nyu.edu. This account is meant for class work, and it will expire after the class is over. Your classmate (Zubin) successfully installed his own Postgresql on his laptop, so he does not need this account.

Once you are in psql, you can issue various online commands. There are two classes of commands: (1) regular SQL commands, and (2) commands.

All SQL commands must be terminated by ";" before it is executed (but note the "
g" exception). This means you can enter SQL commands over two or more lines (the prompt changes to "visdata-¿" when this happens). The command are so-called because the first key stroke is the backslash "\". They are generally short and never require the ";". Here are some useful commands:

```
=> \?
          -- this lists the other \-commands.
=> \h
          -- this lists SQL commands.
=> \q
          -- this quits from sql

To read in a list of sql commands in a file
called commands.sql, you type:

=> \i  commands.sql

If you are working in the DB visdata (under User visclass), and
you want to switch to the DB rawdata (under User tiger), type:

visdata=> \c rawdata tiger

You will be prompted for the password of tiger, and when given, you
will now be working in DB rawdata (prompt becomes "rawdata=>").

A very interesting \-command is \g.  Here is an example:

=> SELECT * FROM Books
=> \g Outfile

Notice that the SELECT command was not";" terminated, so it is
pending.  The command "\g" asks psql to execute the pending command
and put the result in a file called "Outfile".  Of course, you
can have any SQL command besides SELECT.
```

First Introduction to SQL on Postgresql. Here is a small interactive session (the responses of Postgresql are not shown):

```
=> CREATE TABLE Books (
->         Title VARCHAR(20),
->         Price MONEY);
                        -- response is "CREATE"
=> INSERT INTO Books VALUES ('Intro Java', '$20.99');
=> INSERT INTO Books VALUES ('Database Systems', '$40.00');
=> INSERT INTO Books VALUES ('Networking', '$51.00');
=>
```

```
=> SELECT Title FROM Books;
                        -- this displays the 3 titles above
=> SELECT * FROM Books WHERE Price > '$40';
                        -- this displays only 'Networking'.
=> UPDATE Books SET Price = Price - '$1.00'
->         WHERE Title != 'Intro Java';
                        -- the price of two books is reduced
=> DROP TABLE Books;    -- useful when you are experimenting
                        -- with PSQL
```

Note that the DB admistrator have given permission to User visclass (i.e., you) to create its own databases. We suggest that each group works under their own databases, using a name that identifies you:

```
=> CREATE DATABASE chee_db; -- created my own database
=> DROP DATABASE chee_db; -- remove my database just created
=> \l -- lists all the current databases
```

Select Command. We know that JOINs are very important, but Postgres has no operation by this name! The reason is that it could be simulated from the SELECT operation: here is an example from the Postgres tutorial. We have 3 tables called Supplier, Part and Sells. We join them over their common attributes to produce a binary relation between Supplier Name and Part Name:

```
=> SELECT S.Sname, P.Pname
->      FROM Supplier S, Part P, Sells SP
->      WHERE  S.Sno = SP.Sno AND
->     P.Pno = SP.Pno
```

In short, SELECT has the implicit ability to perform Cartesian products and also projections. These, combined with the ability to select tuples, gives us JOINs.

Here is the SELECT Command in its full glory:

```
SELECT [ALL | DISTINCT [ON column] ]
        expression [ AS name ] [, ...]
        [ INTO [TABLE] newTable ]
        [ FROM table [alias ] [, ...] ]
        [ WHERE condition ]
        [ GROUP BY column [, ...] ]
        [ HAVING condition [, ...] ]
        [ UNION [ALL] select ]
        [ ORDER BY column [ ASC | DESC ] [, ...] ]
```

What is important to remember is that the order of these classes (INTO, FROM, WHERE, GROUP, etc) must be presented in this order. Let us explain some of the variables in this command summary:

- `expression`: name of a column or an expression. The column must be from tables referenced by the FROM clause. E.g., if the FROM class refers to tables S and P, then you can use S.Sname and P.Pname in expression.

