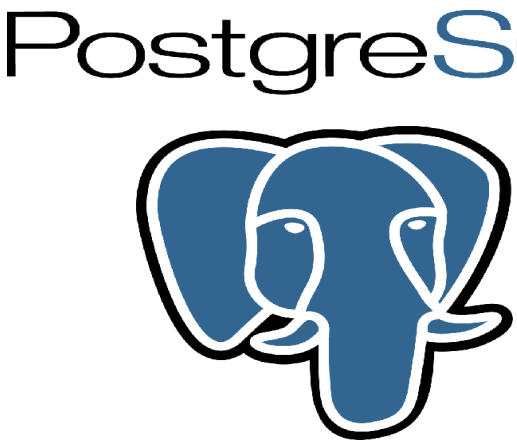
PostgreSQL/PostGIS Workshop

September 2008

Brent Wood

Version 1.0



PostGIS

NIWA
Taihoro Nukurangi

Table of Contents

Introduction.

Relational databases.

Postgres is an Object Relational Database, but we will focus here on it's basic capabilities as a relational database. The relational model, originally described by Codd & Date, is a rigorous model of entities, attributes and relationships. The SQL (Structured Query Language) which is used to interact with any RDBMS (Relational DataBase Management System) is essentially an expression of relational algebra. A record in a table generally represents a real world object, with its columns storing information about various attributes of the object. Some of these attributes, and other tables, can be used to describe the relationships between the objects represented in the tables. The process of abstracting real-world objects and their relationships into database tables is called data modeling.

PostgreSQL.

Postgres is released under a BSD style licence, and is thus free software. As with many other open-source programs, Postgres is not controlled by any single company, but relies upon a global community of developers and companies to develop and maintain it.

Postgres' ancestor was Ingres, developed at the University of California at Berkeley (1977-1985). This was one of the first commercially successful database applications. In 1995 SQL functionality was added to to the package, which became called Postgres95. In late 1996, the name of the database server was changed from Postgres95 to POSTGRESQL. It is a mouthful, but honors both the Berkeley name and its SQL capabilities, although it is often just called Postgres.

Postgres is used by many large commercial companies, including Sun and Yahoo!. Commercial database companies such as EnterpriseDB and Command Prompt sell support and help fund the development of Postgres, as do many other companies using Postgres around the world.

Until recently, Postgres was essentially only supported on UNIX/Linux type operating systems, although it could be made to run under Windows. However, Postgres is now fully supported on Windows, as are some of the common Postgres tools and extensions such as PgAdminIII and PostGIS.

Postgres has a genuine client server architecture, with the server accessed by a client application. This client is often the psql command line application, but a variety of other clients exist, including PgAdminIII, a popular GUI.

PostGIS.

While Postgres has some native support for spatial features (geometries) this is now unsupported and deprecated. PostGIS is used to provide support for managing & querying spatial data in Postgres. PostGIS is an implementation of the Open Geospatial Consortium's SFS specification for storing spatial data in an RDBMS, much like Oracle Spatial, DB2 Spatial, the Informix Spatial Data Blade and, from 2008, MS SQL Server.

PostGIS is developed and maintained by Refractions Research, who rely on users funding development, as well as their own consultancy services in the wider GIS arena. For example, NIWA has funded the

implementation of specific functionality relevant to our use of PostGIS. A number of clients exist to provide a map view of data stored in PostGIS, including QGIS, uDIG, JUMP and gvSIG. ESRI have added PostGIS support this year.

The name "PostGIS" is used hereafter to describe the combination of Postgres plus the PostGIS functionality.

A PostGIS database has all the usual Postgres ORDBMS functionality, including functions for managing and querying various data types, including integer & floating point numbers, character (alphanumeric), date and time, even IP addresses and other specialised types. PostGIS adds to these, providing support for geometry data types, including POINTS, LINESTRINGS and POLYGONS (as well as multi versions of these). Functions for querying, measuring, editing, overlaying, importing & exporting geometries are also provided, in line with the OGC SFS specification.

For a history decribing the development of PostGIS, see:
http://www.refractions.net/products/postgis/history/

Other resources.

Please note that this is a very basic introduction to Postgres, PostGIS and Relational Databases. Some useful online references include:
http://www.postgresql.org/docs/
http://www.postgresql.org/files/documentation/books/aw_pgsql/index.html
http://www.commandprompt.com/ppbook/
http://www.yolinux.com/TUTORIALS/LinuxTutorialPostgreSQL.html
http://www.sqlzoo.net
http://www.intermedia.net/support/sql/sqltut.shtm
http://sqlcourse.com
http://www.3schools.com/sql/default.asp

Note that PostGIS commands will not be covered in Postgres or generic SQL guides. Some useful PostGIS links include:
http://www.postgis.org/documentation/
http://www.postgis.org/support/wiki/
http://www.bostongis.org/

Also recommended is the O'Reilly book SQL Pocket Guide by Jonathan Gennick. This covers Postgres as well as Oracle, DB2, SQL Server and MySQL.

For those interested in comparing Postgres with other database products, Wikipedia has a useful article at: http://en.wikipedia.org/wiki/Comparison_of_relational_database_management_systems

Workshop details.

To participate in this workshop, you will need an account on the server. This account must also have

been given rights to create databases in a Postgres database cluster. This should have been set up for you before the workshop starts, but if you have any problems, please let the speaker know and they will be sorted out.

You will be provided with a connection to a PostGIS server from a workstation to carry out the tasks described in the workshop. More complex operations will generally already be written as scripts or SQL statements that you can run, so you will not need to type them in.

Commands that you type in will be written in italics in this document.

The workshop.

Scripts.

As the Linux servers at Greta Point are Linux based systems, the workshop will be focused on working in a Linux environment, and will utilise simple shell scripts to run SQL's. Some introduction to shell scripting is therefore included.

A Linux shell script is similar in principle to a batch file under Windows (or MSDOS). Essentially it comprises a list of commands to be carried out in order, much as you would type them in at the keyboard.

Some useful features of scripts will be used, notably using variables and redirecting input and output of commands from the screen to files or other commands.

Postgres on-line help within psql is available by typing \? for psql \ commands, or *h* for SQL commands.

PostGIS installation.

PostGIS can be installed on pretty much any modern computer running a wide range of operating systems. The Windows Postgres installer allows you to choose to add several Postgres utilities and extensions in addition to the core Postgres client and server.

It can be compiled from source on most architectures and operating systems, and for Linux, binary packages are available for both RPM and DEB style distributions.

A full installation includes several utilities and libraries, including GEOS, GDAL and Proj.4. These are needed for PostGIS to run, but further detail is beyond the scope of this workshop. Installation is generally straightforward for anyone who needs this functionality in their desktop.

For the purposes of this workshop, PostGIS is pre-installed on a local server, and you will be accessing a PostGIS database on this server.

Create your database.

Given that PostGIS is already installed, the first step will be to create a new PostGIS database. Under Windows, this is as simple as creating a new Postgres database, as Postgis will be installed automatically. Under Linux, it can be set up this way, but is generally not.

Note that when the Postgres server application is started, it connects to what is called a Postgres database cluster. All the databases accessed by this server will be stored within this cluster, but are managed as separate, independent entities. Multiple server applications can be run on the same server system, but they cannot communicate between each other.

There are four steps required to create a new PostGIS database.

- Create the Postgres database
- Install the language the PostGIS functions are written in
- Install the PostGIS functions/operators/datatypes
- Install the data supporting projection management

Note that to be able to create tables or databases, you will need to be granted this ability in the database. The database administrator has full control of who can access what from where in a Postgres database.

To create a database (assuming you have permission), ensure you are logged in and type:

createdb <user>_db
: use your name instead of <user> and this will create a new database. Then type

createlang <user>_db plpgsql
: this will install the plpgsql language in the database

psql -d <user>_db -f /tmp/db_workshop/lwpostgis.sql
: this will use the psql client to connect to the new database and run the SQL commands in the named file. These commands will load the PostGIS components into the database.

psql -d <user>-db -f /tmp/db_workshop/spatial_ref_sys.sql
: as you can guess, this runs the SQL commands to insert the data describing the various projections supported by PostGIS.

You now have your very own PostGIS database. You will be using it for all the exercises in this workshop. The first section will just look at Postgres, and how to insert and query data. Later on we will look at using spatial datatypes and functionality.

Exercise 1: Creating tables and loading data.

Creating tables.

You have just created a database and run files of pre-prepared SQL statements in that database to install PostGIS. To save typing (& typos), most of the SQL statements you will be running can be run from a pre-prepared script, but for now you can use psql to connect to the database and enter some simple queries. They will need to be simple, as there are no data in there to support anything else!

Connect to you database:

psql -d <user>_db

Your prompt will change, as you are no longer typing commands in to a terminal window to your operating system, but into the psql client, so your commands will be sent to the Postgres server, not the operating system.

Create a simple table, with three columns, an integer, numeric (decimal) and character column. The numeric column will support up to 7 significant digits with 3 decimals and the character column will permit variable length strings of up to 10 characters.

```
create table table1
(col1   integer,
 col2   numeric(7,3),
 col3   varchar(10)
);
```

Type *\d* to list the tables & other database contents on screen. (see the Postgres cheatsheet for more information, or type *\?* or *help* for online information about commands. You will see three tables, geometry_columns, spatial_ref_sys, both for PostGIS metadata, and the table you just created.

To view the definition of this table, type: */d table1*

Inserting data.

Of course, right now there is no data stored in this table, but you are about to change that. Type:
insert into table1 values (1,3.142, 'record1');

To retrieve this data type:

*select * from table1;*

The one record will be printed on screen.

Copying data from Empress.

There is a much faster way to insert data than using manual inserts, and that is the Postgres copy command. Which makes this a good time to introduce the concept of database transactions. A transaction is an atomic operation as far as the database is concerned. It comprises an intialisation, one or more operations, and a commit. The results of the operations are not saved until the transaction is committed, thus a failure with any single record will cancel the entire process for all records.

Normally, with a sequence of separate insert statements each insert is a stand alone transaction, which means the result of any insert failing will NOT impact on the others, which will still be inserted correctly. This creates considerable overhead, compared with running all the inserts as a single transaction. Both approaches have advantages & disadvantages.

The fastest way to insert data is using the copy command. This effectively streams a set of data directly into the table. A failure to load any part of the data will result in the transaction being aborted, with NONE of the data loaded.

This exercise will demonstrate a number of capabilities, including simple scripting, running SQL commands from the terminal, piping data between commands and running background processes under Linux. It is getting a bit ahead of things, but provides you with a set of data in Postgres tables to work with.

For Empress users who wish to load data from an Empress table into PostGIS, this provides a quick and easy solution. Both Empress and Postgres allow SQL commands to be run from the terminal command line. It is therefore possible to execute a query on an Empress table, and send the output directly to the Postgres table. Of course this means the data must be in a format supported by both RDBMS systems, and on a system with Empress and at least a Postgres client installed.

You should open a terminal session on neptune to start with. Depending on what operating system and software you are using, how you do this varies. On a linux system, you can generally open a terminal window & type *ssh -X <user>@<host>*, where <user> is your login and <host> is the system you want to connect to. On Windows you can use third party applications such as putty, X-win32 or X-deep. Putty is essentially text based, the other two will support a graphical interface.

The file you need to get to carry this out is called /tmp/db_workshop/postgres.sh. Copy this to your current directory: *cp /tmp/db_workshop/postgres.sh .*

This file (a bash shell script) will be used to run the next few exercises for you. It should save typing, & makes a useful example of some ways that scripts & SQLs can be combined to make tasks easier.

The first exercise, carried out for you by the script, will create tables in your new database to store some research trawl data, including some stratum, station, catch, length and biological data. It will then run SQL commands in Empress to extract the data, and write this data into your Postgres tables. Note that this involves no PostGIS functionality will be used, just basic Postgres & SQL.

Peruse the script (included as text in Appendix 1) and ensure you understand what it does and how it does it. You can run it multiple times with no problems, as it recreates the tables each time you run exercise 1.

To run it, just type *./postgres.sh* and press enter. To avoid being prompted for the name of your database every time, you can pass this in on the command line, ie: *./postgres.sh <user>*

Choose exercise 1 to create the tables & populate them with data extracted from the Empress trawl database, as well as species names and codes from the species_master table in the rdb database.

This is run twice, the first time on neptune to extract the data from Empress, the second time on neptune2 to load the extracted data into Postgres. In future, when Empress is running on neptune2, this will be simpler.

Note that the script drops the tables before creating them, so the first time it is run you will get an error message as you try to drop a table that doesn't (yet) exist.

Exercise 2: Simple queries from a single table.

Select, distinct, count(), group by.

Now you have some data, you can run exercise 2. Run the script as you did above but choose exercise 2. This runs several simple queries to illustrate some basic SQL commands. The SQL's run by the script are shown here, so you can type them in if you prefer. These are short and simple enough to type in, some later queries will be longer.

select trip_code from station;

Pretty obvious. But the station table has a record for every station in every trip. We just want to get each trip code once.

select distinct trip_code from station;

Much tidier, but you could have got the same result (if all is well with the data) by using

select trip_code from trip;

Next, you'll try an aggregate function to find out how many stations were recorded for each trip. This uses a function (count(*)) as well as the group by statement to determine what is counted.

select trip_code, count(*) from station group by trip_code;

The output has headings, which are not necessarily those we'd prefer to have. So you can rename them as you choose:

select trip_code, count(*) as number
from station
group by trip_code;

This sql runs over more than one line, simply because it makes longer statements easier to understand. You can write it all on one line if you prefer.

