

# Programming for Chemical and Life Science Informatics

I573 - Week 8  
(Databases for Chemistry)

Rajarshi Guha

3<sup>th</sup> & 5<sup>th</sup> March, 2009

# Outline

- Overview of databases
- Introduction to SQL
- Programmatic access to SQL databases
- Handling chemistry in databases
- Incorporating cheminformatics in relational databases

# Handling Lots of Data

- Large amounts of data, characterized by features (fields)
- Some data is related to other data, maybe in multiple ways
- Need to quickly retrieve specific items, based on conditions
- How do we do this
  - ▶ Quickly?
  - ▶ Reliably?
  - ▶ In a scalable manner?

- **relation** - a collection of tuples with the same set of attributes (fields). Also called a **table** in SQL
- **tuple** - a single row. A description of a single item. Also called a **row** in SQL
- **attribute** - a feature of a set of items. Also called **column** in SQL
- **key** - a constraint on the tuples in a relation. A common example is that a relation can only have unique tuples

# What Type of Data?

- Databases can be used for literally any type of data you want
- Traditional databases are more suitable for certain data types than others
- We'll restrict ourselves to chemical data
  - ▶ Names (common names, IUPAC nomenclature)
  - ▶ Structures (SMILES, InChI, 3D)
  - ▶ Properties (numerical, textual)
- We're also going to consider large datasets - 100's of thousands of objects
- However a database can be profitably used even for a few hundred objects

# Alternatives to Databases

- Notepad and text files work for a few items (say 10) with a few attributes (say 3 or 4)
- For more data you could use Excel
  - ▶ The use of a spreadsheet in lieu of a database is a gross mistake
- Write your own code - but the DBMS people have already written the stuff that you'll write

# What's Available?

- Well known OSS databases include
  - ▶ Postgres
  - ▶ MySQL
  - ▶ BerkeleyDB
- Which one you choose depends on the task and requirements
- As we'll see cheminformatics can be integrated with Postgres or MySQL
- But they're pretty heavyweight and require tuning
- If you just need a small database for your program, consider [SQLite](#), [Derby](#)
  - ▶ Don't need any separate server process - embedded within your program
  - ▶ Very small (< 2MB) footprint
  - ▶ Can run in memory, so can be very fast

- Most of this lecture is database independent
- But since we will address cheminformatics in the DB, some parts will be Postgres specific
- Postgres is installed on *cheminfo* - so you can play with it there
- Though you can install on your own system, we'll be using a commercial cartridge later on
- Basic concepts will transfer to any other RDBMS

# Accessing Postgres

- Log into *cheminfo*

```
psql -U i573 -d gnova
```

- There is no password
- Can't access it from a remote machine
- You'll get an SQL prompt
- There are **GUI's** to interact with Postgres
  - ▶ **pgAdmin**
  - ▶ **xpg**

# Databases & Tables in Postgres

- A database contains one or more tables
- When logging into Postgres (or connecting to it from a program) you must specify a database name
- In general, you perform queries on one or more tables in a *single* database
- If you need to access multiple databases, things get tricky
- See [here](#) for solutions

Part I

Introduction to SQL

# SQL Data Types

- Postgres supports a wide variety of data types
- Most are described by standard SQL - so are portable to other DB's
- See [here](#) for a detailed description of all available types

## Numeric

- `smallint` (-32768 to +32768)
- `integer` (-2147483648 to +2147483647)
- `bigint`
- `decimal` (no limits)
- `serial` (autoincrementing integer)

# SQL Data Types

## Character

- `varchar(n)` - variable length string with a limit
- `char(n)` - fixed length, space padded
- `text` - unlimited length text

## Bit string

- `bit(n)` - bit string, of exactly `n` bits
- `bit varying(n)` - bit string of upto `n` bits
- Perfect for fingerprint data