- `name`: another name for a column or an expression using the AS clause. Note that "name" cannot be used in the WHERE condition.
- `newTable`: see the SELECT INTO statement for more info.
- `alias`: an alternative name for the previous table, either for brevity or to eliminate ambiguity in joins.

The SELECT INTO variant of this command is useful when you want to implicitly create new tables using the result of a selection. For instance, suppose you want to create a new table called "S1" comprising only the supplier names in the S table whose supplier number (Sno) is greater than 0:

```
=> SELECT S.Sname INTO S1 FROM S WHERE S.Sno > 0;
```

Presumably, when Sno is ≤ 0, the Sname may not represent a real supplier.

The COPY and
copy commands. Suppose you have just created (or have an existing) table called `myExamples`. You want to insert into `myExamples` a set of tuples found in a text file `examples.txt`. Each line of `examples.txt` represents a tuple. Two consecutive components of a tuple are separated by some fixed delimiter character. The default delimiter character is the "tab", but you can replace it by any other character such as "—", provided you tell the copy command about this (see reference manual). If you need the delimiter character in your actual data, just precede it by a backslash (").

There are two "copy" commands to insert the tuples of `examples.txt` into `myExamples`, one is a -command and other in SQL:

```
=> COPY myExamples FROM '/home/user/yap/postgres/examples.txt';
=> \copy myExamples FROM 'examples.txt'
```

Note that difference in syntax in calling these two versions of copy: the SQL version requires a full path name and is ";", the later can use a relative path name and does not expect a ";". Furthermore, in the SQL version, the file examples.txt must be visible to the postgres server (in particular, the file is mounted on the server machine) and you have suitable read/write permission (often superuse permissions). For this reason, the
copy-version is more useful.

It is clear that this command is extremely useful in our handling of TIGER data – all the tiger files can be entered into our database this way.

The copy command can also handle binary input files. There are some common pitfalls in using COPY. One common one is the presence of **roque characters** – e.g., in TIGER, you will encounter the presence of the single quote "'" character. For instance, in TGR36047.RTC, you will encounter a string "NYC-CHANCELLOR'S OFFICE". This will cause Postgres to think that you are about to insert a literal constant, causing error. Another common error relates to NULL values. If you have to enter a null value, use the string 'NULL' instead of just an empty white space. The default representation of NULLs is "
N" (backslash-N) but this can be changed with the clause "WITH NULL AS 'alternateChar'". If you have empty fields in your data, then use the clause "WITH NULL AS " " in the COPY command. Otherwise, COPY will think that empty fields represent empty strings, and will generate an error.

Arrays. An extremely useful "datatype construct" in Postgres is the array construct. For any datatype T, we can define a new datatype called an array of T, and denoted T[]. For instance, if T=TEXT (a string), then TEXT[] or even TEXT[][] are valid types. An array of integers would be INTEGER[] (or simply INT[]). How do you specify array constants? As in C-language, by enclosing a comma-separated list of constants within braces: thus is an array of 4 integers is written `'{1, 2, 3, 4}'`, and here is an array of two strings `'{"aaa", "123"}'`. Note that we still need to enclose everything within the single quote mark (`'`). An empty arrays is denoted `'{}'`. Notice that the length of the arrays are not pre-specified.

To address the i-th element of an array, use an indexing that begins with 1. Thus, in our Supplier-Part running example, if `P.Components` has type INT[] (representing a list of part component numbers), then we can write queries such as:

```
  => SELECT * FROM P
WHERE P.Components[1]='4';
```

For multidimensional arrays, we use colon as separators for the indices. E.g., if `P.Components` has type INT[][], we can write `P.Components[1:3]` to refer to the value in the first row and third column.

SEE ALSO: an online collection of useful Postgresql topics at http://cs.nyu.edu/ yap/classes/visual/curr/lect/l9/sql/.