So now we have a list (with a renamed column) showing the number of stations from each trip. It is generally useful if such be lists are in some sort of order. Data in a relational database has no inherent order, so this has to be imposed explicitly in the SQL command

select trip_code, count(*) as number
from station
group by trip_code
order by trip_code;

Note that some databases (such as Empress) return ordered data from a group by operation automatically. This is NOT standard practice, and you cannot rely on this form of SQL working the same way elsewhere.

There are only a few trips in the result set, but say we just want the top three trips by number of stations. You can limit the number of records returned, by using, wait for it, limit:

```
select trip_code, count(*) as number
from station
group by trip_code
order by number desc
limit 3;
```

By default, ordering is done is ascending order, which obviously would not give us the desired result, so we specify the sort order to be descending using *desc*.

Lastly, at least for exercise 2, the where clause. This is used to restrict the records returned to those meeting specified criteria, which are always criteria based on the data itself. Assuming we only want information relating to a single trip returned by the query, we use a where clause in the select statement to accomplish this.

```
select trip_code,
        stratum,
        count(*) as number
from station
where trip_code='tan9801'
group by trip_code,
            stratum
order by stratum;
```

This is a good time to describe some conventions used in Postgres SQL statements. Strings are enclosed in single quotes: 'HOK', not double quotes. Column names will normally be converted to all lower case.

This is a far from complete introduction to querying a single table, but should be enough to give you an understanding of basic SQL syntax to work with and experiment further.

Exercise 3: Data integrity & indexes.

Primary keys.

A relational database, as mentioned earlier, stores information about relationships between objects, as well as the data pertaining to these objects. The trip table, displayed when you start exercise 3, stores some data describing some trips, but we have not discussed how these data relate to stations (and catches, etc), or even how we can set constraints on the database to ensure data errors are rejected. As data can only be queried (using select) to identify and access an individual record, we need to ensure that it has some unique combination of attributes which distinguish it from every other record in the table. This is called the primary key. While it is often implemented using the object's data attributes, it can be more efficient (and in some cases is required) to use a database key, such as a column of integers where every one is unique. This provides a quick and easy handle for each record.

None of the tables we are using have any keys (yet). A primary key is effectively a unique index built against columns of a table.

Start exercise 3, and you will see a listing of the trip table (the table definition, not its contents). The sql to do this is:

\d trip

This table has an obvious natural primary key, the trip_code, as there should only ever be one record in the table for any trip. So lets build it (from the script, or type).

alter table trip add primary key (trip_code);

Things get a bit trickier with the station table. The natural primary key is a combination of trip_code and station_no. To have the database maintain this rule, and only allow one entry of each station for each trip, the SQL is:

alter table station add primary key (trip_code, station_no);

Generally a primary key is created when the table is created, using the syntax:

```
create table trip
( trip_code      char(7)     primary key,
  ...
```

or where there is a composite key,

```
create table station
( trip_code char(7),
  ...
  max_gdepth  integer,
  primary key (trip_code, station_no));
```

Another common constraint is to require a column to always have a value. A trip record can be required to have both a start_date and finish date, for example:

```
alter table trip alter column date_s not null;
alter table trip alter column date_f not null;
```

For the purposes of this workshop, not null constraints are not required, therefore they are not implemented in the script, but they can be created by typing in these SQL statements if you desire.

We can now create primary keys on the catch and lgth tables. The fish_bio table cannot support a natural primary key. This is not good practice, so we will create an artificial key. The ideal natural key for fish_bio is a composite key of trip_code, station_no, (subcatch in the original as well, but ignored in this demonstration), species and fish_no. Unfortunately, as data can be entered on multiple workstations, it is possible to have duplicates of these, as workstation is not stored as a column.

This key is created using (script):

alter table fish_bio add column id serial;

The serial datatype is a useful Postgres datatype, which is shorthand for generating an automatically incrementing sequence of integers. By adding a new serial column, each record will be given a unique number. Note, however that a user is able to change these numbers, so to prevent such changes resulting in duplicates, a constraint can be added, in this case the column can be made a primary key, as above, or have a unique index created on it.

create unique index fish_bio_id_idx on fish_bio(id);

To generally improve performance, indices can be used on columns. To create an index on species_code, in the species_master table, for example, use:

create index sp_master_code_idx on species_master (code);

Foreign keys & referentials.

A foreign key is applied to tables to ensure referential integrity. This is perhaps best illustrated by an example. No record in the station table should be allowed to have a trip_code which does not exist in the trip table. i.e., all stations MUST belong to a recorded trip. This is implemented by creating a foreign key on station:trip_code referencing the trip:trip_code column. If this constraint is set, the database will not allow any station records to have a trip_code which is not present in the trip table.

Implementing this constraint is an example of defining the relationship between trips and stations explicitly in the database. It is created in the script by the SQL:

```
alter table station
add constraint station_trip_fk foreign key
                (trip_code) references trip (trip_code);
```

Again, these are generally implemented during table creation, using the SQL syntax:

```
create table station
( trip_code    char(7) not null references trip (trip_code),
  ...
  ...);
```

Foreign keys can reference multiple columns, so all lgth records should have a matching station record:

```
alter table lgth
add constraint lgth_trip_stn_fk (trip_code, station_no)
references station (trip_code, station_no);
```

Note that a foreign key can only reference columns which have a primary key on them in the referenced table.

The relationship defined by the foreign key is a many to one relationship, eg, for each trip there can be many stations, but each station can only come from (link to) one catch. Similarly, a given lgth record can only reference the one station, but one station can have many length records.

Exercise 4: joining tables and retrieving data.

Please re-run exercise 1 to re-build a clean database before running this exercise.

Simple joins.

Joins are used to extract data from more than one table. Usually primary & foreign keys are used to make the inter-table join. For example, to retrieve positions at which species were caught, with the weight caught, requires a join of the station table (for positions) to the catch table (for species). An SQL to accomplish this for just the species HOK is:

```
select s.trip_code,
        s.station_no,
        ((dlon_s + dlon_f)/2)::decimal(9,6) as lon,
        ((dlat_s + dlat_f)/2)::decimal(8,6) as lat,
        species,
        sum(c.weight) as weight
         from station s,
                 catch c
        where s.trip_code=c.trip_code
          and s.station_no=c.station_no
          and c.species='HOK'
        group by s.trip_code,
                    s.station_no,
                    lon,
                    lat,
                     species
         order by trip_code,
                    station_no;
```

This example introduces a few new aspects of SQL. It shows some simple arithmetic applied to values in the database (giving the midpoint between the start and finish positions), the "::" operator which causes a change in datatype, to, in this case, give a better formatted output, and the join between the two tables, based on the trip_code and station number. A join like this will output a record for every combination of matching station and catch records. It also introduces aggregate functions (sum) and group by, which defines how the data is to aggregated.

Of course, you may want to retrieve biological or length data as well as catch and station. This will require joining further tables, and requesting columns of data from them. While not required, it can be useful to order the output rows, as this allows a quick check of the output by eye.

Subqueries.

A subquery is a query within a query. For example, to find any stations from voyage tan9901 which did NOT catch HOK, we need a subquery, to select from the station table any stations which did not record HOK in the catch table. This also introduces the distinct operator, which removes duplicate output rows from the query result.

```
select station_no
from station
where trip_code='tan9901'
   and station_no not in (select distinct station_no
                          from catch
                          where trip_code='tan9901'
                             and species='HOK');
```

This provides a way to extract zero catch data from the database, by combining the output from two queries, one for stations with catch data and one without. The union of two queries is obtained using the union operator.

```
select s.trip_code
       s.station_no,
       s.min_gdepth,
       s.max_gdepth,
       c.species,
       sum(c.weight) as catch
from station s,
      catch c
where s.trip_code='tan9901'
   and c.trip_code=s.trip_code
   and c.station_no=s.station_no
   and c.species='HOK'
group by s.trip_code
          s.station_no,
          s.min_gdepth,
          s.max_gdepth,
          c.species

union

select trip_code,
       station_no,
       min_gdepth,
       max_gdepth,
       'HOK',
       0.0
from station
where trip_code='tan9901'
   and station_no not in (select distinct station_no
                          from catch
```

```
                    where trip_code='tan9901'
                      and species='HOK'

order by trip_code,
         station_no;
```

Outer joins.

The simple joins we have seen so far are actually called inner joins. With such joins, a row is returned for every occurrence of a matching record in all tables in the join. Thus, if there are no catch records for the selected species in any station, that station's data will not be retrieved. It is possible, as described above, to have multiple SQL statements merged using the union operator to achieve this, but SQL has an alternative approach: outer joins.

An outer join essentially carries out the inner join, then retrieves any requested data which were not returned using the inner join. Outer joins are available in left, right & full versions depending on which tables the extra records are required from. Ons difference from this approach and the above union is that with the union, the SQL explicitly provides values to be used in place of nulls, with outer joins, nulls are returned as nulls (unless you explicitly specify something else).

```
select s.trip_code,
       s.station_no,
       s.min_gdepth,
       s.max_gdepth,
      c.species,
      sum(c.weight) as "catch"
from station s
left join catch c on  c.trip_code=s.trip_code
                  and c.station_no=s.station_no
                  and c.species='HOK'
where s.trip_code='tan9901'
group by s.trip_code,
         s.station_no,
         s.min_gdepth,
         s.max_gdepth,
         c.species
order by s.trip_code,
         s.station_no;
```

Populating a series (creating data on a whim).

This is a bit more esoteric, but worth mentioning in the context of methods for filling nulls in a series. This example shows how to fill "holes" in a time series, using the Postgres generate_series command. Essentially, this generates an artificial sequence of values, concatenating the hours from the series with

a string representing the date. In the context of this workshop, don't expect to fully understand this immediately, it is only provided as an example of the sorts of data manipulation that are possible, even when there is no data.

select '2008-08-01 00:00:00'::timestamp+generate_series(0,23)'1 hour'::interval as hour;*

```
hour
---------------------
2008-08-01 00:00:00
2008-08-01 01:00:00
...
2008-08-01 23:00:00
```

For a more complete example, introducing the coalesce function, which returns the first non-null value in a list, such a hourly series can be used to fill missing timestamps in a data extract. This also illustrates the use of a select as a virtual table in a query, which is not possible in all RDBMS packages. This example selects values for one day, filling any null times with a value of 0.

```
select s.hour::int, coalesce(t.value,0)
from generate_series(0,23) s(hour)
left outer join
(select count(id) as value, extract(hour from start_time) as hour
from <table> where date_trunc('day',start_time) = '2008-08-01'
group by hour) as t
on s.hour = t.hour;
```

Indexing.

So far we have really only used indices (in exercise 3) to provide keys (a unique index). Indices are also used to speed up access to data. The last query takes only a few seconds, but is over a very small dataset by database standards, so we can get away without indexing. With no index, every record in every table must be scanned. With an index, the Postgres query optimiser can make informed guesses about how to carry out the query in the fastest way. We are comparing all trip_code and station_no values in three tables, as well as asking for just HOK catches and lengths. All these columns will be indexed, and the time taken to run it will be compared (script). It should be significantly faster, as shown in the script..

```
alter table trip add primary key (trip_code);
alter table station add primary key (trip_code, station_no);
create index station_trip_idx on station(trip_code);
create index station_station_idx on station(station_no);
create index catch_trip_idx on catch(trip_code);
create index catch_station_idx on catch(station_no);
create index catch_species_idx on catch(species);
create index lgth_trip_idx on lgth(trip_code);
create index lgth_station_idx on lgth(station_no);
create index lgth_species_idx on lgth(species);
```

Output formatting.

It is useful to be able to have an output more easily read into a spreadsheet or statistics package. There are a number of controls provided by Postgres to modify the output format. The output to date has been aligned into columns on the page, includes column headings and a count of lines output at the end. You have seen how you can run SQL commands from a shell script, and from within the psql command. Both of these offer ways to control the output format.

Showing the interactive commands first, (so this section will not be run from the script) enter the SQL command:

*select * from station limit 50;*

Type *\x* and run the same SQL again (use the arrows to navigate the command history). This gives an extended output format, much like that provided by the Empress list command. Type *\x* again to turn it off.

Now try *\t* and repeat the exercise. This is a toggle for returning tuples (data) only, or choosing whether to include the headings & record count lines in the output.

Next, use *\a* to toggle alignment into fixed width columns.

The last one to try here is *\f*, but this is slightly different. All the output you have seen so far uses the pipe character "|" as a field separator. This may not always be appropriate, and *\f* is used to change this, so it needs to be followed by the character you want to use. So, to create a comma separated file, use *\f ','* (remember strings are enclosed in single quotes). Note, this ONLY APPLIES FOR UNALIGNED OUTPUT. Aligned (formatted) output is unaffected, as it does not use a field separator character.

For a full list of all the commands starting with a \, type *\?* There are several more, not all of which are covered here. Type \h for help with SQL commands.

As mentioned, similar commands exist when invoking psql from the command line. To list these, from the shell (not psql) command line, type *psql --help*

These are most useful in writing scripts to generate output files to be read by other applications, and will be covered there.

Output redirection within psql.

While running psql, you can run an SQL from a file, or send output to a file, instead of to screen. This is useful to preview the results of a query to ensure they appear sensible before writing the file.

To send screen output to a file, type *\o <file>*, *\o* with no file will send output to screen. Similarly, to run an SQL stored in a file, use *\i <file>*.