# SQL Data Types

- There are many more data types
  - ▶ Date/time
  - ▶ Enumerations
  - ▶ Geometric
  - ▶ Arrays
  - ▶ XML (not available on *cheminfo*)
- The choice of data type to use is important
  - ▶ Size - number of heavy atoms in a molecule probably won't cross 32768 (ignoring proteins), so use a `smallint` rather than `integer`
  - ▶ Indexing - certain data types may not be indexable, so you can't do fast queries

# Database Design

- Database design can be a very complex topic
  - ▶ Choosing data types is just one small part of this
  - ▶ An interesting [blog post](#)
  - ▶ There are a number of texts available on this topic
- Depending on how tables are set up and what type of relationships are defined between them, queries can be very fast or very slow
- Database design is also important to maintain integrity of the data
- We won't go into this aspect very deeply

# Table Creation

- Detailed [description](#)
- At the very least we specify
  - ▶ column names
  - ▶ column types
- We can add various constraints separately

```
create table chemicals (  
  cid varchar(30),  
  smiles text,  
  inchikey text,  
  nheavy integer);
```

- You can paste this at the Postgres prompt

# Get Descriptions

- If you build many tables, you want to see the current tables in the database

```
gnova=> \dt
```

```
      List of relations
```

```
 Schema |                Name                | Type | Owner
```

```
-----+-----+-----+-----
```

gnova	amw	table	postgres
gnova	formats	table	postgres
gnova	public166keys	table	postgres
gnova	smiles	table	postgres
gnova	system	table	postgres
gnova	tests	table	postgres
gnova	tpsa	table	postgres

```
...
```

# Get Descriptions

- Given a table name, you can get a description of the fields and associated keys, triggers etc

```
gnova=> \d pubchem_3d
```

```
Table "public.pubchem_3d"
```

Column	Type	Modifiers
cid	character varying(256)	not null
structure	text	not null
method	text	not null
momsim	cube	

```
Indexes:
```

```
"pubchem_3d_pkey" PRIMARY KEY, btree (cid)
```

```
Foreign-key constraints:
```

```
"pubchem_3d_fkey" FOREIGN KEY (cid) REFERENCES  
pubchem_compound(cid) ON DELETE CASCADE
```

## Table Creation - Primary Key

- Good database design suggests that one of the columns be unique and every row of that column should have a value
- This can be implemented by creating a unique index and is called the *primary key* (PK)
- Sometimes the PK is on a column that is part of your data
  - ▶ An ID number generated by someone (SSN, PubChem ID etc)
  - ▶ If your data doesn't have some unique identifier, you can create a column with autogenerated values

```
create table chemicals (  
  cid varchar(30) primary key,  
  smiles text,  
  inchikey text,  
  nheavy integer);
```

## Table Creation - Primary Key

- Sometimes your data may not have a column that will have unique values
- But a combination of two columns will have unique values
- A table can only have one primary key

## Table Creation - Default Values

- After creating table, you will load data into it
- Sometimes, you know that if not specified, a column can have a default value
- You can indicate this in the create table statement

```
create table chemicals (  
  cid varchar(30) primary key,  
  smiles text default 'C',  
  inchikey text,  
  nheavy integer default 0);
```

- For a given row, if you don't give a value for the smiles and nheavy fields, they will get the default values noted above

# Constraints

- One of the nice things about a DB is that you can off-load a number of integrity checks
  - ▶ Less code for you to write/debug
  - ▶ DB will tell you when there is a problem
- What type of things can be checked for in the DB?
  - ▶ Check for a certain value or range
  - ▶ Ensure that a value is not NULL
  - ▶ Ensure no duplicates in a column
  - ▶ By using triggers and stored procedures, you can implement any sort of constraint you want

# Constraints

- Though you can always add constraints after table creation, we'll include them in a create table statement

```
create table chemicals (  
  cid varchar(30) primary key,  
  smiles text default 'C',  
  inchikey text,  
  nheavy integer default '0' check (nheavy > 0));
```

- A primary key is simply a combination of a unique constraint and a not null constraint
- The above constraints are *column* constraints since they refer to specific values of a column