§4. Indices and Geometric Types

An **index** over any attribute is a data structure that facilitates fast responses to queries on that attribute. For instance, if we want to quickly search for given titles in our Books relation, we would create an index on the "Title" attribute. Postgres supports three kinds of indices: B-Tree, R-Tree and Hash. The default is kind of index is B-Tree, but the user can specify explicitly any kind of desired index:

```
=> CREATE INDEX TitleIndex ON Books(Title};
        -- creates a B-Tree index on Title
=> CREATE INDEX PriceIndex ON Books USING HASH (Price);
        -- creates a Hash index on Price
=> DROP INDEX TitleIndex FROM Books;
        -- delete the index on Title
=> CREATE INDEX BoxIndex ON TigerData USING RTREE (GObj));
```

The last command assumes we have a table named TigerData which has an attribute named GObj whose datatype is a geometric type such as "BOX" or "POLYGON" (see below). The R-Tree index is appropriate to such data types. Although the user decides on whether or not to create an index, the way to use an index (or even to use it at all), is an automatic decision of the Postgres query engine that the user has no control over.

Geometric Datatypes. We have just noted the presense of geometric data types in Postgres in connection to R-Trees. Postgres is rather unique among DBMS in its strong support of a rich set of geometric data types. Naturally, such data types are very important for our visualization applications. Let us now go over the various geometric datatypes: there is nothing surprising in the following list.

A **point** is a pair (x, y) of floating point numbers. A **line Segment** is represented by a pairs of points, $((x_1, y_1), (x_2, y_2))$. where (x_i, y_i) are the end points of the line segment. A **box** is represented by pairs of opposite corners of the box, $((x_1, y_1), (x_2, y_2))$. So, syntactically, it is indistinguishable from a line segment. However, it is conventional to have (x_1, y_1) as the upper right corner and (x_2, y_2) as the lower left corner. A **path** is represented by sequence of points, either $p = ((x_1, y_1), ..., (x_n, y_n))$ (open) or $p = [(x_1, y_1), ..., (x_n, y_n)]$ (closed). The path p is "open" if the first and last points of the sequence are distinct, and "closed" otherwise. Functions popen(p) and pclose(p) force a path to be open or closed, while functions isopen(p) and isclosed(p) test for either type in a query. A **polygon** is represented by sets of points, as follows $P = ((x_1, y_1), ..., (x_n, y_n))$. **Circles** are represented by a center point and a radius, as in $< (x, y), r >$ or $((x, y), r)$. Circles are output using the first syntax.

Postgres provides a very rich set of operations involving these objects. Let us just focus on some operations on boxes and points. First, we can access the components of a point as in an array: $p[0]$ and $p[1]$ would get the first and second coordinates. Similarly, $B[0]$ and $B[1]$ would return the first and second point in the box B. Given boxes B, B' and a point p, we can translate the box with the operators $B + p$ or $B - p$, and we can scale the box as in $B * p$ and B/p. We can check if B intersects B' or p is in B with the predicates $B \# B'$ and $p \# B$. The predicate box $B \,\&< B'$ tells us whether B is left of, or overlaps, B'. The two predicate $\&>$ is similarly defined (with right instead of left). The query $B = B'$ tests whether B is the "same as" B'. The query $B \&\& B'$ tests whether two boxes overlap. When dealing with polygons P, the following operations would be useful: **polygon(**B**)** converts a box to a 4-sided polygon, and **box(**P**)** converts a polygon to a bounding box. Similarly, we can interconvert between between polygons and paths.

EXAMPLE. Suppose we want a table `TigerLines` where each row represents a Tiger Line. Each line has a unique `Tlid` attribute, a `StartPt, EndPt` and also a path `Detail` that begins with the `StartPt` and ends with `EndPt`. Unfortunately, Postgres does not allow RTree indexing on paths, only on polygons. For this reason, we will pretend that `Details` is really a polygon in the following:

```
=> CREATE TABLE TigerLines
->    (Tlid INT, StartPt POINT, EndPt POINT, Details POLYGON);
=> ...
=> SELECT COUNT(*)
-> FROM TigerLines
-> WHERE StartPt @ box('(-74083679, 40603082), (-74002678, 40613100)');
=> SELECT COUNT(*)
-> FROM TigerLines
-> WHERE box(Details) &&
->           box('(-74083679, 40603082), (-74002678, 40613100)');
```

Thus operator `@` returns true iff a point lies inside a box. So this query will return the number (i.e., COUNT) of Tiger Lines whose `StartPt` lies in the box defined as `box('(-74083679, 40603082),(-74002678, 40613100)')`.