Redirection from within a shell script is covered in the next exercise.

Exercise 5: shell scripts.

As you have seen from running the workshop script with the example SQL statements, it can be useful to store commands you can run to retrieve standard datasets. Scripts consist of a series of commands stored in a file, that you can invoke to tell the computer to carry out. Sort of a simple program. Several scripting languages work with Postgres, including Perl, Python & PHP, but the example here is a Linux shell script. Installing the free Cygwin package on Windows will allow shell scripts to run in that environment.

As you already know, you type *psql -d <db>* to connect to a Postgres database. Unlike some databases, such as Empress, where you use different commands for interactive access instead of running SQL 's from the command line, with Postgres, psql does it all, and command line parameters are used to tell it what to do.

From the shell command line, type *psql -l* to list the available databases on this server. To run an SQL command from the shell (for this workshop), type:

psql -d <user>_db -c "\d"

This connects to the -d database, and runs the command \d. To generate non-aligned, data only output from the previous query, but run from a script, try (type or from the script):

psql -d <user>_db -Atc "*select s.trip_code,*
s.station_no,
((dlon_s + dlon_f)/2)::decimal(9,6) as lon,
((dlat_s + dlat_f)/2)::decimal(8,6) as lat,
species,
sum(c.weight) as weight
from station s,
catch c
where s.trip_code=c.trip_code
and s.station_no=c.station_no
and c.species='HOK'
group by s.trip_code,
s.station_no,
lon,
lat,
species
order by trip_code,
station_no;"

The answer appears on screen, but can be redirected to a file in two ways. psql can be told, using the -o <filename> instruction, eg: psql -d <user>_db -o output.txt -Atc "...". Note that the alignment toggle is uppercase on the command line, but lowercase within psql (-a means something else).

The second way is to have the shell do the redirection instead of psql writing to the file. This is done using the ">" operator after the command. eg:

psql ...
    ...
    station_no;" > output.txt

This approach provides a great deal of power in terms of processing the output on the fly as it is created, using the pipe "|" operator. This takes the output of one command and sends it as input to another, so commands like sed, awk, tr, etc, can be applied to the data. If you look at the workshop script, when it populates the Postgres tables with data from the Empress trawl database, a pipe is used to redirect the output from the Empress empcmd SQL's into a Postgres psql command to write the data to the Postgres table.

It is useful to note that psql accepts parameters for the host that the Postgres server is running on, as well as the user name and password, so if you have access, you can retrieve data form a local psql client accessing a remote Postgres database anywhere on the internet.

Given that a shell script is an executable program, somehow the system needs to be told this so you can run it. There are two ways. As the shell used is the bash shell (one of the early shells was called sh, the Bourne shell, after its inventor, an enhanced version is called the Bourne again shell, hence bash), you can type bash <file>, so the system can start the bash interpreter & tell it which file to process. The second way is to make the script an executable program. This requires two steps, firstly you must change the mode of the file to allow it to be run using the chmod (change mode) command. Files can have separate permissions for the user who created it, groups & others.
chmod ugo=x <file> will allow everyone to run it. *chmod u=x* will allow just the user.

So now it is executable, but somehow the system needs to be told how to run it, whether it is a shell, perl, python, etc, script. To do this, the system will read the first line in the script, if it starts with #! then whichever program is named after the #! will be used, thus #! /bin/bash will mean run this script using the bash interpreter (the bash program in the /bin directory).

To illustrate this, we will now create and run a simple script to run an sql on a remote database.

Start an editor (emacs) to type in the script:

#! /bin/bash
psql -d antarctic -h otter -c "\d"

Press CTRL-X followed by CTRL-S to save the file (choose any name you like)

Now type:

chmod u+x <file>

This is now an executable script, but before you try, there is one more quirk involved in running it. When you type the name of a command, the system has a list of directories to look in to find it. This list (the search path) does not usually contain the current directory (where you just saved the file). So

running it will result in a message like "Command not found". You need to tell it where the command is. You may be familiar with the cd command to change directories, & how cd .. takes you up to the parent directory. "." is the current directory, ".." is its parent. So to run your new script, explicitly telling where to find the file, type:

./<file>

If everything has been done correctly, the system will look for the file <file> in the current directory, find it, check that it is executable, read the first line to find out how to run it, start the bash interpreter to run it, which will then start psql, access the antarctic database on the server otter, & return a list of all the tables there.

PostGIS.

As mentioned previously, PostGIS is a third party extension to Postgres. It provides OGC SFS compliant spatial data management capabilities within Postgres databases. This includes the facility for storing points, lines and polygons in database columns, as well as functions and operators for querying & manipulating spatial data. A useful PostGIS cheatsheet is available from Boston GIS at http://www.bostongis.com/postgis_quickguide.bqd. The full documentation is available at http://www.postgis.org. If you install Postgres on Windows using the standard installer, you can tick the PostGIS checkbox to include this as part of the standard install.

This session comprises a basic introduction to PostGIS using the databases used in the previous session on Postgres. These include stratum and station tables, both of which have spatial data. The strata are polygons, while stations can be thought of as a line between the start and finish points, as well as the start and finish points.

One of the main advantages of storing spatial data as spatial structures in a genuine database is that they can be plotted in maps with no import/export required, as long as your GIS/mapping tool can access PostGIS tables. PostGIS can be natively accessed by several Open Source GIS/mapping tools, including QGIS, JUMP, uDIG, gvSIG and others. A PostGIS database can also be used as a repository for spatial data, from where it can be extracted in a variety of formats, including Mapinfo, shapefiles and GMT files. PostGIS is supported by the current version of ArcMap as an ESRI geodatabase engine.

A demonstration of this using QGIS is included in this session, as is a script to plot a catch map for every species in the database using GMT. Such data can also be made available on the internet via OGC web services, and web catalogue services, which are also natively supported by mant desktop mapping & GIS applications.

The SQL's in this session are stored in the file postgis.sh. Run & use this file like the postgres.sh script in the previous session.

Recently a new naming convention has been applied to PostGIS functions. Historically, they were given names such as area(), buffer(), geomunion(), etc. This caused some ambiguities, as PostGIS was extended to support the SQL/MM standard as well as the OGC/SFS standard. To avoid confusion all PostGIS functions are now prefixed with "ST_". Thus within() becomes ST_within(). Given Postgres is case insensitive when it comes to function names "ST_" is the same as "st_". At present, to maintain support for existing scripts, etc, use of the old (non-ST) names is still supported. This is deprecated, however, and will not be continued indefinitely.

Installing PostGIS.

Before we can use PostGIS capabilities in the database, we need to install PostGIS in the Postgres database. There are four steps to this, as described previously. All are included in the postgres.sh script.

PostGIS is implemented using the plpgsql language, so you need to install support for this language in the database. This is done by the command *createlang plpgsql <db>*.You need dba access to do this. Then install the PostGIS functions by running a file of SQL statements, postgis.sql This is run via *psql -d <db> -f <path>/postgis.sql*, where path tells the psql command where to find the file. If you require projection support (recommended) then you will also need to run spatial_ref_sys.sql (*psql -d <db> -f*

<path>/spatial_ref_sys.sql). Under Windows, if you selected PostGIS when you installed Postgres, this is done automatically when you create your databases.

Creating geometry columns.

Once PostGIS is installed, the next exercise is to create geometry columns in the database for start and finish points, then take the start and finish latitude and longitude values in the station table and build start and finish points from these. Two approaches are shown below.

```
select addgeometrycolumn('','station','startp',4326,'POINT',2);
update station set startp=geometryfromtext('POINT('||dlon_s||' '||dlat_s||')',4326);
select addgeometrycolumn('','station','endp',4326,'POINT',2);
update station set endp=setsrid(makepoint(dlon_f,dlat_f),4326);
```

This introduces the Postgres SQL "||" operator. This concatenates two strings, and is an ANSI standard SQL operator, although not all databases (eg, MS SQL Server) follow this standard. The geometryfromtext() function builds a geometry object from an appropriate text string. Makepoint build a point feature from the X & Y values. Postgis still needs to be told what coordinate system the numbers represent, so the setsrid function is used to tell Postgis the srid (spatial reference identifier) to use for the point feature.

It also demonstrates the use of the PostGIS addgeometrycolumn() function. The parameters for this function are:

schema	defaults to public	
table	name of the table to add the column to	
name	name of the new column to add	
SRID	number of the spatial reference id (eg: lat/long=4326, NZMG=27200)	
type	geometry type, one of:	POINT (or MULTIPOINT)
		LINESTRING (or MULTILINESTRING)
		POLYGON (or MULTIPOLYGON)
dims	number of dimensions of each ordinate (lat/lon = 2)	

Having created the two new point geometry columns, the update statement uses the geometryfromtext function to insert a point feature, built up from the start lat & lon columns (in decimal degrees) using the concatenation operator. It also demonstrates the use of the PostGIS addgeometrycolumn function.

Next, we create a linestring column and take the points for each station to generate the line. Be aware, the line is between the start & fish points of the vessel, not the gear. Also, the actual track was probably not a straight line. Note. we could build a trackline populated with points at 1 minute intervals from the DAS database, but that is beyond an introductory workshop.

```
select addgeometrycolumn('','station','trackline',4326,'LINESTRING',2);
update station set trackline=makeline(startp, endp);
```

Given we have the doorspread and trackline, the next step is to create a polygon representing the estimated swept area of each station. This is done using the buffer function, which creates a buffer zone around a geometry. However, to do this we should use a projection which will not unduly distort the

area, so this will be deferred until we look at projections in a bit more detail.

The last geometry we have is that of the strata. The geometry from text function can build a binary geometry from a text string. While Empress cannot store a binary geometry, it can store the text which represents it, even though it cannot do anything with this. The strata polygons can be built from these text strings.

select addgeometrycolumn('','stratum','geom',4326,'POLYGON',2);
update stratum set geom=setsrid(geometryfromtext(stratum_txt),4326);

The new function here is setsrid. The WKT (Well Known Text) string stored in Empress does not have any projection information included. This command, therefore, creates a geometry of unknown projection from the WKT, sets the projection to that of SRID:4326 (lat/long WGS84), and stores this in the new polygon geometry column in the stratum table.

As with most RDBMS data, performance will generally be enhanced significantly if indexes are built on data columns. A spatial index is different to most text & numeric indexes, and is implemented in PostGIS as a Generalised Search Tree, or GIST index. Effectively this stores a bounding box of each geometry, and spatial searches check for the bounding box relationship before the actual geometries are compared.

Extracting spatial data.

While the station table does have the start and finish coordinates as numbers, this is not always the case, so this exercise will show how to retrieve numbers from spatial values.

```
select trip_code,
        startp,
        endp
from station
order by trip_code,
           station_no;
```

This returns the binary geometries, which is not very helpful. To see the values as text, which is somewhat useful for points, but less so for lines and polygons (they tend to comprise long strings of numbers which are not very readable), use the PostGIS astext() function.

```
select trip_code,
        astext(startp) as startp,
        astext(endp) as endp
from station
order by trip_code,
           station_no;
```

This is more readable, but still not very useful as a method of extracting the coordinates. We can retrieve the x & y coordinates using the x() and y() functions:

```
select trip_code,
```

```
        astext(startp) as startp,
        x(startp) as start_x,
        y(startp) as start_y
from station
order by trip_code,
            station_no;
```

Remember these can be extracted as un-aligned columns suitable for reading into spreadsheets or other software packages as discussed in the Postgres session. Now assume we only have the trackline geometry. As this comprises the points (and therefore coordinates, retaining the points and coordinates is redundant, but there is no need to remove them at this stage. An SQL to return the startpoint & its coordinates is:

```
select trip_code,
        station_no as stn,
        astext(trackline) as text,
        astext(startpoint(trackline)) as text,
        x(startpoint(trackline)) as x,
        y(startpoint(trackline)) as y
from station
order by trip_code,
            station_no;
```

One important point to note is that geometries are implemented as one or more X/Y coordinate pairs. This means they are normally in lon/lat order, not the maritime convention of lat/lon. You can store geometries with the sequence reversed, but don't expect them to plot in the right place on a map.

A quick example of some measurement functions, distance() and length(). You can get the length of a point geometry, but it will always be 0. Less obvious, but length is also pretty meaningless when applied to polygons, which will also have a length of 0. Linestringss do have a length. All PostGIS measurement functions return the measurement in the default units for the projection of that feature. Getting a length in degrees (the unit for SRID:4326) is not useful anywhere except along the equator. The unit for NZMG is metres, so for now we will use that, although it is increasingly inaccurate as the distance from NZ increases. (We will cover more of this in the exercise on projections).

By default, PostGIS does not implement any checks on the validity of spatial features (later versions will do more of this). It does, however, provide the means to check this, and to implement this check as a database constraint if required. The function is isvalid().

```
select trip_code,
        station_no
from station
where not isvalid(trackline);


select stratum_key,
        stratum_code
from stratum_def
where not isvalid(geom);
```

Any errors identified can be fixed manually, generally with some spatial editing tool such as QGIS (described later).

Projections 101.