# Inserting Data

- After creating a table, we need to insert data
- If you're inserting a small amount of data you can use the `insert` command

```
insert into chemicals (cid, smiles) values  
    ('187563', 'CCCC');
```

- Though this is valid, it won't work for our table since we have said that `nheavy > 0` but the default is 0

```
insert into chemicals (cid, smiles, nheavy) values  
    ('187563', 'CCCC', 4);
```

- If you don't specify the columns, you have to provide a value for every column in your table

# Loading Data

- insert becomes very tedious very quickly. It's also slow
- If you have a lot of data, it is fast to do a bulk load using copy
- Create a plain tab delimited text file, containing the values for each row - should specify a value for each column

11435978	CC1=C(C=C(C=C1)C(C#N)C2=CC=CC=C2)C	VIDPFDPVOLNIM-UHFFFAOYAP	17
11436066	CC(=O)OC(C1=CC=CC=C1)C(C=C) (F)F	SVALUHPMXILJNR-NSHDSACABW	16
11436067	C1=CC=C(C=C1)C(=O)NC(=O)C2=CN=CC=C2	JVYJILSJVZYFKY-UHFFFAOYAV	17
11436077	CC1=C(C(=O)CC1)C=CCCC2=CC=CC=C2	OGHDBGDSJGNAIH-POHAHGREBX	17
11436091	C1=CC=C(C=C1) [Se]CCCC=O	OXCCQRNIKXXVRU-UHFFFAOYAO	12
11436204	CC(=CCCOC(=O)CCC1=CC=CC=C1)C	VXVUUNKERUJOGI-UHFFFAOYAL	17
11436215	C[Si](C)(C)C#CCCOC1=CC=CC=C1	QVXWSAHJMRAWIV-UHFFFAOYAN	16
...			

- You can use CSV if you want
- Each record is on a new line
- If a certain field can contain new lines you need to escape them

# Loading Data

- Add the following line to the top of the file

```
copy chemicals(cid,smiles,inchikey,nheavy) from stdout;
```

- Add the the following line to the end of the file

```
\.
```

- Load the data into the database by doing

```
psql -U i573 -d gnova < data.txt
```

- You can write programs to insert data into the database, but `insert` is much slower than `copy`
- Better to write a program that will generate a file that can be loaded using `copy`

# Querying

- Queries can be of varying complexity

```
SELECT col1, col2, ... FROM table WHERE condition
```

- The simplest is to simply retrieve all columns for all rows

```
select * from chemicals;
```

- If you do this on *cheminfo* you'll get back 1,000,000 rows and your screen will scroll forever
- We can specify that specific columns should be retrieved

```
select cid, inchikey from chemicals;
```

# Querying

- Getting the whole table is not very interesting
- We usually are interested in rows satisfying a condition

```
select cid from chemicals where nheavy < 10;  
  
select cid from chemicals where  
    nheavy < 10 and  
    cid ~ '^114';
```

- ~ means that the string field matches a regex
- Sometimes you're just interested in how many rows get returned for a certain condition

```
select count(cid) from chemicals where  
    nheavy < 10 and  
    cid ~ '^114';
```

# SQL Functions

- The SQL standard defines a number of functions
- Examples include count, avg, min etc

```
gnova=> select max(nheavy), avg(nheavy) from chemicals;
max |          avg
-----+-----
576 | 28.1789330000000000
(1 row)
```

- The functions work on the result of the query

```
gnova=> select max(nheavy), avg(nheavy) from chemicals
           where nheavy < 20;

max |          avg
-----+-----
19  | 15.3613683127572016
(1 row)
```

# Making Some Assay Data

- Let's add some assay data for our compounds

```
create table assay (  
  id bigserial primary key,  
  cid varchar(30),  
  aid varchar(30),  
  activity integer  
  check (activity >=0 and activity <= 100));
```

- We fill it with simulated data according to the following conditions
  - ▶ 50% of the compounds have been assayed
  - ▶ A compound is tested in upto 3 assays
  - ▶ Assay values range from 0 (inactive) to 100 (most active)