The second selection returns those tiger lines whose bounding box intersects the specified query box. In general, the operator `&&` is the intersection operator (but the types of the two input arguments must be the same (box-box, circle-circle, polygon-polygon, etc).

Storage of TIGER coordinates in Postgres. Consider the problem of entering the Longitute/Latitude data in TIGER into our database. In TIGER, such values are represented by 9 or 8 digits, with a sign $(+/-)$ as prefix. E.g., $(-73921406, +40878178)$. This is to be interpreted as $(-73.921406, +40.878178)$ in degrees. The simplest solution is to store these values as a pair of "text strings". But you lose the ability to use

geometric operations or spatial queries on such data. We suggest you store such data as Postgress points, but as a pair of integers and *not* as a pair of floating point values i.e., as $(-73921406, +40878178)$ and not $(-73.921406, +40.878178)$. There is a slight paradox here: Postgres points are stored as a pair of machine floats. So, even though you store integer values, it is really represented by machine floats. Why this paradox? Because we will lose precision if we explicitly convert the integers into floats. On the other hand, since each integer in a Lon/Lat pair is at most 10 digits, these integers have exact representations as machine floats.

§5. JDBC

In the previous section, we saw an interactive interface to a Postgres. But to build up a credible database, you want to enter a substantial volume of data into the database. This data is pre-existing somewhere, and you want to enter the data programmatically. More generally, how do you use a program to interact with a database?

The answer lies in Java's JDBC ("Java Database Connection") Package which allows Java programs to talk to all major relational database management systems (Postgres, Oracle, MS Access, MS SQL, Informix, IBM Database 2, DBASE, Sybase, mySQL, Ingres, FoxPro, etc). Part of this universal property comes from the fact that all modern relational DBMS's support a SQL-like langauge.

Here are the mechanics of using JDBC:

- Database URLs: these "URLs" provides JDBC with details about the data server. Our database URL looks like this:

```
url = "jdbc:postgresql://slinky.nyu.edu:5432/visdata"
```

  The url is (basically) a ":" separated list of arguments indicating that (1) postgresgl is our protocol, (2) slinky.nyu.edu is the host, (3) 5432 is the port number (4) visdata is the database name.

- Making connection: assuming we have imported the sql package, we can now make a connection to the data server:

```
import java.sql.*;
...
dbUser = "visclass";
dbPassword = "visclass";
con = DriverManager.getConnection(url, dbUser, dbPassword);
```

- Prepare SQL command, which is just a string:

```
String command = "UPDATE Books"
                + " SET Price = Price - '$5.00'"
                + " WHERE Title !~ 'Networking'";
```

  We are using the Books relation created in the interactive psql session in a previous example. This UPDATE will lower by $5.00 the Price of all Titles other than 'Networking'.

- Execute the command: First we have to use the Connection con to give us a `Statement` object. Then we use it to execute our command.

```
        Statement s = con.createStatement();
        int n;
n = s.executeUpdate(command);
```

The returned value n is the number of tuples that are updated. The `executeUpdate` is used for SQL "updating" commands such as INSERT or DELETE.

- Getting results from a Query: instead of updating, some SQL commands such as SELECT returns a potentially large set (called a `ResultSet`). We need a variant of `executeUpdate` called `executeQuery`.

```
        ResultSet rs;
String query = "SELECT * FROM Books";
rs = s.executeQuery( query );
```

- Processing result sets:

```
        while (rs.next()) {
     System.out.println(  "("
                 + rs.getString(1) + ", "
                 + rs.getMoney(2) + ")"
        }
```

Note that we need to do at least one `rs.next()` in order to extract the results. Also, instead of `getString(1), getMoney(2)` we could have used `getString('Title'), getMoney('Price')`.