PostGIS allows you to reproject geometries on the fly, using the transform() function. Thus to get the length of a trackline in both km & nm, and with just three decimal points:

```
select trip_code,
        station_no,
        (length(transform(trackline,27200))/1000.0)::decimal(6,3) as lgth_km,
        (length(transform(trackline,27200))/1852.0)::decimal(6,3) as lgth_nm
from t_station;
```

What this has done is to project the coordinates into NZMG, etc, to determine the measurement. A value in cartesian degrees as a length is not very useful. Any representation of relative spheroidal locations onto a flat sheet of paper (map) involves some distortion. There are three aspects to representing such data: shape, distance and size. Any projection of such data will distort at least one and often two of these. To generate reasonably accurate & consistent areas, an equal area projection is used, to maintain distances, an equidistant projection. A wide (and confusing) variety of projections exit to permit appropriate representation of all regions of the globe.

There are a few useful projections to be aware of working around NZ. These include:

- NZMG (New Zealand Map Grid) EPSG:27200
```
+proj=nzmg +lat_0=-41 +lon_0=173 +x_0=2510000 +y_0=6023150 +ellps=intl +datum=nzgd49 +units=m +no_defs
```
- NZTM2000 (New Zealand Transverse Mercator 2000) EPSG:2193
```
+proj=tmerc +lat_0=0 +lon_0=173 +k=0.999600 +x_0=1600000 +y_0=10000000 +ellps=GRS80
+towgs84=0,0,0,0,0,0,0 +units=m +no_defs
```
- NIWA Mercator (NIWA offset Mercator projection) EPSG:3752
```
+proj=merc +lat_0=-70 +lon_0=100 +k=1 +x_0=0 +y_0=0 +units=m +ellps=WGS84 +a=6378137
+b=298.2572235629972 +no_defs
```
- Local equal area (arbitrary Albers Equal Area)
```
+proj=aea +lat_1=-30 +lat_2=-50 +lat=-40 +lon_0=175 +x_0=0 +y_0=0 +ellps=WGS84 +datum=WGS84 +units=m
+no_defs
```
- SCAR Antarctic polar
```
+proj=stere +lat_0=-90 +lat_ts=-71 +lon_0=0  +x_0=0 +y_0=0 +ellps=WGS84 +datum=WGS84 +units=m
```
- NOAA Natice polar
```
+proj=stere +lat_0=-90 +lat_ts=-60 +lon_0=180 +x_0=0 +y_0=0 +ellps=WGS84 +datum=WGS84 +units=m
```

To create an instance of a projection in PostGIS you simply create a record in the spatial_ref_sys table. For example, to create an entry for the Albers Equal Area projection described above:

```
psql -d $DB -c "insert into spatial_ref_sys values
            (27201,
             'BAW',
             27201,
             '',
             '+proj=aea +lat_1=-30 +lat_2=-50 +lat=-40 +lon_0=175 +x_0=0 +y_0=0
+ellps=WGS84 +datum=WGS84 +units=m +no_defs';"
```

This is used later in the section on measurements.

That's about it as far as projections go for this workshop, if you want to learn more, try these:

http://www.colorado.edu/geography/gcraft/notes/mapproj/mapproj_f.html
http://mathworld.wolfram.com/MapProjection.html

For an in depth discussion including the math, from the original author of the projection library used by PostGIS, see: http://dl.maptools.org/dl/proj/PROJ.4.3.I2.pdf

The reference list of EPSG projections, et is available from:
http://www.epsg-registry.org/ (testing only at present)
http://www.epsg.org/

180 degrees

One problem affecting maritime GIS users in NZ is the conventional splitting of the world at 180°. Longitudes are conventionally expressed in degrees between -180° & 180°. This results in the part of the EEZ in the western hemisphere plotting at the left side of a world map, with the eastern hemisphere on the right. Coordinates can be reprojected to UTM or a local projection to cater for this situation, but it is often convenient to store data in degrees, with a range of 0°-360°, creating a 180-centric map, ideal for NZ. To support this, NIWA funded the implementation of a custom PostGIS function, shift-longitude. This function adds 360 to any longitudes less than 0 in any feature, providing a convenient way of working around the problem.

Note that the positions already loaded into the database are using a 0-360° longitude range. The next SQL shows the result of shifting (no reprojection is carried out) them to a standard 4326, (-180°-180°), via NZMG, and then of shifting them back. You will need to scroll down to find some longitudes > 180° to see the effect.

```
select trip_code as trip,
        station_no as stn,
        X(startp)::decimal(7,4) as raw_lon,
        Y(startp)::decimal(6,4) as raw_lat,
        X(transform((transform(startp,27200)),4326))::decimal(7,4) as lon_4326,
        Y(transform((transform(startp,27200)),4326))::decimal(6,4) as lat_4326,
        X(shift_longitude(transform((transform(startp,27200)),4326)))::decimal(7,4) as lon_shift,
        Y(shift_longitude(transform((transform(startp,27200)),4326)))::decimal(6,4) as lat_shift
from station
order by trip_code,
           station_no;
```

Note also, that the specification for the EPSG:4326 does not distinguish between +-180° and 0-360° longitude ranges. Any use of 0-360° degree longitudes is not guaranteed to work with all GIS software, given the defacto standard of +-180°. In PostGIS, EPSG:4326 supports both.

Point in polygon.

One of the more useful tasks PostGIS supports is identifying points lying within a polygon. One of the

main uses of this is to check data, but is can also be useful to, for example, identify the QMS or statistical area where a fishing event occurs.

There are two functions to do this, contains & within, but a brief discussion of spatial relationships is warranted here. For a full discussion, see: http://gis.hsr.ch/wiki/images/3/3d/9dem_springer.pdf

Essentially, note that a point has no area, so cannot overlap a polygon, however, the definition of a polygon (contains the point set of all points inside it"s boundary) means a polygon can contain a point. Similarly a line can cross, but not intersect a polygon or line. Issues like "is a point on the boundary of a polygon inside or outside?" are addressed by having specific functions for such cases. Note that one ambiguity exists in the standard - spatially equals. It is not specified in the OGC standard how this addresses lines or polygons with their vertices/endpoints in reverse sequence, or if polygons or linestrings with extra vertices which do not change the shape of the feature (ie: they lie directly between two vertices). Also note that for a point to lie on a line, the result is related to the calculated distance between the point & the interpolated line. Ideally this should be zero, but computers working with floating point values have an interesting interpretation of zero, yeilding some anomalies, and different answers to the same question between, for example, 32bit & 64bit systems. It is very much an issue of tolerances, rather than absolutes.

An example query is to return the calculated stratum a station lies within, with the recorded stratum.

```
select s.trip_code,
        s.station_no,
        s.stratum,
        m.stratum_code
from station s,
       stratum m,
       stratum_trip t
where s.stratum=t.stratum
  and t.stratum_key=m.stratum_key
  and contains(m.geom,s.trackline)
order by trip_code,
           station_no;
```

This uses the stratum_trip table (which defines a relationship, rather than stores data) to link the stratum geometry to the trip. Technically, this reduces a many to many relationship between strata and trips into two many to one relationships.

While this works well, what is more useful is to simply return those stations where the the position does not lie within the specified stratum, as an error check. This requires one extra line added to the above SQL, and when run, hopefully returns no records.

```
select s.trip_code,
        s.station_no,
        s.stratum,
        m.stratum_code
from station s,
       stratum m,
       stratum_trip t
where s.stratum=t.stratum
```

```
   and t.stratum_key=m.stratum_key
   and contains(m.geom,s.trackline)
   and s.stratum != m.stratum_code
order by trip_code,
         station_no;
```

As you can see, we now just return records where the specified and calculated stratum codes are different, with one extra criteria in the where clause of the sql.

Querying for measurements.

Given the basic understanding of projections covered above, it should be apparent that the way to retrieve an accurate measurement depends on doing the measuring in a suitable projection. Within PostGIS you can create custom or non-standard projections. What is needed is a unique number to identify the projection, and the parameters needed for the projection transformation. PostGIS, like several other applications dealing with spatial data, uses the Proj.4 application/library to do any reprojection required. So, as well as this unique id, PostGIS also stores the parameter Proj.4 needs. EPSG (the European Petroleum Survey Group) established a set of codes and standard projections, called EPSG codes. These are all supplied with PostGIS, and are stored in the table spatial_ref_sys, any new projections needed can be added to this table.

We need to create an equal area projection suitable for use around NZ. To do this, we'll add a custom equal area projection to the standard list.

```
insert into spatial_ref_sys
values ( 27201,
         'WOODB, NIWA',
         27201,
         null,
         '+proj=aea +lat_1=-30 +lat_2=-50 +lat=-40 +lon_0=175 +x_0=0 +y_0=0 +ellps=WGS84
+datum=WGS84 +units=m +no_defs');
```

The SRID for NZMG is 27200, so I arbitrarily chose 27201 for this projection. The last string entered contains the parameters required by the Proj.4 reprojection library used by PostGIS.

Having gone through this exercise, which shows the preferred way to measure areas, PostGIS has a couple of options which make life easier, at least for linear measurements. The functions distance_sphere, distance_spheroid and length_spheroid always return values in meters. They use a sphere (or spheroid, respectively) instead of the native spheroid/datum, so are generally not as accurate, but will often meet all that is needed for a reasonably accurate distance measurement. Note that the *_spheroid functions require the spheroid radius & offset to be given. There are no equivalents for areal measurements (yet), so these will always require reprojection.

To see the difference between spherical, spheroidal & projected distance measurements, try this modified version of the projection example SQL above:

```
select trip_code,
        station_no as stn,
        ST_length(trackline)::decimal(8,6) as native,
        (ST_length(trackline)*(60*1.852))::decimal(6,3) as native_km,
        (ST_length(ST_transform(trackline,27200))/1000)::decimal(6,3) as NZMG_km,
        (ST_length(ST_transform(trackline,27201))/1000)::decimal(6,3) as AEA_km,
        (ST_distance_sphere(startp, endp)/1000)::decimal(6,3) as sphere,
        (ST_length_spheroid(trackline,'SPHEROID[\"WGS84\",6378137,298.257223563]')/1000)::deci
mal(6,3) as spheroid
from station
order by trip_code,
            station_no;"
```

This returns six distances, cartesian degrees, cartesian degrees converted to km (1.852km/minute), km projected to NZMG, km projected to the custom AEA projections, km projected to a sphere and km projected to the WGS84 spheroid. Most of these give similar answers, all are correct answers, as far as they go. Which is most correct is debatable. NZMG has increasing distortion the further from land we go; the AEA is strictly an equal area projection, so has small inherent issues with distances, partly depending on the direction between the points and distance from the centre; spheroidal is likely to give a good estimate anywhere, but will vary depending on the spheroid used; spherical is quick but subject to some error as the world is not a sphere. The native cartesian degree value is highly dependent on the direction being measured and the distance from the equator, and is pretty definitely the least accurate.

When it comes to areas, the AEA projection will be reasonably reliable & consistent around New Zealand. To get the specified areas for each stratum, and the one calculated by PostGIS, using the new AEA projection and NZMG, try this sql.

```
select t.trip_code,
        s.stratum,
        s.area_km2,
        (area(transform(m.geom,27201))/1000000)::decimal(7,2) as AEA_area,
        (area(transform(m.geom,27200))/1000000)::decimal(7,2) as NZMG_area,
        (area(transform(m.geom,2193))/1000000)::decimal(7,2) as NZTM_area
from stratum s,
       stratum_def m,
       trip t,
       stratum_trip st
where st.stratum_key = m.stratum_key
   and s.stratum=st.stratum
   and st.trip_code = t.trip_code
   and s.trip_code=t.trip_code
order by trip_code,
            stratum;
```

Data aggregation.

All the usual SQL aggregate functions can be used with the PostGIS spatial ones, indeed several of the PostGIS functions are aggregate functions.

To get the total area of all strata for a trip, there are a couple of approaches. We can sum the areas of the strata, or we can combine the strata polygons and derive the area. Just to make things interesting, we can combine the polygons in two ways, by creating a new polygon which includes all the areas encompassed by the individual strata polygons, or my creating a single multipolygon from all the individual polygons. All three approaches will be demonstrated.

```
select t.trip_code,
       sum(s.area_km2),
       sum((area(transform(m.geom,27201))/1000000)::decimal(7,2)) as AEA_area,
       sum((area(transform(m.geom,27200))/1000000)::decimal(7,2)) as NZMG_area
from stratum s,
      stratum_def m,
      trip t,
      stratum_trip st
where st.stratum_key = m.stratum_key
  and s.stratum=st.stratum
  and st.trip_code = t.trip_code
  and s.trip_code=t.trip_code
group by t.trip_code
order by trip_code;
```

Of course there is a major problem with the result, remember some of the strata polygons failed to build correctly? These will have had area_km2 values entered, but will not have any polygon to aggregate in the result. Databases, in fact computers in general, are very good at giving you what you asked for, not what you think you asked for. In such situations, adding a non null constraint to the geom column would require that the data be present (but not that the value entered be accurate or correct). However, the sql still worked as it should.

We will now generate the area from a single multipolygon comprising all the separate strata polygons.

```
select t.trip_code,
       sum(s.area_km2),
       (area(transform(geomunion(buffer(m.geom,0.0)),27201))/1000000)::decimal(9,2) as AEA_area,
       (area(transform(geomunion(buffer(m.geom,0.0)),27200))/1000000)::decimal(9,2) as NZMG_area
from stratum s,
       stratum_def m,
       trip t,
       stratum_trip st
where st.stratum_key = m.stratum_key
  and s.stratum=st.stratum
  and st.trip_code = t.trip_code
  and s.trip_code=t.trip_code
group by t.trip_code
order by trip_code;
```

These values can be smaller that the others, but should not be. The reason for this is that any overlaps in the digitised polygons will only be counted once in the single merged polygon derived from the individual strata boundaries, but will be counted twice in the other approaches. Indeed, comparing the results from these is a useful technique for checking that strata polygons do not overlap, although it will

not help locate any slivers (gaps) between the polygons. (There are other ways to do this, such as comparing the total area of the constituent polygons with the area of the outer perimeter of the combined polygons).