# Analyzing the Assay Data

- Since we generated the data, we'd like to see what we got
- Remember that a given compound may occur more than once in the assay table
- How many assay entries do we have?

```
select count(*) from assay;
```

```
count
```

```
-----
```

```
1001633
```

```
(1 row)
```

# Analyzing the Assay Data

- How many unique compounds were assayed? How many unique assays are there?

```
select count(distinct cid) from assay;
```

```
count
```

```
-----
```

```
500901
```

```
(1 row)
```

```
select count(distinct aid) from assay;
```

```
count
```

```
-----
```

```
5
```

```
(1 row)
```

# Analyzing the Assay Data

- How many unique compounds were inactive in any assay that they were tested in?

```
select count(distinct cid) from assay where activity = 0;
```

```
count
```

```
-----
```

```
9830
```

```
(1 row)
```

## Analyzing the Assay Data

- Find the CID's that have been tested in exactly 3 assays and have an activity value greater than 75 in all of them. Report the CID's and the average value of the activity in the 3 assays

```
select cid, avg(activity) from assay
       group by cid
       having count(cid) = 3 and min(activity) >= 75;
```

cid	avg
1000322	81.6666666666666667
1001476	89.0000000000000000
1002116	89.0000000000000000
1002127	90.6666666666666667
...	

# Querying Multiple Tables

- You can compare rows from multiple tables

```
select chemicals.cid from chemicals, assay where
    chemicals.cid = assay.cid and
    chemicals.nheavy < 20 and
    assay.activity > 80
```

- Identifies CID's that have been assayed, have less than 20 heavy atoms and have an activity greater than 80

## Querying - Ordering and Limiting

- Sometimes you just want the first 10 results

```
select cid from chemicals where nheavy < 30 limit 10;
```

- Maybe you want the results ordered by the number of heavy atoms

```
select cid from chemicals where nheavy < 30  
      order by nheavy limit 10;
```

## Part II

# Optimizing Performance

- To determine how long a query takes enter `\timing` at the Postgres prompt
- All subsequent queries will report the elapsed time
- If you run the `group by` query on the `assay` table it takes 17.2s
  - ▶ This is too long for a database response
- On the other hand identifying CID's in the chemicals database which has less than 10 heavy atoms takes 0.47s
  - ▶ Not bad. But we can do better

# Improving SQL Performance

- Database tuning
  - ▶ Shared memory settings, I/O optimization, logging
  - ▶ Requires knowledge of hardware, type of load etc
- Rewriting SQL queries
- Generating indexes
  - ▶ Easiest way out and gives good results
  - ▶ Not necessarily the best solution
- Optimizing database performance is like any other optimization procedure
  - ▶ Use it only after the results are known to be correct
  - ▶ Use the appropriate tools - don't guess

- An index is a quick way to identify rows
- Hash tables are a very simple type of index
- Lots of CS theory behind good indexing
- Postgres supports 3 types of indexing schemes: B+ Tree, R Tree and GiST
  - ▶ B+ tree is general purpose (very good for strings or 1D range queries)
  - ▶ R tree is usually for spatial data

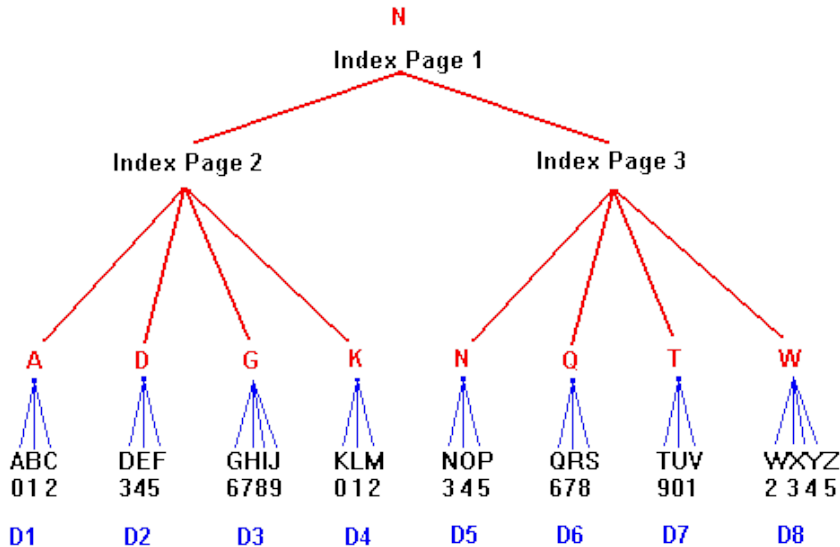
# What Does an Index Solve?