- Closing the Connection: `con.close()` will do.

In addition to the executeUpdate and executeQuery commands above, there is a generic plain `execute` command that covers both possibilities (the user than has to determine which kind of execution took place, and to retrieve the ResultSet or the returned integer accordingly.

Exercises

Exercise 5.1: Consider the use of the GROUP BY clause in SELECTION. Normally, you follow the GROUP BY clause with a HAVING clause, which eliminates the groupings. What is the nature of the ResultSet when you do SELECT using a GROUP BY clause, but without using the HAVING clause? For more information, see the Postgres User's Guide, section 4.2. (e.g., http://jamesthornton.com/postgres/7.3/). ♢

End Exercises

§6. Example of Reading Tiger Files

We illustrate the means to enter the TIGER dataset into Postgres, and to access tables in Postgres. The following example from Bansi Kotecha shows you how to do it.

PROGRAMMING NOTE: Even if you can compile the following programs, you may encounter errors at runtime. If you get NULL pointer errors, we suggest you make sure that your java runtime environment have access to the postgresql classes. On the CS machines, you should make sure that your CLASSPATH variable contains the following directory: `/usr/unsupported/packages/postgresql/postgresql/share/postgresql.jar`. If you are compiling the

Reader.java program we provide, then make sure that "." is also in your CLASSPATH, since it will have to find RawDataReader.class. Note that you can also put postgresql.jar in (e.g.) `/usr/java/j2sdk1.4.1` (for java 1.4), but for some other `.../jre` directory (depending on your installation). You can download your own PostgresQL JDBC Drivers (prebuilt) from `http://jdbc.postgresql.org/` (it is some effort to build your own).

```
import java.sql.Connection;
import java.sql.DriverManager;

abstract public class RawDataReader {

    public static String database = "visdata";
    public static String dbHost = "slinky.cs.nyu.edu";
    public static String dbPort = "5432";
    public static String dbUser = "visclass";
    public static String dbPassword = "visclass";
    public Connection con = null;

    public Connection establishConnection() {
        String url = "";
        try {
            url = "jdbc:postgresql://" + dbHost + ":" +
                dbPort + "/" + database;
            Class.forName("org.postgresql.Driver");
            con = DriverManager.getConnection(url, dbUser, dbPassword);
        con.setAutoCommit(false);
        }catch (Exception e){
            e.printStackTrace();
        }
               return con;
    }

    public void commit() {
        try {
                con.commit();
        }catch(Exception e) {}
    }
}
```

We next want to write a program to read any TIGER file into the database. This requires an intermediate concept of a "format file". A line beginning with a `#` is deemed a comment and is ignored. White spaces and empty lines are also insignificant.

```
# Sample Format for reading an RTS TIGER file

# Number of attributes
6

# Number of fields per attribute
#        (this number is always 4 for TIGER files)
4
```

```
# The four attribute have the form:
#        (name, type, begin, end)

# Attribute 1
STATE N 6 7

# Attribute 2
COUNTY N 8 10

# Attribute 3
FIPS N 15 19

# Attribute 4
FIPSCC A 20 21

# Attribute 5
PLACEDC A 22 22

# Attribute 6
NAME A 53 112

# END
```

Consider the first attribute: it says that the name of this attribute is STATE, and that it is a numeric ("N") datatype, and it is the 6th to the 7th character of each input line. The other indicated by "A" is for algbetic or text datatype.