Note the use of buffer() in this query. Overlaying nearly parallel lines can cause topological errors in the underlying geometry engine (GEOS). These can often be remedied by buffering the source polygons with a distance of zero, to generate cleaner congruent polygons to use in later operations. Later versions of PostGIS will include functions to fix topological errors in geometries.

Utility programs.

Several programs exist which facilitate working with Postgres and PostGIS.

As discussed before, the most common graphical client used with Postgres is PgAdminIII, which is bundled with the Windows Postgres installer, and is available as a package for Linux. PgAccess is also available for Linux clients, but has largely been superceded by PgAdminIII. If Postgres is installed as an ODBC (or JDBC) data source, then a variety of applications supporting this, such as MS Excel & MS Access can natively provide access to Postgres tables.

For the purposes of this workshop, the focus will be on PostGIS, rather than Postgres, as one of the most commonly required action to be undertaken with spatial data is to view it in a map context.

Spatial data repository.

PostGIS is very suited to use as a spatial data repository, as it is relatively easy to extract data in a variety of common GIS formats, for use with other software packages. Two of the utility programs supplied wiyh Postgis are pgsql2shape and shape2pgsql. These utilities can derive a shapefile from data held in PostGIS, or generate the SQL statements from a shapefile which will create and populate the equivalent PostGIS table. Typing either command with no parameters will provide usage information.

More advanced, and more versatile is the program ogr2ogr. This is part of the GDAL (Geographic Data Abstraction Library) package, maintained by Frank Warmerdam, who also maintains the Proj.4 package used by PostGIS (& others) for projection and datum shift operations. ogr2ogr supports a variety of formats, including PostGIS, shapefile, Mapinfo & GMT.

A simple example, building a shapefile of the tan9901 station trackline data from the PostGIS data:

ogr2ogr -f "ESRI Shapefile" -nln tan0802 tan0802 PG:dbname=stations -sql "select track, trip_code, station_no from station where trip_code='tan0802' and track notnull" -overwrite

This will create a new directory, tan0802, with the shapefile (tan0802.*) in it. The shapefile will contain the data requested in the SQL parameter. For Mapinfo users, try:

ogr2ogr -f "Mapinfo File" -nln tan0802 tan0802 PG:dbname=stations -sql "select track, trip_code, station_no from station where trip_code='tan0802' and track notnull" -overwrite

This does the same thing, but writes the Mapinfo .tab, .map & .dat files instead. Similarly, to produce a file for plotting with GMT (Generic Mapping Tools), use:

ogr2ogr -f "GMT" -nln tan0802 tan0802 PG:dbname=stations -sql "select track, trip_code, station_no from station where trip_code='tan0802' and track notnull" -overwrite

The file tan0802.gmt will be written. Note that this file contains metadata & attribute data as comments, as GMT cannot currently process these fields. Version 5 of GMT, due for release in 2011 will be able to use these values.

In a similar fashion, shapefiles, Mapinfo files, etc, can be inserted into PostGIS tables. Obviously

Format specific restrictions will apply, eg: a shapefile can only store a single geometry type, so in the above example, the sql specified the track was extracted, rather that the start or end points. To provide both points & lines in shapefile format will require two shapefiles.

Mapping applications.

As well as supporting the extract of data in formats suitable for use with common commercial desktop mapping and GIS applications, there are several Open Source applications available for plotting PostGIS data directly in maps. These include gvSIG, uDIG, the various JUMP packages,QGIS and GMT.

Commercial GIS applications such as ArcMap support the use of PostGIS as an underlying geodatabase engine using SDE.

gvSIG (Generalitat Valenciana, Sistema d'Informació Geogràfica) is a Spanish Java based GIS tool with native support for several common formats, and includes an English version. uDIG (User-friendly Deskto Internet GIS) is another Java based application, provided by the PostGIS developers. As it's name suggests, it has good support for internet based data supplied via Web Services, although given the increasing use of Web Services to supply such data for internet mapping clients, this caoability is pretty common in desktop mapping applications.JUMP (Java Unified Mapping Program) is another Java application, which has generated several versions including OpenJump & SkyJump.

The application I will focus on here is Quantum GIS (QGIS). Currently (04/2011) at version 1.6, it is under active development, and Gavin Macaulay, of NIWA, is one of the developers of this application. QGIS is available for Linux & Windows, and using GDAL/OGR as a data access library, it can natively open & display data in all the formats supported by these, including shapefiles and Mapinfo files.

QGIS can be used here to view data in the PostGIS database, as well as the Mapinfo files and shapefiles created above. QGIS can also be used as a heads up editing tool for these data, provided you have write access to the files or database tables.

QGIS can be downloaded from http://www.qgis.org, or contact IT support. Start it by typing qgis at the command prompt. Instructions for runnig QGIS are not given here, it is a relatively intuitive GUI based application.

GMT (Generic Mapping Tools - http://gmt.soest.hawaii.edu) is a suite of programs for gridding data, working with the grids and drawing maps. It is command line based, so generally invoked using scripts, and is somewhat arcane, but is capable of rendering very high quality cartographic output.

As both PostGIS tables and GMT vector formats are supported by OGR, it becomes straightforward to generate map series, for instance species occurrence maps drawn directly from the database, using scripts utilising the following approach:

```
LIST=`psql -d trawl -Atc "select distinct species from table;"

for SPECIES in LIST ; do

  ogr2ogr postgis -> GMT for SPP
  plot map
```

done

Given GMT can grid & contour data, it is also possible to largely automate the derivation of contours and rendering of them in maps using a similar approach.

A script to plot a single map of stations for a selected trip is included in Appendix 5. To get a copy to run: cp /tmp/.../gmt_demo .

Lastly, a note regarding web mapping applications. Modern web mapping applications typically use WMS (Web Map Service) and WFS (Web Feature Service) to access spatial data to plot on screen as a map. UMN mapserver and Geoserver are two mature Open Source tools for providing data from Postgis to WMS and WFS client applications. There are also tools to derive KML from PostGIS data for use with Google mapping services.