- In the absence of an index, the DB must look at every row and see if it satisfies the query conditions
- This is called a *sequential scan*

```
select cid from chemicals where nheavy < 10;
```

- For a million rows, it's not too bad. But 10M rows?
- It'd better if we could some how group the rows, so that those with `nheavy < 10` were in one group, `10 < nheavy < 20` in another and so on.
- Given a condition we can first identify the group and then look just within the group

# B+ Tree



# Creating an Index

- Choose which column you want the index on and select an index type (default is B+ tree)

```
create index nheavy_idx on chemicals (nheavy);
```

- For large tables it'll take some time to run
- Which columns should be indexed?
  - ▶ The ones that are most frequently used in a query
- If queries usually involve multiple columns, use a combined index

```
create index combo_idx on chemicals (nheavy, inchikey);
```

## Queries Using an Index

- You usually don't have to specify that an index is to be used
- If an index is available, Postgres uses it
- How many compounds with less than 10 heavy atoms (*no index*)

```
gnova=> select count(cid) from chemicals where nheavy < 10;
count
-----
 11654
(1 row)

Time: 478.094 ms
```

## Queries Using an Index

- After the index was created?

```
gnova=> select count(cid) from chemicals where nheavy < 10;
count
-----
11654
(1 row)

Time: 68.094 ms
```

## How Useful Was It?

- From our point of view 0.5s versus 0.06s is not much - both seem instantaneous
- The advantages become more dramatic for larger tables
- Also, if the query is part of a larger query, this can lead to significant improvements
- So, should you just put indexes on all columns?
  - ▶ No
  - ▶ Indexes themselves take up size
  - ▶ Lookups take computation
  - ▶ Should examine common queries, and create the appropriate indexes

# Indexes vs Better SQL

- Consider the following query on the assay table

```
select cid, avg(activity) from assay
  group by cid
 having count(cid) = 3 and min(activity) >= 75;
```

- With no index it takes 17.2s
- Since we're querying on activity and cid separately, lets put indexes on these columns

```
create index cididx on assay(cid);
create index actidx on assay(activity);
```

## Indexes vs Better SQL

- The same query now takes 16.4s
- Why such a small improvement?
- To understand how Postgres is performing query, add `explain analyze` to the beginning of the query

```
explain analyze select cid, avg(activity) from assay
  group by cid
  having count(cid) = 3 and min(activity) >= 75;
```

# Analyzing Query Performance

- Good discussion of understanding the output of `explain analyze` is [here](#)

```
QUERY PLAN
-----
GroupAggregate (cost=172012.70..186236.50 rows=75706 width=15) (a
  Filter: ((count(cid) = 3) AND (min(activity) >= 75))
    -> Sort (cost=172012.70..174516.79 rows=1001633 width=15) (act
      Sort Key: cid
        -> Seq Scan on assay (cost=0.00..34520.33 rows=1001633 w
```

- For now ignore the numbers, and read up from the bottom
- The problem is we're doing a sequential scan on the whole table and not really making use of the indexes we created
- Furthermore, since `group by` has to sort the CID's, we're really sorting the whole table

- This is an example of where guessing is not a good strategy for optimization
- We can speed things up, by making sure that the `group by` only has to work with the rows which have an `activity` greater than 75

```
select cid, activity from assay where activity > 75;
```