The Reader Class, defined next, is an extension of RawDataReader and it performs the following actions:
(1) use JDBC to open a connection to the postgres server,
(2) read and process a format file,
(3) read each line from an input (TIGER file) and inserting the corresponding information into an appropriate table, and finally
(4) commit the changes and close the connection.

```
import java.io.*;
import java.util.*;
import java.sql.*;

public class Reader extends RawDataReader {

    private final int fileExtLength = 3;
    private String frmtFile; // e.g., RT1.fmt
    private String dataFile; // e.g., TGR34017.RT1
    private String tableName; // Extracted from dataFile, e.g., RT1
    private String[][] frmtdata;
    private int noOfAttributes;
    private int fieldsPerAttribute;
    private int countRows = 0;
    private Statement stmt;
    private char option;
```

```
    public Reader(String[] args) {
getArguments(args);
tableName = getTableName(dataFile);
readFrmtFile();

try {
    stmt = establishConnection().createStatement();
} catch(SQLException se) {
    se.printStackTrace();
}

if((option == 'c')  || (option == 'C'))
    createTable();
readRawDataFile();
    }


    public void getArguments(String args[]){
if(args[0].startsWith("-")) {
    // In this simplified program, we assume only one option.
    option = args[0].charAt(1);
    this.frmtFile = args[1];
    this.dataFile = args[2];
}
else {
    this.frmtFile = args[0];
    this.dataFile = args[1];
}
    }

    private String getTableName(String filename) {
String ext = filename.substring(filename.length() - fileExtLength);
return ext;
    }

    private void readFrmtFile() {
        int count = 0;
BufferedReader br = null;
try {
    br = new BufferedReader(new FileReader(frmtFile));
    String line;
    int i = 0;
    while((line = br.readLine()) != null){
line = line.trim();
int x = line.indexOf("#");
if  (x >= 0)  line = line.substring(0,x).trim();
if (line.length() == 0) continue;
count++;
if(count == 1) {
noOfAttributes = Integer.parseInt(line);
}
else if(count == 2) {
fieldsPerAttribute = Integer.parseInt(line);
frmtdata =
```

```
  new String[noOfAttributes][fieldsPerAttribute];
}
else {
if(i != frmtdata.length) {
    parseFrmtdata(line, i);
    i++;
}
}
    }
}catch(Exception e){}
    }

    private void parseFrmtdata(String line, int i) {
StringTokenizer st = new StringTokenizer(line, " ");
int j = 0;
while (st.hasMoreTokens()) {
    frmtdata[i][j] = st.nextToken();
    j++;
}
    }

    private void createTable() {
try {
    String query =  "create table " + tableName + "(";
    for(int i = 0; i < frmtdata.length; i++) {
if(i == (frmtdata.length - 1))
    query += frmtdata[i][0] + " "
+ getDataTypeAsString(frmtdata[i][1]);
else
    query += frmtdata[i][0] + " "
+ getDataTypeAsString(frmtdata[i][1]) + ", ";
    }
    query += ")";

    //print("create query is : " + query);
    int i =  stmt.executeUpdate(query);
    if(i == 1)
print("Table " + tableName + " created successfully");
    else
print("Error: Could not create table " + tableName);
} catch(SQLException se) {
    se.printStackTrace();
}
    }

    private String getDataTypeAsString(String given){
if(given.equals("N"))
    return "integer";
if(given.equals("A"))
    return "text";
return ""; // should be error
    }

    private int stringToInt(String value) {
```

```
if(value.equals("")) {
    return -1;
}
else {
    return Integer.parseInt(value);
}
    }

    /*********************************************************************
      readRawDataFile is the main reading function
      -- it reads each line of dataFile and inserts into database table.
     *********************************************************************/
    private void readRawDataFile() {
//print("readRawDataFile() called");
try {
    BufferedReader br = new BufferedReader(new FileReader(dataFile));
    String record, query;
    String[] attributes;
    while((record = br.readLine()) != null) {
attributes = parseRecord(record);
query = createQuery(attributes);
insertIntoDatabase(query);
    }
    commit();
    print("\n" + countRows + " rows inserted from file " + dataFile);
}catch (Exception ie) {}

    }

    private String[] parseRecord(String record) {
String[] arrayOfAttributes = new String[frmtdata.length];
for(int i = 0; i < frmtdata.length; i++) {
    arrayOfAttributes[i]
= record.substring(Integer.parseInt(frmtdata[i][2]) - 1,
    Integer.parseInt(frmtdata[i][3])).trim();
}
return arrayOfAttributes;
    }

    /*********************************************************************
      Quoting all roque characters in the raw input files: \'
     *********************************************************************/
    private String checkCharacters(String value) {
StringBuffer sb = new StringBuffer(value);
for(int j = 0; j < sb.length(); j++) {
    if(sb.charAt(j) == '\'') {
sb.insert(j-1, '\'');
    }
}
return sb.toString();
    }

    private String createQuery(String[] attributes) {
String query = "";
```

```
    query += "insert into " + tableName + " values (";
    for(int i = 0; i < frmtdata.length; i++) {
if(i == (attributes.length - 1)) {
    if(frmtdata[i][1].equals("N"))
query += Integer.parseInt(attributes[i]);
    else {
//attributes[i] = checkCharacters(attributes[i]);
query += "'" + attributes[i] + "'";
    }
}
else {
    if(frmtdata[i][1].equals("N"))
query += stringToInt(attributes[i]) + ",";
    else {
//attributes[i] = checkCharacters(attributes[i]);
query += "'" + attributes[i] + "',";
    }
}
    }
    query += ")";
return query;
    }

    public void insertIntoDatabase(String query) {
try {
int i = stmt.executeUpdate(query);
    if(i == 1) {
countRows++;
if ((countRows % 200) == 0)
    print("Row " + countRows + " successfully inserted");
    }
    else
print("could not complete insertion");
}catch(Exception se) {
    se.printStackTrace();
}
    }

    /*****************************************************************
      The rest of the code are for testing
     *****************************************************************/

    private void print(String s) {
System.out.println(s);
System.out.flush();
    }


    /*****************************************************************
      The main program
     *****************************************************************/
    public static void main(String args[]){
Reader r = new Reader(args);
    }
```

```
}
```