Appendix 1. Shell script for exercises 1-5

```
#! /bin/bash

# NOTE: the first line tells the operating system which command to use to run
#       this script. In this case this is a bash script, run by /bin/bash
# otherwise any line beginning with a # is treated as a comment

# copy.sh   bash script to demonstrate copying data from Empress to Postgres
#
# V1.0 B Wood, August 2008

# get database name

if [ "$1" = "" ] ; then
  echo -e "Target database: \c"
  read DB
else
  DB=$1
fi

# now check the database exists...
COUNT=`psql -l | grep $DB | wc -l`
if [ "$COUNT" = "0" ] ; then
  echo "$COUNT"
  echo "database $DB not found"
  exit
fi


echo "             Exercises"
echo "   1.    Create Postgres tables & copy Empress data"
echo "   2.    Simple SQL's for extracting data"
echo "   3.    Data integrity, keys and referentials"
echo "   4.    Joins and indices"
echo "   5.    Shell scripts"

echo ""
echo -e "       Choose exercise: \c"
read ANS
echo ""

# exit if nothing is selected
if [ "$ANS" = "" ] ; then
  exit
fi



# exercise 1
```

```
if [ "$ANS" = "1" ] ; then
  # build a Postgres table to load the data into

  psql -d $DB -c "drop table stratum cascade;"
  psql -d $DB -c "create table stratum
              (stratum_key    integer primary key,
               stratum_code   varchar(20),
               min_yr         integer,
               max_yr         integer,
               stratum_desc   varchar(256),
               stratum_txt    varchar(35000));"

  psql -d $DB -c "drop table stratum_trip;"
  psql -d $DB -c "create table stratum_trip
              (trip_code      char(7),
               stratum        char(4),
               stratum_key    integer);"


  psql -d $DB -c "drop table trip cascade;"
  psql -d $DB -c "create table trip
              ( trip_code    char(7),
                date_s       date,
                date_f       date,
                areas        varchar(24),
                mainspp      varchar(15));"


  psql -d $DB -c "drop table station cascade;"
  psql -d $DB -c "create table station
              ( trip_code    char(7),
                station_no   integer,
                stratum      char(4),
                date_s       date,
                time_s       integer,
                dlat_s       decimal(8,6),
                dlon_s       decimal(9,6),
                date_f       date,
                time_f       integer,
                dlat_f       decimal(8,6),
                dlon_f       decimal(9,6),
                min_gdepth   integer,
                max_gdepth   integer,
                dist_doors   decimal(4,1)
                );"


  psql -d $DB -c "drop table catch cascade;"
  psql -d $DB -c "create table catch
```

```
            ( trip_code    char(7),
              station_no   integer,
              species      char(3),
              weight       decimal(7,1));"

psql -d $DB -c "drop table lgth;"
psql -d $DB -c "create table lgth
            ( trip_code    char(7),
              station_no   integer,
              species      char(3),
              subcatch_no  integer,
              percent_samp decimal(5,2),
              lgth         integer,
              no_a         integer,
              no_m         integer,
              no_f         integer);"

psql -d $DB -c "drop table fish_bio cascade;"
psql -d $DB -c "create table fish_bio
            ( trip_code    char(7),
              station_no   integer,
              subcatch_no  integer,
              species      char(3),
              fish_no      integer,
              lgth         decimal(4,1),
              weight       float,
              sex          char(1),
              gonad_stage  char(1));"

# create a table for species names & codes
psql -d $DB -c "drop table species_master cascade;"
psql -d $DB -c "create table species_master
            ( code     char(3),
              com_name varchar(40),
              sci_name varchar(80));"



#define the Empress field separator character
export MSVALSEP="|"

# now let's load stratum data

echo "stratum data"
empcmd trawl "select stratum_key,
                stratum_code,
                year_from,
                year_to,
                stratum_desc,
                stratum_def
```

```
            from t_stratum_defn
            where stratum_def not like '%(0 41)%'dump;" | \
  psql -d $DB -c "copy stratum from STDIN with delimiter '|';"

  #stratum/trip data
  echo "stratum-trip data"
  empcmd trawl "select * from t_trip_stratum
            where trip_code like \"tan9%01\" dump;" | \
  psql -d $DB -c "copy stratum_trip from STDIN with delimiter '|';"


  # trip data
  echo "trip data"
q  empcmd trawl "select trip_code,
                  date_s,
                  date_f,
                  areas,
                  mainspp
            from trip
            where trip_code like \"tan9%01\" dump;" | \
  psql -d $DB -c "copy trip from STDIN with delimiter '|' null ";"


  # and some station data
  echo "station data"
  empcmd trawl "select trip_code,
                  station_no,
                  stratum,
                  date_s,
                  time_s,
                  dlat_s,
                  dlong_s,
                  date_f,
                  time_f,
                  dlat_f,
                  dlong_f,
                  min_gdepth,
                  max_gdepth,
                  dist_doors
            from station
            where trip_code like \"tan9%01\" dump;" | \
  psql -d $DB -c "copy station from STDIN with delimiter '|' null ";"


  # and species names
  echo "species code & name data from rdb"
  empcmd rdb "select code,
                com_name,
                sci_name
           from species_master dump;" | \
```

```
  psql -d $DB -c "copy species_master from STDIN with delimiter '|' null ";"


  echo "and the catch data"
  empcmd trawl "select trip_code,
                  station_no,
                  species,
                  weight
              from t_catch
              where trip_code like \"tan9%01\" dump;" | \
  psql -d $DB -c "copy catch from STDIN with delimiter '|' null ";"


  echo "the length data"
  empcmd trawl "select trip_code,
                  station_no,
                  species,
                  subcatch_no,
                  percent_samp,
                  lgth,
                  no_a,
                  no_m,
                  no_f
              from lgth
              where trip_code like \"tan9%01\" dump;" | \
  psql -d $DB -c "copy lgth from STDIN with delimiter '|';"

  echo "the biological data"
  empcmd trawl "select trip_code,
                  station_no,
                  subcatch_no,
                  species,
                  fish_no,
                  lgth,
                  weight,
                  sex,
                  gonad_stage
              from fish_bio
              where trip_code like \"tan9%01\" dump;" | \
  psql -d $DB -c "copy fish_bio from STDIN with delimiter '|' null ";"

  #end of exercise 1
fi



if [ "$ANS" = "2" ] ; then
  # start of exercise 2
  # single table queries
  # select out some data from the station table
```

```
# simple select
echo "a simple select of trip codes"
psql -d $DB -c "select trip_code
                from station;"

echo "Press Enter to continue"
read GO

# use distinct to drop duplicates
echo "retrieve unique (distinct) trip codes"
psql -d $DB -c "select distinct trip_code
                from station;"

echo "Press Enter to continue"
read GO

# getting counts
echo "getting counts of records, an example of agggregating"
psql -d $DB -c "select trip_code,
                count(*)
           from station
           group by trip_code;"

echo "Press Enter to continue"
read GO

# labeling output
echo "renaming output"
psql -d $DB -c "select trip_code,
                count(*) as number
           from station
           group by trip_code;"

echo "Press Enter to continue"
read GO

# sorting output
echo "ordering output"
psql -d $DB -c "select trip_code,
                count(*) as number
           from station
           group by trip_code
           order by trip_code;"

echo "Press Enter to continue"
read GO

# top n records
echo "limiting number of records returned"
psql -d $DB -c "select trip_code,
```

```
            count(*) as number
        from station
        group by trip_code
        order by number desc
        limit 3;"

  # a where clause
  echo "a where clause to get counts of stations by stratum for a single trip"
  psql -d $DB -c "select trip_code,
            stratum,
            count(*) as number
        from station
        where trip_code='tan9801'
        group by trip_code,
            stratum
        order by stratum;"

  echo "Press Enter to continue"
  read GO

  echo "exercise 2 done"
  # end of exercise 2
fi


if [ "$ANS" = "3" ] ; then
  # start of exercise 3
  # table keys
  echo "an introduction to keys & referentials"
  echo ""

  echo "the trip table:"
  psql -d $DB -c "\d trip"

  echo "Press Enter to continue"
  read GO

  # create a primary key
  echo "create a primary key on trip"

  psql -d $DB -c "alter table trip add primary key (trip_code);"

  echo "Press Enter to continue"
  read GO

  # create primary key on station
  echo "create a primary key on station"

  psql -d $DB -c "alter table station add primary key (trip_code, station_no);"
```

```
  echo "Press Enter to continue"
  read GO


  # add keys on other tables
  echo "adding primary keys on fish_bio"

  psql -d $DB -c "alter table fish_bio add column id serial;"
  psql -d $DB -c "create unique index fish_bio_id_idx on fish_bio(id);"

  echo "Press Enter to continue"
  read GO


  # add a foreign key on station:trip
  echo "creating a foreign key"

  psql -d $DB -c "alter table station
                add constraint station_trip_fk foreign key
                   (trip_code) references trip (trip_code);"

  echo "Press Enter to continue"
  read GO


  # foreign key on multiple columns
  echo "creating multiple column foreign key on catch, lgth & bio tables"

  psql -d $DB -c "alter table catch
                add constraint catch_trip_stn_fk foreign key
                   (trip_code,station_no)
                   references station (trip_code, station_no);"

  psql -d $DB -c "alter table lgth
                add constraint lgth_trip_stn_fk foreign key
                   (trip_code, station_no, species)
                   references catch (trip_code, station_no, species);"

  echo "end of exercise 3"
  # end of exercise 3
fi




if [ "$ANS" = "4" ] ; then
  # start of exercise 4

  # join station & catch
```

```
echo "a simple join on station & catch to retrieve species & position"
psql -d $DB -c "select s.trip_code,
                      s.station_no,
                      ((dlon_s + dlon_f)/2)::decimal(9,6) as lon,
                      ((dlat_s + dlat_f)/2)::decimal(8,6) as lat,
                      species,
                      sum(c.weight) as weight
                from station s,
                     catch c
                where s.trip_code=c.trip_code
                  and s.station_no=c.station_no
                  and c.species='HOK'
                group by s.trip_code,
                         s.station_no,
                         lon,
                         lat,
                         species
                order by trip_code,
                         station_no;"

echo "Press Enter to continue"
read GO

# simple joins with union to simulate outer join
# start off with subquery example
echo "an example of a subquery"
psql -d $DB -c "select station_no
                from station
                where trip_code='tan9901'
                  and station_no not in (select distinct station_no
                                         from catch
                                         where trip_code='tan9901'
                                           and species='HOK');"



echo "Press Enter to continue"
read GO


# subquery & union
echo "subquery with union"
psql -d $DB -c "select s.trip_code,
                      s.station_no,
                      s.min_gdepth,
                      s.max_gdepth,
                      c.species,
                      sum(c.weight) as "catch"
                from station s,
                     catch c
```

```
            where s.trip_code='tan9901'
              and c.trip_code=s.trip_code
              and s.station_no=c.station_no
              and c.species='HOK'
            group by s.trip_code,
                    s.station_no,
                    s.min_gdepth,
                    s.max_gdepth,
                    c.species

            union

            select trip_code,
                   station_no,
                   min_gdepth,
                   max_gdepth,
                   'HOK',
                   0.0
            from station
            where trip_code='tan9901'
              and station_no not in (select distinct station_no
                                     from catch
                                     where trip_code='tan9901'
                                       and species='HOK')

            order by trip_code,
                    station_no;"


echo "Press Enter to continue"
read GO


echo "left outer join"
# example of a left outer join to do the same as the above SQL

psql -d $DB -c "select s.trip_code,
                    s.station_no,
                    s.min_gdepth,
                    s.max_gdepth,
                    c.species,
                    sum(c.weight) as "catch"
            from station s
            left join catch c on  c.trip_code=s.trip_code
                              and c.station_no=s.station_no
                              and c.species='HOK'
            where s.trip_code='tan9901'
            group by s.trip_code,
                    s.station_no,
                    s.min_gdepth,
```

```
                s.max_gdepth,
                c.species
        order by s.trip_code,
                s.station_no;"


echo "ensuring all indices are dropped"

#drop any indexes to run intial timer
psql -d $DB -qc "drop index trip_trip_idx;"
psql -d $DB -qc "drop index station_trip_idx;"
psql -d $DB -qc "drop index station_station_idx;"
psql -d $DB -qc "drop index catch_trip_idx;"
psql -d $DB -qc "drop index catch_station_idx;"
psql -d $DB -qc "drop index catch_species_idx;"
psql -d $DB -qc "drop index lgth_trip_idx;"
psql -d $DB -qc "drop index lgth_station_idx;"
psql -d $DB -qc "drop index lgth_species_idx;"


#run 3 table join with timer
echo "time to run unindexed:"
time psql -d $DB -c "select s.trip_code,
                s.station_no,
                ((dlon_s + dlon_f)/2)::decimal(9,6) as lon,
                ((dlat_s + dlat_f)/2)::decimal(8,6) as lat,
                c.species,
                sum(c.weight) as weight,
                l.lgth,
                sum(no_m) as males,
                sum(no_f) as females,
                sum(no_a-(no_m+no_f)) as unsexed
        from station s,
            catch c,
            lgth l
        where s.trip_code=c.trip_code
          and s.station_no=c.station_no
          and l.trip_code=s.trip_code
          and l.station_no=s.station_no
          and c.species='HOK'
          and l.species=c.species
        group by s.trip_code,
                s.station_no,
                lon,
                lat,
                c.species,
                lgth
        order by trip_code,
```

```
                station_no,
                species,
                lgth;"  > /dev/null


echo "Press Enter to continue"
read GO

echo "creating indices"
psql -d $DB -c "alter table trip add primary key (trip_code);"
psql -d $DB -c "alter table station add primary key (trip_code, station_no);"

psql -d $DB -qc "create index station_trip_idx on station(trip_code);"
psql -d $DB -qc "create index station_station_idx on station(station_no);"
psql -d $DB -qc "create index catch_trip_idx on catch(trip_code);"
psql -d $DB -qc "create index catch_station_idx on catch(station_no);"
psql -d $DB -qc "create index catch_species_idx on catch(species);"
psql -d $DB -qc "create index lgth_trip_idx on lgth(trip_code);"
psql -d $DB -qc "create index lgth_station_idx on lgth(station_no);"
psql -d $DB -qc "create index lgth_species_idx on lgth(species);"

psql -d $DB -c "vacuum full;"

echo "time to run indexed:"
time psql -d $DB -c "select s.trip_code,
                s.station_no,
                ((dlon_s + dlon_f)/2)::decimal(9,6) as lon,
                ((dlat_s + dlat_f)/2)::decimal(8,6) as lat,
                c.species,
                sum(c.weight) as weight,
                l.lgth,
                sum(no_m) as males,
                sum(no_f) as females,
                sum(no_a-(no_m+no_f)) as unsexed
            from station s,
                catch c,
                lgth l
            where s.trip_code=c.trip_code
              and s.station_no=c.station_no
              and l.trip_code=s.trip_code
              and l.station_no=s.station_no
              and c.species='HOK'
              and l.species=c.species
            group by s.trip_code,
                s.station_no,
                lon,
                lat,
                c.species,
                lgth
            order by trip_code,
```

```
                    station_no,
                    species,
                    lgth;"  > /dev/null

  # end of exercise 4
fi


# for exercise 5, shell scripts, all notes are in the guide
#  this entire file is a collection of examples
```

Appendix 2. PostGIS script.

```
#! /bin/bash
#
#

# get database name

if [ "$1" = "" ] ; then
  echo -e "Database: "
  read DB
else
  DB=$1
fi

# now check the database exists...
COUNT=`psql -l | grep $DB | wc -l`
if [ "$COUNT" = "0" ] ; then
  echo "$COUNT"
  echo "database $DB not found"
  exit
fi

H=neptune
H=woodb-ws


echo "            Exercises"
echo "   1.   Creating & populating geometry columns"
echo "   2.   Extracting data"
echo "   3.   Introduction to projections"
echo "   4.   180 degrees"
echo "   5.   Point in polygon"
echo "   6.   Measuring distances & areas"
echo "   7.   Aggregating data"



echo ""
echo -e "       Choose exercise: \c"
read ANS
echo ""

# exit if nothing is selected
if [ "$ANS" = "" ] ; then
  exit
fi

# exercise 1
```

```
if [ "$ANS" = "1" ] ; then
  # create & populate geometry columns

  echo "crerating start/finish points"
  psql -d $DB -h $H -c "select addgeometrycolumn('','station','startp',4326,'POINT',2);"
  psql -d $DB -h $H -c "update station set startp=geometryfromtext('POINT('||dlon_s||' '||
dlat_s||')',4326);"
  psql -d $DB -h $H -c "select addgeometrycolumn('','station','endp',4326,'POINT',2);"
  psql -d $DB -h $H -c "update station set endp=geometryfromtext('POINT('||dlon_f||' '||dlat_f||')',4326);"

  echo "Press Enter to continue"
  read GO

  psql -d $DB -h $H -c "select addgeometrycolumn('','station','trackline',4326,'LINESTRING',2);"
  psql -d $DB -h $H -c "update station set trackline=makeline(startp, endp);"

  echo "Press Enter to continue"
  read GO

  echo "generate stratum polygons"
  psql -d $DB -h $H -c "select addgeometrycolumn('','stratum_def','geom',4326,'POLYGON',2);"

  #get list of strata to update
  #  done individually coz of errors
  psql -d $DB -h $H -Atc "select stratum_key
                from stratum_def;" > strata.txt

  while read STRATUM ; do
    psql -d $DB -h $H -c "update stratum_def
                set geom=setsrid(geometryfromtext(stratum_txt),4326)
                where stratum_key=$STRATUM;"
  done < strata.txt

  echo "End of exercise 1"
  exit
fi

if [ "$ANS" = "2" ] ; then
  echo "Extracting spatial data"

  psql -d $DB -h $H -c "select trip_code,
                      startp,
                      endp
                from station
                order by trip_code,
                        station_no;"

  echo "Press Enter to continue"
  read GO
```

```
psql -d $DB -h $H -c "select trip_code,
                    astext(startp) as startp,
                    astext(endp) as endp
                from station
                order by trip_code,
                      station_no;"

echo "Press Enter to continue"
read GO

psql -d $DB -h $H -c "select trip_code,
                    astext(startp) as startp,
                    x(startp) as start_x,
                    y(startp) as start_y
                from station
                order by trip_code,
                      station_no;"

echo "Press Enter to continue"
read GO

psql -d $DB -h $H -c "select trip_code,
                     station_no as stn,
                     astext(trackline) as line_txt,
                     astext(startpoint(trackline)) as point_txt,
                     x(startpoint(trackline)) as x,
                     y(startpoint(trackline)) as y
                 from station
                 order by trip_code,
                       station_no;"


echo "Press Enter to continue"
read GO

echo "invalid station tracklines"
psql -d $DB -h $H -c "select trip_code,
                         station_no
                    from station
                    where not isvalid(trackline);"


echo "invalid strata"
psql -d $DB -h $H -c "select stratum_key,
                         stratum_code
                    from stratum_def
                    where not isvalid(geom);"


echo "end of exercise 2"
```

```
  exit
fi

if [ "$ANS" = "3" ] ; then
  echo "introduction to projections"

  psql -d $DB -h $H -c "select trip_code,
                        station_no as stn,
                        length(trackline)::decimal(10,8) as native,
                        (length(transform(trackline,27200))/1000)::decimal(6,3) as km,
                        (length(transform(trackline,27200))/1852)::decimal(6,3) as nm
                from station
                order by trip_code,
                         station_no;"


  echo "end of exercise 3"
  exit
fi

if [ "$ANS" = "4" ] ; then
  echo "180 degrees"

psql -d $DB -h $H -c "select trip_code as trip,
                  station_no as stn,
                  X(startp)::decimal(7,4) as raw_lon,
                  Y(startp)::decimal(6,4) as raw_lat,
         X(transform((transform(startp,27200)),4326))::decimal(7,4) as lon_4326,
         Y(transform((transform(startp,27200)),4326))::decimal(6,4) as lat_4326,
         X(shift_longitude(transform((transform(startp,27200)),4326)))::decimal(7,4) as lon_shift,
         Y(shift_longitude(transform((transform(startp,27200)),4326)))::decimal(6,4) as lat_shift
              from station
              order by trip_code,
                       station_no;"



  echo "end of exercise 4"
  exit
fi


if [ "$ANS" = "5" ] ; then
  echo "Point in polygon queries"

  psql -d $DB -h $H -c "select s.trip_code,
                        s.station_no,
                        s.stratum,
                        m.stratum_code
                from station s,
```

```
                stratum_def m,
                stratum_trip t
            where s.stratum=t.stratum
              and t.stratum_key=m.stratum_key
              and contains(m.geom,s.trackline)
            order by trip_code,
                  station_no;"

  echo "Press Enter to continue"
  read GO
  echo "return mismatches only"

  psql -d $DB -h $H -c "select s.trip_code,
                  s.station_no,
                  s.stratum,
                  m.stratum_code
            from station s,
                stratum_def m,
                stratum_trip t
            where s.stratum=t.stratum
              and t.stratum_key=m.stratum_key
              and contains(m.geom,s.trackline)
              and s.stratum != m.stratum_code
            order by trip_code,
                  station_no;"



  echo "end of exercise 5"
  exit
fi



if [ "$ANS" = "6" ] ; then
  echo "spatial measurements"

  psql -d $DB -h $H -c "delete from spatial_ref_sys where srid=27201;"
  psql -d $DB -h $H -c "insert into spatial_ref_sys
                  values ( 27201,
                        'WOODB, NIWA',
                        27201,
                        null,
                        '+proj=aea +lat_1=-30 +lat_2=-50 +lat=-40 +lon_0=175 +x_0=0 +y_0=0
+ellps=WGS84 +datum=WGS84 +units=m +no_defs');"

  echo "Press enter to continue"
  read GO
```

```
  psql -d $DB -h $H -c "select trip_code,
                      station_no as stn,
                      length(trackline)::decimal(10,8) as native,
                      (length(transform(trackline,27200))/1000)::decimal(6,3) as \"NZMG_km\",
                      (length(transform(trackline,27201))/1000)::decimal(6,3) as \"AEA_km\",
        (distance_sphere(startpoint(trackline), endpoint(trackline))/1000)::decimal(6,3) as sphere,
        (length_spheroid(trackline,'SPHEROID[\"WGS
84\",6378137,298.257223563]')/1000)::decimal(6,3) as spheroid
                from station
                order by trip_code,
                        station_no;"


  echo "Press enter to continue"
  read GO

  psql -d $DB -h $H -c "select t.trip_code,
                          s.stratum,
                          s.area_km2,
                          (area(transform(m.geom,27201))/1000000)::decimal(7,2) as AEA_area,
                          (area(transform(m.geom,27200))/1000000)::decimal(7,2) as NZMG_area
                      from stratum s,
                          stratum_def m,
                          trip t,
                          stratum_trip st
                      where st.stratum_key = m.stratum_key
                        and s.stratum=st.stratum
                        and st.trip_code = t.trip_code
                        and s.trip_code=t.trip_code
                      order by trip_code,
                              stratum;"



  echo "end of exercise 6"
fi


if [ "$ANS" = "7" ] ; then
  echo "aggregating data"

  psql -d $DB -h $H -c "select t.trip_code,
                          sum(s.area_km2),
                          sum((area(transform(m.geom,27201))/1000000)::decimal(7,2)) as AEA_area,
                          sum((area(transform(m.geom,27200))/1000000)::decimal(7,2)) as NZMG_area
                      from stratum s,
                          stratum_def m,
                          trip t,
                          stratum_trip st
                      where st.stratum_key = m.stratum_key
```

```
                  and s.stratum=st.stratum
                  and st.trip_code = t.trip_code
                  and s.trip_code=t.trip_code
                group by t.trip_code
                order by trip_code;"

  echo "press enter to continue"
  read GO

  psql -d $DB -h $H -c "select t.trip_code,
                        sum(s.area_km2),
                        (area(transform(collect(m.geom),27201))/1000000)::decimal(9,2) as AEA_area,
                        (area(transform(collect(m.geom),27200))/1000000)::decimal(9,2) as NZMG_area
                from stratum s,
                    stratum_def m,
                    trip t,
                    stratum_trip st
                where st.stratum_key = m.stratum_key
                  and s.stratum=st.stratum
                  and st.trip_code = t.trip_code
                  and s.trip_code=t.trip_code
                group by t.trip_code
                order by trip_code;"



  echo "press enter to continue"
  read GO

  psql -d $DB -h $H -c "select t.trip_code,
                        sum(s.area_km2),
                        (area(transform(geomunion(buffer(m.geom,0.0)),27201))/1000000)::decimal(9,2) as
AEA_area,
                        (area(transform(geomunion(buffer(m.geom,0.0)),27200))/1000000)::decimal(9,2) as
NZMG_area
                from stratum s,
                    stratum_def m,
                    trip t,
                    stratum_trip st
                where st.stratum_key = m.stratum_key
                  and s.stratum=st.stratum
                  and st.trip_code = t.trip_code
                  and s.trip_code=t.trip_code
                group by t.trip_code
                order by trip_code;"

  echo "press enter to continue"
  read GO

  echo "end of exercise 7"
```