- Takes 0.5s and gives us 257,596 rows
- We can now combine this result with the original `group by` so that we look at this subset (rather than 1M rows)

- We make use of a sub query

```
select cid, avg(activity) from (  
    select cid, activity from assay where activity >= 75  
) as xcid group by cid having count(cid) = 3;
```

- Takes 3.8s - 4.5x improvement

# How Is It Working?

- We take a look at how Postgres performed the query by prepending explain analyze

```
-----  
GroupAggregate (cost=59706.11..62601.69 rows=19329 width=15) (actu  
Filter: (count(cid) = 3)  
-> Sort (cost=59706.11..60345.44 rows=255732 width=15) (actual t  
Sort Key: assay.cid  
-> Bitmap Heap Scan on assay (cost=4226.26..31926.91 rows=  
Recheck Cond: (activity >= 75)  
-> Bitmap Index Scan on actidx (cost=0.00..4162.33 r  
Index Cond: (activity >= 75)
```

- Main bottleneck is still the sorting step - but no way around that if we want to use group by

## Part III

# Programmatic Access

# RDBMS Support

- Given that SQL is a standard, most databases should be accessible in an identical manner from a program
- In fact the SQL is not the issue
- It's how one sends and receives data from a database that differs
- Thus, most languages provide a unified API to RDBMS's
  - ▶ JDBC for Java
  - ▶ DB-API for Python
  - ▶ DBI for Perl
- If you use these API's it doesn't matter what database is used as the backend - your code should work identically

**Programmatic access to an RDBMS is basically about sending SQL to a database and getting back results**

# Python & PostgreSQL

- Python supports many databases - PostgreSQL, MySQL, Oracle etc
- Each database is supported by it's own package
- For PostgreSQL we use [psycopg2](#)
- To switch to another database you need only change
  - ▶ The database specific module that you import
  - ▶ The connect statement

# Connecting

- Connecting is not very Pythonic
- Create a string with the keyword arguments

```
import psycopg2

con = psycopg2.connect('dbname=gnova user=i573')
cursor = con.cursor()
```

- The cursor object is what we use to interact with the database

## Sending a Query

- Construct the SQL as a string
- For web-based programs this can be a very dangerous step
  - ▶ If some user input goes into the SQL statement, must make sure that the user input is not malicious
  - ▶ So don't blindly add stuff from an HTML text box into your SQL!

```
sql = '''select * from chemicals where nheavy < 10'''  
cursor.execute(sql)
```

- If there is an error in your SQL or some problem in sending the query an exception is thrown

# Getting Back Results

- There are two methods of the cursor object that we can use
  - ▶ `fetchone()` returns a single row of the result set. Good for very large result sets
  - ▶ `fetchall()` returns all the rows of the result set as a list
- In both cases a single result is a list object

```
while True:
    row = cursor.fetchone()
    if not row: break
    print row
```

- This is memory efficient, whatever the size of the result set
- `cursor.fetchall()` gives you a list of lists

# Transactions

- To ensure data integrity an RDBMS views each SQL statement as a transaction
- It's possible to indicate that multiple statements together constitute a single operation (transaction)
- If some system failure occurs during a transaction, the RDBMS can *rollback* to it's original state before the transaction
- This is very important for INSERT, UPDATE and DELETE statements

# Commit

- Each call to `execute()` represents the start of a single transaction
- However, the database does not know when the transaction ends
- So if your program exits, the DB considers that the transaction *failed*
  - ▶ Any changes made by your program will be lost
- To indicate that a (set of) transaction is complete call the `commit()` method on the connection object before you exit
- You can call it after each `execute()` - but this will make the database slow
- If you're manipulating lots of data, call `commit()` at certain intervals

## Finishing Up

- Before exiting close the connection using `close()`
- Most times, Python will clean up, but it's good practice
- DB access from Python is pretty easy
- It's very similar in Java (Perl, Ruby etc.)
- In these types of programs, it's the SQL statements and database tables that should be the focus of performance