Note that we deal with null numeric values by writing the methods `stringToInt(String)` which returns a -1 when the null string is detected.

After you have compiled Reader.java, you can call in one of the following two ways:

```
> java Reader RT1.fmt TGR36061.RT1
> java Reader -c  RT1.fmt TGR36061.RT1
```

Use the first form if you already have a relation called TGR36061 in your database. Use the second form if you need to create the relation. In either form, you will need a "fmt" file as specified above, and a raw TIGER file.

Exercises

Exercise 6.1: Consider the problem of entering the Longitute/Latitude data in TIGER into our database. In TIGER, such values are represented by 9 or 8 digits, with a sign $(+/-)$ as prefix. E.g., $(-73921406, +40878178)$. This is to be interpreted as $(-73.921406, +40.878178)$ in degrees. The simplest solution is to store these values as "text strings". But you lose the ability to use geometric operations or spatial queries on such data. So we want you to modify our FORMAT file to allow a new datatype indicated by the character "L" (for Longitude/Latitude). We want to store a pair of such signed integers (e.g., $(-73921406, +40878178)$ as a point). Unfortunately, this means that a L-datatype requires 6 fields in the FORMAT file: Instead of the 4 fields, `(name, type, begin, end)`, we now have `(name, type, begin-lon, end-lon, begin-lat, end-lat)`, That means the second numeric argument of our FORMAT file should now be re-intepreted to mean "*maximum* number of fields per attribute". The actual number of fields need depends on the individual data type. Thus, "N" (numeric) and "A" (text) continues to need needs 4 fields.
(a) Please modify the Reader class to accomodate such "L"-datatype.
(b) Use the new Reader to enter the RT1 data of your favorite TIGER county into the Postgres database; now perform some box queries on these points.
(c) Write a display program to display all the points of your favorite county. Your program shoul allow the user to "highlight" any subset of these points that falls within a query box.
(d) In addition to "L", we also want another data type denoted "B" for "Box type". This requires 2 points and hence 10 fields. Please ♢

End Exercises

END OF LECTURE