fi

Appendix 3. Sample GMT script

```
#! /bin/bash
#
# gmt_demo
#           script getting data from Postgis & plotting with GMT
#
#  v1.0 B Wood Augaust 2008

DB=baw

# get list of trips...

echo "Please type in the trip code you want plotted, available trips are:"
echo ""

psql -d $DB -c "select distinct trip_code
              from trip order by trip_code;" | grep -v rows

echo ""
echo -e "  Trip code: \c"

read TRIP

# check it exists....
COUNT=`psql -d $DB -Atc "select count(*) from station
                    where trip_code = '$TRIP';"`

if [ "$COUNT" = "0" ] ; then
  echo "trip $TRIP not found"
  exit
else
  echo "Plotting $COUNT stations"
fi


# extract station start points for plotting...
#   easier not to use ogr2ogr for points

rm -f trip.gmt
psql -d $DB -Atc "select X(startp),
                    Y(startp)
                from station
                where trip_code='$TRIP'
                  and startp notnull;" | tr "|" " " > trip.gmt


# get/set data extent
minmax -C trip.gmt > extent.txt
```

```
# read extent
while read W E S N ; do
  break
done < extent.txt

echo "$W $E $S $N"

N=`echo "$N + ( 0.08 * ( $N - $S ) )" | bc`
S=`echo "$S - ( 0.08 * ( $N - $S ) )" | bc`
E=`echo "$E + ( 0.09 * ( $E - $W ) )" | bc`
W=`echo "$W - ( 0.09 * ( $E - $W ) )" | bc`

echo "$W $E $S $N"

TY=`echo "$N + ( 0.06 * ( $N - $S ) )" | bc`
TX=`echo "( $E + $W ) / 2" | bc`

echo "$TX $TY"

# set R to region
R=${W}/${E}/${S}/$N

# define lat/long annot format
gmtset PLOT_DEGREE_FORMAT dddF

# get clever with annot spacing if required....

MAP=${TRIP}.ps

# plot simple basemap & coast
pscoast -R$R -Dh -W2 -JM6 -B4/2::WeSn -K > $MAP

#plot contours
psxy /home/woodb/plot_spp/nzbathy.gmt -R -W1 -J -M -O -K >> $MAP

#add label
echo "$TX $TY 10 0 1 MC $TRIP" | pstext -R -J -N -O -K >> $MAP

# plot points
psxy trip.gmt -R -J -W2/255/0/0 -Sc0.05 -O >> $MAP

# convert to raster
#ps2raster -A -Tg

# lets see...
ggv $MAP
```

Appendix 4. Postgres introduction.

This is intended as a guide for simple queries for staff intending to extract data from databases, but not create databases or tables, or insert their own data into tables. It is a simplifieed version, adapted for Postgres, of Susan Ng's 1993 Empress guide, at http://seaspray.niwa.co.nz/stockmon/database%20document/Guide%20to%20EmpressSQL.pdf

For further information about Postgres, see: http://www.postgresql.org/docs/

Introduction.

Postgres is a Relational DataBase Management System (RDBMS). An RDBMS is a tool for managing data stores in relations (tables) with rows (records) and columns (attributes). Data is manipulated using a well defined language called Structured Query Language (SQL).

Getting started.

The interactive client to Postgres is *psql.* This guide will use the trawl database as an example. connect to this database, use psql, and tell it which database to connect to using the -d parameter:

psql -d trawl

Postgres will respond with a copyright message, instructions on how to find some online help and the prompt (<database>=#):

Welcome to psql 8.2.4, the PostgreSQL interactive terminal.

Type: \copyright for distribution terms
\h for help with SQL commands
\? for help with psql commands
\g or terminate with semicolon to execute query
\q to quit

trawl=#

SQL commands are terminated with a semi-colon (;), and can be entered over multiple lines, until a semi-colon is entered. The prompt will change, while entering a multi-line SQL to <database>-#, replacing the "=" with a "-".

Any typing mistakes will generally NOT be recognised until the semi-colon is entered, because until then, Postgres does not try to carry out the command. Any such errors will generate a message like:
ERROR: syntax error at or near "kk"
which gives you some idea of the text near which the error lies.

Any output taking up more than one screen will be displayed in a buffer which you can use the up/down arrow keys, and the PgUp/PgDn keys to navigate. Press *q* to exit this buffer, and return to the psql command line. Thus

select * from catch;

is equivalent to:

```
select
*
from
catch;
```

The previous command can be retrieved using the up/down keys from the psql command line. Any desired changes can be edited in and the command run again.

To exit Postgres, as per the help given when you connect, type *\q*

Displaying tables.

Each database is organised into tables usually related to one subject area. Tables comprise attributes (columns) defined by their names, datatypes and constraints, which are retrieved as rows (records).

\d is used to display the tables (and some other structures) in a database. *\d <table>* will display the list of columns and constraints making up the table. There are several variants of \d to display further information about database objects. You can generally append a "+" to a \d command for more detailed output. Use \? from the psql command line for information.

Datatypes.

When tables are created, each column is defined as a specific datatype, typically characters or numbers, but date & time datatypes are also supported. For a full list of supported datatypes, see: http://www.postgresql.org/docs/8.2/static/datatype.html

The most commonly used datatypes are described below.

NUMERIC(n,d) (DECIMAL(n,d)	a formatted decimal number value, with n digits, d of which are right of the decimal point: decimal(3,2) (3.25)
SMALLINT	2-byte integer (-32,768 to 32767)
INTEGER	4-byte integer (-2,147,483,648 to 2,147,483,647)
BIGINT	8-byte integer (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
REAL*	inexact floating point value (generally $+-e^{37}$)
DOUBLE PRECISION*	inexact floating point value (generally $+-e^{307}$)
VARCHAR(n)	variable length character string up to n characters long
CHAR(n)	fixed length character string of n characters, blank padded if required
TEXT	variable length character string of virtually unlimited length

*Note that these datatypes are inexact, as some rounding is implied. Thus what went in may not be quite

the same as what comes out, and any checks using equality may have uncertain results.

The select command.

The SQL command *select* is used to retrieve data from the tables. This is a powerful and complex command used to define which records from which tables are to be retrieved. You can also use a wide range of SQL operators and functions on the data.

The simplest form of select is simply "select * from <table>;", eg:

*select * from catch;*

The asterisk ("*") is a wild card representing all columns. Instead of retrieving all columns, you can explicitly choose which columns:

select trip_code, station_no, species, weight from catch;

Postgres supports an extended display, which is useful when the output is wider than the screen, this is controlled using the toggle *\x,* which displays each column on a new line.

Also note that output displayed on screen is automatically displayed one screen at a time. Pressing Enter displays another row, pressing the space bar displays another page. Pressing *q* exists the output display and returns to the psql prompt.

When specifying columns to output in a select statement (or in a where clause, below) it is often useful to cast the values to a different type. A floating point number is not very readable on screen, but converted to a decimal() it is much easier. The SQL standard for casting values is:

select cast ('1' as integer);

This returns teh number one instead of the character '1'. Note that a decimal value of 1.0000 does NOT necessarily equal 1, but cast(1.0000 to integer) does. A convenient Postgres shorthand for this is the '::' operator. This is particularly useful when working with dates and times, for example:

select trip_code, station_no from station where date_s < '2000-01-30'::date;

This converts the character string '2000-01-30' to an internal date value, and returns only those rows where date_s is before (less than) this date.

The where clause.

Normally some criteria are specified to determine which records are retrieved. This is done using the where clause, for example, to just retrieve catches from one trip, tan0801, use:

select trip_code, station_no, species, weight from catch where trip_code = 'tan0801';

Note the word (or string) 'tan0802' must be enclosed in single quotes, which ensures they will not be

confused with column names. Numeric values do not have quotes.

A where clause is a series of expressions that evaluate as true or false. A complex series of expressions can be built up to specify the subset of available data to retrieve. Only records where all expressions evaluate to true will be retrieved, as in the following example:

```
select trip_code,
        station_no,
        species,
        weight
from catch
where trip_code = 'tan0801'
   and station_no=1;
```

Expressions can be added with either *and* or *or,* and parentheses are used to clarify precedence. For example, the where clause:

```
where trip_code= 'tan0801'
    and species = 'SSO'
    or species = 'BOE':
```

will return all SSO records for trip tan0801, and all records for all trips for species SSO. What is meant is:

```
where trip_code= 'tan0801'
    and (species = 'BOE'
    or species = 'SSO'):
```

Which returns SSO and BOE records just from trip tan0801.

The operators supported by Postgres within a where clause are:

<> or !=	not equals
<	less than
<=	less than or equal to
=	equal to
>=	greater than or equal to
BETWEEN	between two values (*station_no between 10 and 24*)*
EXISTS (NOT EXISTS)	tests whether records exist matching specified conditions: *select trip_code from trip t where exists (select * from fish_bio b where b.trip_code = t.trip_code);*
IN	tests whether a value is a member of a list: *select station_no from station where trip_code in ('tan0901','tan0902');* alternatively using a subquery: *select trip_code, staff from trip where trip_code in (select distinct trip_code from catch where species='ORH');*
ISNULL	tests whether a field has a value or not:

<> or !=	not equals
(NOTNULL)	*select lgth, weight from fish_bio where species = 'ORH' and lgth notnull and weight notnull;*
LIKE (ILIKE)	pattern matching (case sensitive) - use "%" as wildcard: *select trip_code from trip where trip_code like '%tan%';* (this will return values with 'tan' in them, but not 'TAN') ILIKE is a Postgres specific extension for case insensitive LIKE *select trip_code from trip where trip_code Ilike '%tan%';*
SIMILAR TO	SQL regular expression** *select waterfall_name from waterfalls where name similar to '(Fumee|Fumie|Fumy) Falls';*
regexp	not described here, full POSIX compliant regexp support.

* Be careful using between with character columns, as alphabetically 123 is between 12 and 14.

** Similar to supports the following regexp operators:

_	matches any single character
%	matches any string of characters
(...)	defines a subexpression
|	separates alternatives
*	matches zero or more
+	matches one or more
[...]	matches any of a set of characters
[^...]	matches any character NOT in a set

Select functions & operators.

Select provides a number of operators for manipulating values, so much more can be done with data than simply retrieving it. The "||" is the concatenation operator, converting the values on each side of it to character strings and concatenating them.

Select supports aggregate functions to return calculated values, based on aggregating rows. These include count(*), sum(*), avg(*), min(*), max(*) and variance(*).

Statistical aggregate functions:

AVG(*)	returns the mean value of a set of numbers
COUNT(*)	returns the number of rows in the set
MAX(*)	returns the largest value in the set
MIN(*)	returns the smallest value in the set
SUM(*)	returns the sum of the values in the set
VARIANCE(*)	returns the statistical variance of the numbers in the set

Thus to get the minimum and maximum depths where ORH was caught from all surveys (using

concatenate and min(), max() functions:

```
select min(min_gdepth), max(max_gdepth) from station
where trip_code||station in (select distinct trip_code||station_no from catch
                             where species='ORH');
```

Numeric or mathematical functions:

ABS(nn)	returns the absolute value of nn
CEIL(nn)	returns the smallest integer greater than or equal to nn (be careful with negative values)
EXP(nn)	returns *e* (2.71828183...) to the power of nn
FLOOR(nn)	returns the largest integer smaller than or equal to nn (be careful with negative values)
LN(nn)	returns the natural log of nn
LOG(nn)	returns log base10 of nn
LOG(base, nn)	returns the log of nn in the specified base
MOD(nn,dd)	returns the remainder of nn/dd
ROUND(nn[,pp])	returns nn rounded to pp decimal places (default is 0, returning an integer)
SIGN(nn)	returns '-', 0 or '+' depending on whether nn is negative, 0 or positive
TRUNC(nn[,pp])	truncates nn to pp decimal places, default being 0

Note that is is common practice to cast values to specific datatypes/formats as well as using functions to modify values. This is discussed later in the section on formatting output.

Date/time functions.

Postgres supports date and time datatypes, with full timezone support, and functions for working with these.

CURRENT_DATE	returns the current date (2008-12-06)
CURRENT_TIME[(pp)]	returns the current time with time zone (16:39:45.543042+12) with optional precision, pp
CURRENT_TIMESTAMP (NOW)	returns the current date & time (2008-08-06 16:41:43.189645+12) with time zone
LOCALTIME[(pp)]	returns local time, without time zone, (16:43:35.020812) with optional precision, pp
LOCALTIMESTAMP[(pp)]	returns current date and time (2008-08-06 16:43:35.020812) with optional precision, pp
TIMEZONE('TZ',TS)	returns date and time in timezone TZ where local timestamp is TS: *select timezone('PST',now());* (2008-08-05 20:50:40.239986)
DATE_PART('unit','date or time')*	returns specified part of the date or time: *select date_part('minute',now()) (55)*
EXTRACT(unit from date/time)**	returns specified part of the date or time: select extract(day from now()); (6)

* DATE_PART() is a Postgres specific function, so any SQL using this will be non-portable. Extract is the SQL standard way to accomplish this.

** EXTRACT() is the ANSI SQL standard for this. SQL's written using this function will be portable across most RDBMS packages. Standard units supported by the Postgres version of EXTRACT() are second, minute, day, month, year. Postgres also supports other non-standard units; century, decade, dow (day of week), doy (day of year), epoch (no. seconds since 1 Jan 1970), microseconds, millenium, milliseconds, quarter, timezone (offset from UTC in seconds), hour, timezone_minute and week. All of which makes Postgres a relatively useful package for working with temporal data.

Spatial (geometry functions).

The internal Postgres spatial datatypes and functions are deprecated, and have been superceded by a third party spatial engine, PostGIS. PostGIS is not discussed in this document.

Formatting output.

Output formatting in Postgres is essentially modal, meaning that you issue an instruction to toggle various aspects of output formatting off or on. For formatting individual columns in the output, see the description of cast() and the '::' operator above.

The commands are all \ commands, and on line help with these is available by typing \? on teh psql command line.

\t	toggles the output of header & trailer output (column names & output record counts)
\a	toggles aligned vs unaligned output (similar to the Empress *dump* instruction)
\f ','	specifies the character to use as a field separator for CSV style output (with \a)
\x	expanded output, columns each displayed as a row (similar to the Empress *list* instruction)

Column names can be set, for derived and real output columns using *as*.

The lgth table stores the number of all fish, male fish and female fish measured for each trip/station/species/length. Thus to extract the number of male, females & unsexed oreo's from a survey, you could use:

```
select trip_code,
        station_no,
        species,
        lgth ,
        no_a,
        no_f,
        no_a-(no_m+no_f) as no_u
from lgth
where trip_code = 'tan9908';
```

Sorting output.

When records are inserted into Postgres tables (as in most RDBMS's) they are allocated the first available free space. As data is inserted & removed, where this space occurs is not sequential, so the order data is retrieved in is indeterminate.

Any ordering of output is done explicitly in the SQL statement, using the *order by* command. Some databases provide an implicit sort when a distinct or group by command is specified, Postgres does not.

The following SQL, modified slightly from the previous example, sorts by trip, station, species and length:

```
select trip_code,
        station_no,
        species,
        lgth ,
        no_a,
        no_f,
        no_a-(no_m+no_f) as no_u
from lgth
where trip_code = 'tan9908'
order by trip_code,
            station_no,
            species,
```

lgth;

Grouping output: group by

Rows returned by a select statement can be grouped by common attributes. This is particularly useful when using the aggregate functions described previously. Grouping is done using a group by statement to specify the attributes the grouping will be done by. The grouping can be nested by giving a list of attributes to group by, rather than just one.

To return the total catch weight for all species for each station in a specified trip:

```
select trip_code,
        station_no,
        sum(weight) as weight
from catch
where trip_code='tan9908'
group by trip_code,
             station_no;
```

or the number of stations undertaken by each trip:

```
select trip_code,
        count(*) as count
from station
group by trip_code;
```

In a group by SQL, every column which is not an aggregate MUST be specified in the group by clause.

The SQL having clause is similar to the where clause, but is only applicable to grouped queries. Like a where clause, the having clause is a sequence of statements each of which evaluates to true or false, and only rows where all constraints are true will be returned. To restrict the rows returned by the above query to only trips which completed more than 100 stations, and order these by trip code:

```
select trip_code,
        count(*) as count
from station
group by trip_code
having count(*) > 100
order by trip_code;
```

Table joins.

All queries so far have been from a single table. Few queries are normally this straightforward, and more than one table is queried. Such queries utilise a relational join to link the tables. A malformed join can still return rows, but maybe not the rows that were intended, so carefully check the results of such

queries to ensure the results are sensible.

To apply a join, the tables being joined should be logically linked by one or more common attributes. The station table is linked to the trip table by the common attribute, trip_code. In a properly crafted join between these tables, the attributes of the appropriate trip will be linked to very station from that trip, but no others.

To retrieve records of fish catches along with the positions, for example, the station table (position data) must be joined to the catch table (catch weights). Two columns are required to specify the correct join, trip_code and station_no.

```
select s.trip_code,
        s.station_no,
        lat_s,
        long_s,
        species,
        sum(weight) as weight
from station s,
       catch c
where s.trip_code = 'tan9908'
   and c.trip_code = s.trip_code
   and c.station_no = s.station_no
group by s.trip_code,
            s.station_no,
            s.lat_s,
            s.long_s,
            c.species
order by s.trip_code,
            s.station_no,
            c.species;
```

The constraints forming the join are in the where clause, where every station record with a trip of tan9908 is joined to every catch record where the trip_code and station_no are the same in both tables. When the target tables were identified in the from clause, an alias was specified for each table, and this was applied to every column where there may be an ambiguity about which table the column will come from. The aggregate function sum() was applied to the weights, and the group by defined how the would be aggregated to generate the sum. Finally, the order the output is to be returned is specified in the order by clause.

Optimising joins.

Historically the sequence specified in the where clause could have a significant impact on the performance of the database engine carrying out the query. Modern databases such as Postgres utilise a query optimiser, which examines statistics about tables, attributes and indices automatically stored in the database, and uses these to optimise the way the query is run. The query optimiser is not infallible, but generally the way a query is written will have little impact on the performance. Database design &

proper indexing is much more important in this regard.

Subqueries.

A subquery is a query within a query. They are generally used where data in one table is used to determine which records from another table should be returned. An example is given earlier in the table describing select query operators. To get the trip_code and staff list for trips which had stations where ORH were caught:

```
select trip_code,
        staff
from trip
where trip_code in (select distinct trip_code from catch where species='ORH');
```

Postgres will first evaluate the subquery, which returns a list of trip codes where ORH were recorded from the catch table, then use this list to select the rows from the trip table to return.

Handling output.

So far all output has been to the screen. More often, output is sent to files to be used with other packages for further analysis. The \o toggle command is used to redirect output to a file instead of to screen. This command needs a filename to tell psql where to send the output. eg:

```
\o output.txt
```

This will write the output to the file output.txt. As described in the getting started section, Postgres provides a command history which can be navigated using the up/down arrow keys (and some other commands described in the \? online help). This means an SQL can be entered & the output viewd on screen to ensure it is correct, then type \o <*file*>, then recall the required SQL to run using the arrow keys to re-run it, with the output now going to the file.

Note that the facility for a one page-at-a-time output on screen does not apply to file output.

Controlling the output format is discussed above.

Selecting data into other tables.

At times, it can be useful to take the output from a query and insert the rows into a new table. Note that this can only be done if you have been given the appropriate database privileges.

To do this you need to tell Postgres where the output will be going, then run the SQL to generate the output. If the target table exists, the data will be appended to that table, otherwise the table will

generally be created to store the output. To create a new table with just one trip's aggregated catch data, with catch weights from all methods combined:

```
insert into tan9908_catch
select trip_code,
        species,
        sum(weight) as weight
from catch
where trip_code='tan9908'
group by trip_code,
            species;
```

Batched queries.

It can be more convenient to use an editor to write the SQL commands to a file, then run them from that file, instead of typing them in in at the psql prompt. If you do this, and have an SQL (or more than one) in a file, you can invoke them with \i <*filename*>. This tells psql to read this file and run the SQL statements there. Any line starting with two hyphens, "--" is ignored, so such files can contain comments prefixed with -- to make them more human-readable.

Appendix 5. Cheat sheet for Empress users.

Quick reference of PostgreSQL commands for Empress users.

Action	PostgreSQL syntax	Empress syntax
invoke database	psql -d database *eg* psql -d cod	empsql database *eg* empsql trawl *or* ms trawl
quit or exit database	\q	exit *or* stop *or* quit;
display or describe database ie list tables (add + to most \d options to output more detail	\d[+]	display db;
display table	\d <tablename>	display <tablename> ;
help on sql commands	\h	help;
database help	\? shows psql commands preceded by '\' as above etc	
vertically list output	\x	… **list** [into pager] ;
display previous queries	\s (or up/down arrows)	rc all ; (or up/down arrows)
redirecting output to a file	\o	select * from <tablename> **into** <outfile> ;
data only output	\t (tuples only – suppresses headers and trailers)	dump
formatting output	\a (toggles alignment – eliminates white space between columns)	dump
SQL commands	NB the psql syntax below works in Empress also	Some Empress commands that do **NOT** work in psql
select or retrieve data	select * from <tablename> ; NB defaults to 'pager' i.e., displays one screen full initially, (* is required if no columns are specified)	select from <tablename> (* is not required)
sort ie output from select	order by	sort by (or order by)
pattern matching	like % NB wildcard character is %, like is a case sensitive pattern match. Use ilike for case Insensitive search (but ilike will not work in Empress).	match * NB wildcard character is * , match is not case sensitive. eg select * from trawl:t_trip where trip_code like 'TAN08%'; will not return a result, but … match 'TAN08%' will.
re-labeling column names in output	eg. select no_a **as** no_all from t_lgth;	select no_a **print** no_all from t_lgth;
text strings in SQL	delimited by single quote	single or double quote
select data in a range between values	select * … where <attr> **between** <value1> and <value2> ;	select * … where <attr> range <value1> to <value2>: