Relational Database Basics

Ramses van Zon

May 29, 2023

About this course

About this course

SciNet

Aim

The aim of this course to make you familiar with different ways to store and manipulate databases in research computing so you know how to choose the right tool for your data.

Prerequisites

Some Linux command-line experience. Python programming knowledge is strongly advised.

Introduction

What are Databases?

SciNet

Database:

An organized collection of data for storing, managing and retrieving information

There are many different types to organize data.

Which type to use depends on the type of data and the way in which it will be used.

Often, the term "database" is used for a particular type called a "relational database".

We will look at a number of common database types and their uses, particularly for research purposes.

Overview of database types

Database types

SciNet

1. Relational databases (RDB)

   Bunch of tables, client-server model, with a standard querying language ("SQL").

   We will see that not all data fits nicely in this, so there are so-called "NoSQL" alternatives, e.g.

2. Key-value
3. Wide column databases
4. Document databases
5. Graph databases
6. Array databases

   These often still have a server-client model and a particular method to query the data. For performance and generality, alternatively, one can also use

7. Portable. structured binary storage.
   E.g. NetCDF, HDF5

Relational databases

- Often back-end to a website or application.
- Typically structured as tables.
- Each row is a *relation* between columns
- Relations between tables through keys.
- Standard language to use database: Structured Query Language (SQL)

  Usually used from another host language.

Why use RDBs in research computing?

SciNet

- Your application relies on it.
- It tends to be more structured than using a file system.
- It is somewhat self-documenting.
- Relative easy and efficiency of individual data entry, updates and deletions, retrieval and summarization, e.g.

  Get the data from simulations in which the pressure was less than 2 MPa and the number of molecules was less than 500.

```
SELECT * FROM measurement M JOIN parameter P ON M.runid=P.runid WHERE M.p<2 AND P.N<500
```

Even if RDBs don't fit all (or any of) your data, they are a good way to start thinking about how to organize and store data.

When would you not use a relational database?

SciNet

- Although databases allow storage of binary data, accessing, updating, etc. can be cumbersome and heavy on I/O operations.
- This is especially so for large binary datasets (e.g. data on a grid).
- There are better self-documenting formats for binary data (netcdf, hdf5, ...)
- When running many cases in parallel, you do not want all of them access a database simultaneously. At best, this creates a bottleneck.
- No parallel I/O.

Combining tools

SciNet

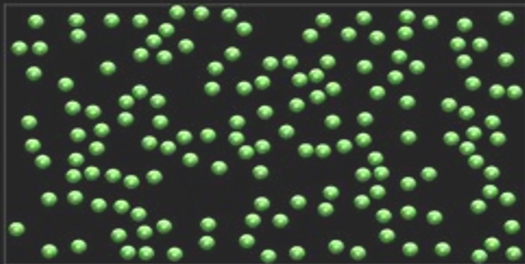
You can combine relational databases with more traditional approaches:

- Leave binary data on the file system.
- Store other things in a database, such as job details, simulation records, locations of binary files, and overall properties.

Use case: Parameter sweeps

Use Case

SciNet



- We are to perform a set of molecular dynamics simulations on the fluid *Argon*.
- We want to get the pressure of Argon as a function of temperature and density.
- Need to simulate a grid of (temperature,density) points: Parameter sweep

- Let's say the application to run the simulations is given to us, and can be fed different input parameters through configuration files or command-lines arguments.
- Need to keep track of parameters, jobs, results, ...

Use Case (continued)

SciNet

- The simulation computes the motion of atoms.
- Averaging along the trajectories of a fixed number of atoms in a volume ($\rightarrow$ density), one finds the pressure and temperature.
- Changing the size of the simulation box, one gets different densities.
- Starting from a near-regular configuration, takes a few picoseconds to reach equilibrium.
- Adjusting velocities of the atoms during equilibration, one gets different temperatures.
- Typical units:

picoseconds	time
Ångstrom	length ($=10^{-10}$m)
Kelvin	temperature
kJ/mol	energy
mol/l	density
MPa	pressure

Use Case on Niagara

SciNet

```
$ ssh USER@niagara.scinet.utoronto.ca
$ cp -r /scinet/course/dbb23 $SCRATCH
$ cd $SCRATCH/dbb23
$ source setup
$ make
```

- This created the simulation app `lj` (for the *Lennard-Jones* force field).
- To simulate Argon gas at 100 Kelvin with a density of 2.5 mol/l using 500 atoms, for 12 ps of simulated time taking 5 fs time steps, and using a random seed of 17, equilibrating for 2 fs:

```
$ ./lj outputdir Ar 100 2.5 500 12 0.005 17 2.0 T
$ ls -l outputdir/
-rw-r--r-- 1 rzon scinet   114620 Dec  6 09:58 output.dat
-rw-r--r-- 1 rzon scinet 33768834 Dec  6 09:58 output.xsf
-rw-r--r-- 1 rzon scinet     1055 Dec  6 09:58 report.json
```

- `report.json` contains averages total, potential, and kinetic energy, temperature and pressure.
- The `xsf` file contains the trajectory. To omit output, changing the last argumentxs from T to F.

SQLite

Some Relational Database Management Systems

SciNet

- Commercial:
  - Oracle Database
  - IBM DB2
  - Microsoft SQL Server, Microsoft Access
  - SAP Sybase
- Open Source:
  - postgreSQL
  - MySQL/MariaDB
  - SQLite

We're choosing SQLite+Python here for the examples, but virtually everything will apply to other RDBMSs.

If your group has a good case for a hosted postgres database to use in your Niagara jobs, contact us at support AT scinet DOT utoront DOT ca

What is SQLite?

- Open-source RDBMS.
- It is light and relatively simple.
- Serverless: Each database is a file (unless it's in memory).
- Command-line interface (`sqlite3`) for SQL queries.
- Comes standard with Python since version 2.5.
- There are also interfaces for C, php, perl, R, . . .
- Third-party gui interfaces (e.g. `sqlitebrowser`)

SQLite Datatypes

SciNet

One thing that varies among RDBMSs are the supported data types.

For SQLite, the list of storage classes is relatively small:

- NULL
- INTEGER
- REAL
- TEXT
- BLOB

Common SQL types (TINYINT, VARCHAR, DATE, ...) are accepted, but have affinity with a storage class.

SQLite+Python: A Simple Example

name	type
Missy	cat
Puppy	dog
Bingo	dog
Kitty	cat

```
import sqlite3
db=sqlite3.connect('path/filename.sqlite3')
dbc=db.cursor()
dbc.execute("CREATE TABLE pet(name TEXT,type TEXT)")
dbc.execute("INSERT INTO pet VALUES ('Missy','cat')")
dbc.execute("INSERT INTO pet VALUES ('Puppy','dog')")
dbc.execute("INSERT INTO pet VALUES ('Bingo','dog')")
dbc.execute("INSERT INTO pet VALUES ('Kitty','cat')")
db.commit()
dbc.execute("SELECT * FROM pet").fetchall()
db.close()
```

Python/SQL mix

SciNet

```
db.execute("CREATE TABLE pet(name TEXT, type TEXT)")
```

- Outer part (`dbc.execute(...)`) is Python, the *host language*.
- Inner part is *SQL*.
- SQL the same in all host languages.

SQL overview

SQL

```
CREATE TABLE pet(name TEXT, type TEXT)
```

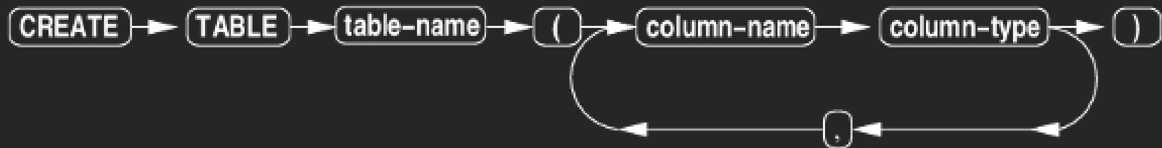
- This is a bit of SQL
- SQL=Structured Query Language
  - A ‘fourth generation’ programming language: verbose, expresses intention more than implementation
  - Case insensitive
  - Despite ANSI standard, every implementation differs in details
- Common commands

- CREATE
- DROP
- ALTER
- INSERT
- UPDATE
- SELECT

SciNet

Creates an empty table given a schema.

Partial syntax:



Example

```
CREATE TABLE pet(name TEXT, type TEXT)
CREATE TABLE IF NOT EXISTS pet(name TEXT, type TEXT)
```

Creates a table pet in current database with two columns, name and type, both of which will contain text.

DROP

Removes a table from the database.

Partial syntax:

Example

```
DROP TABLE pet
```

Removes the table ‘pet’ from the current database.

ALTER

Adds a column to an existing table.

(Can also rename a table)

Partial syntax:

Example

```
ALTER TABLE pet ADD COLUMN age INTEGER
```

Adds a column called ‘age’ that will contain integers to the table ‘pet’.

INSERT

Inserts a row into a table.

Partial syntax:

Substitute `INSERT` by `REPLACE` to overwrite (must have unique id).

Example

```
INSERT INTO pet VALUES ('Nala',dog',4)
```

Inserts a row into the 'pet' table with the values 'Nala', 'dog', and 4.

UPDATE

Updates an existing row in a table.

Partial syntax:

Example

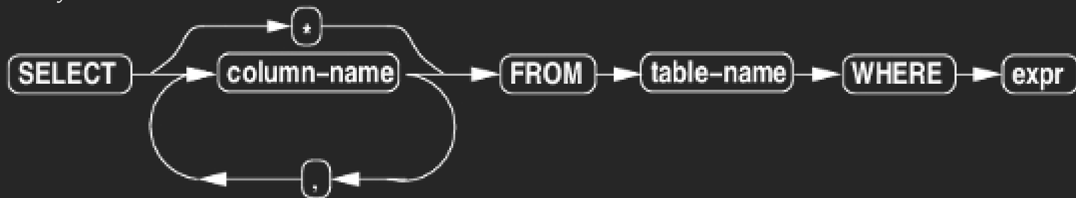
```
UPDATE pet SET type='cat' WHERE name='Nala'
```

Updates (corrects) the ‘type’ field in the row where the name is ‘Nala’.

SELECT

Looks up specific rows in a table.

Partial syntax:



Example

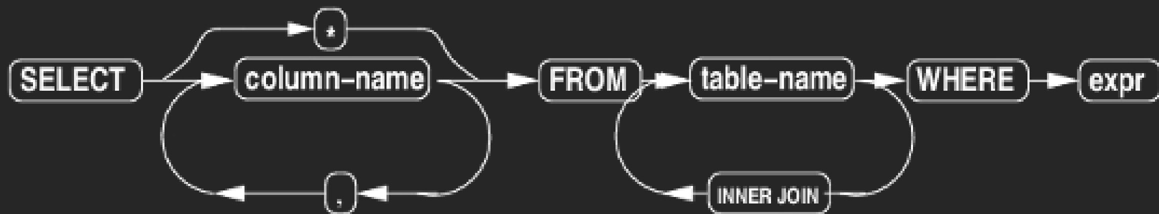
```
SELECT name,type,age FROM pet WHERE type='cat'
SELECT * FROM pet WHERE type='cat'
```

SELECT with JOIN

Looks up specific rows in combination of tables.

Instead of a table, you can specify joined tables.

Partial syntax:



Example

```
SELECT A.*, B.directory
    FROM simulation_attribute A INNER JOIN simulation_path B
    WHERE A.simulation_id=B.simulation_id
```

Full Syntax Reference of These Commands

SciNet

www.sqlite.org

Sqlite/Python interface

Sqlite/Python interface - Open the Database

SciNet

- Import the sqlite3 module in python:

```
import sqlite3
```

- You can then open a database with connect.

```
db = sqlite3.connect('somename.sqlite3')
```

- You should then get the database cursor

```
dbc = db.cursor()
```

Much of what you'd do with the cursor dbc, you can also do with the connection db, but this is non-standard.

Sqlite-Python interface: Query the Database

SciNet

- Any SQL command can be executed using `execute`:

```
cur = dbc.execute("CREATE TABLE MY_TABLE ...")
cur = dbc.execute("SELECT * FROM MY_TABLE")
```

  The result of an execute is called a 'cursor' again. . . .

- When you've executed a select, you can get the results out as follows: . . .
  - If expecting one row result, this will give it, as a single sequence: . . .

    ```
    cur.fetchone()
    ```

  - Get all the row results as a list of tuples: . . .

    ```
    cur.fetchall()
    ```

  - Get a partial list: . . .

    ```
    cur.fetchmany()
    ```

    Call repeated until you get an empty list.

Sqlite/Python interface - Getting results

SciNet

Example

```
import sqlite3
db=sqlite3.connect('path/filename.sqlite3')
dbc=db.cursor()
cur=dbc.execute("SELECT name,type FROM pet WHERE name='Missy'")
row=cur.fetchone()
print(row)
db.close()
```

output

```
('Missy','cat')
```

Sqlite/Python interface - Database Entry

SciNet

- Executing many inserts or updates can get tedious:

```
dbc.execute("INSERT INTO pet VALUES ('Missy','cat',4)")
dbc.execute("INSERT INTO pet VALUES ('Puppy','dog',2)")
dbc.execute("INSERT INTO pet VALUES ('Bingo','dog',2)")
dbc.execute("INSERT INTO pet VALUES ('Kitty','cat',3)")
```

- `executemany` helps:

```
dbc.executemany("INSERT INTO pet VALUES (?,?,?)",
  [('Missy','cat',4), ('Puppy','dog',2), ('Bingo','dog',2), ('Kitty','cat',3)])
```

  The argument should be a list of tuples.

- This is not only convenient, but also more efficient.

Sqlite/Python interface - Syncing the Database

- Multiple processes are allowed to open the database.
- But they will not see each others changes until:
- You commit them:

```
db.commit()
```

- You can then close the connection as well:

```
db.close()
```

Close does not imply commit! Commit, then close.

Database design

Database design

- How to setup your tables?
- Thinking ahead can avoid a lot of trouble.
  (integrity, maintenance, extensibility, . . .)
- Good to know some best practices.

First: Some Terminology

SciNet

- **Null**
  Fields can have missing (or unknown) values. These are known as NULL. A field can be explicitly defined as being not-null. This is not the same as an empty string or the value 0.
- **Schema**
  The definition of the columns of a table, with their data types and possible relations to other tables.
- **Row=Record=Relation=Tuple**
- **Column=Field=Attribute**
- **Key**
  A special field, or group of fields to identify an entity. Usually a unique integer. A *primary key* is a field that identify specific rows in the table. A *foreign key* identifies a row in another table. This allows you to define *relationships* between tables.
- **Index**
  A structure used in a RDBMS to improve data look-up. Does not influence the logical database design. Quite different from a key.

Database design

Keep in mind our simulation database. If we simply recorded input parameters and output variables, it could look something like this:

simulation_data

number	T	rho	seed	time	dt	P	E	system
343	340	0.1	13	10	0.01	10	-2.0	nitrogen
343	80	0.1	13	10	0.01	-1	-3.0	argon
343	340	0.1	13	10	0.01	10	-2.0	nitrogen
343	80	0.1	13	10	0.01	-1	-3.0	argon
343	340	0.1	13	10	0.01	10	-2.0	argon
343	80	0.1	13	10	0.01	-1.1	-3.0	argon
343	340	0.1	13	10	0.01	10	-2.0	argon
343	80	0.1	13	10	0.01	-1	-3.0	argon
343	340	0.1	13	10	0.01	10	-2.0	argon

Can you think of ways to improve the way this data is organized and stored?

Database design

SciNet

How about our simpler pet database?

pet

name	type	age
Missy	cat	4
Puppy	dog	3
Bingo	dog	2
Kitty	cat	2

Some issues with this table:

- `cat` and `dog` appear twice.
- How to add another two-year old dog named Puppy?
- What if we wanted to change `cat` to `domestic cat`?

These affect maintainability and usability.

Database design principle #1

Let every record be unique

E.g. not:

pet

name	type	age
Missy	cat	4
Kitty	cat	2
Puppy	dog	3
Bingo	dog	2
Kitty	cat	2

Why?

Because it is unclear if there are 2 Kitty's or if it's entered twice?

Note: The order of rows and of columns should not matter.

Database design principle #2

Give every record a unique key

name	type	age
Missy	cat	4
Kitty	cat	2
Puppy	dog	3
Bingo	dog	2

id	name	type	age
1	Missy	cat	4
2	Kitty	cat	2
3	Puppy	dog	3
4	Bingo	dog	2

You should define a `id` field as `INTEGER PRIMARY KEY`.
That field gets numbered automatically if you `INSERT` a `NULL` or do not set that field.

Example

```
db=sqlite3.connect("path/filename.sqlite3")
dbc=db.cursor()
dbc.execute("""CREATE TABLE pet( id INTEGER PRIMARY KEY,
                                 name TEXT, type TEXT, age INTEGER)""")
dbc.executemany("INSERT INTO pet VALUES (NULL,?,?,?)",
    [('Missy','cat',4), ('Kitty','cat',2), ('Puppy','dog',3), ('Bingo','dog',2) ])
db.commit()
```

SciNet

Let records be entities

Each table should be a list of things/entities, and fields should describe those things.

The `pet` tables seems to be okay in this respect.

The table `simulation_data` less so.

Note: Sometimes you'll need to deviate from this, such as with linking tables that express a many-to-many relationship.

Database design principle #4

Do not duplicate data

pet

id	name	type	age
1	Missy	cat	4
2	Kitty	cat	2
3	Puppy	dog	3
4	Bingo	dog	2

→ Use a *foreign key* →

pet

id	name	type_id	age
1	Missy	1	4
2	Kitty	1	2
3	Puppy	2	3
4	Bingo	2	2

where `type_id` is a key in another table:

pet2

type_id	name	...
1	cat	...
2	dog	...

How To Foreign Keys

- You can promiss yourself that you will only fill `type_id` in `pet` with values from `pet2`.
- Or, you can have SQLite enforce this.
- First, need to enable this after `connecting`:

```
db.execute("PRAGMA foreign_keys = ON")
```

- Then need to add to the table definition that certain fields are foreign keys, by appending the definition with e.g.:

```
FOREIGN KEY(type_id) REFERENCES pet2(type_id)
```

Foreign Keys Example

```
db.execute("PRAGMA foreign_keys = ON")

dbc.execute("CREATE TABLE pet2  (type_id INTEGER PRIMARY KEY, name TEXT)")
dbc.execute("""CREATE TABLE pet (id INTEGER PRIMARY KEY,
                                name TEXT,
                                type_id INTEGER,
                                age INTEGER,
                                FOREIGN KEY(type_id) REFERENCES pet2(type_id))""")

dbc.executemany("INSERT INTO pet2 VALUES (?,?)",[(1,'cat'),(2,'dog')])
dbc.executemany("INSERT INTO pet VALUES (NULL,?,?,?)",
            [('Missy',1,4), ('Puppy',2,3), ('Bingo',2,2), ('Kitty',1,2) ])

db.commit()
```

RecConstruct the original table

```
dbc.execute("SELECT * FROM pet JOIN pet2 ON pet.type_id=pet2.type_id").fetchall()
```

SciNet

Use clear table and field names

You cannot really comment a database, so your field identifiers should be easy to understand, i.e.

- Unambiguous yet concise;
- Plain english;
- Avoid spaces, special characters, and reserved SQL words;
- Stick to a convention.

Examples

`pet2` ⟶ `pet_type`

`age` ⟶ `age_in_years`

`CPP0311` ⟶ `cplusplus_03_or_11_compliance`

Database design principle #6

Column values should be atomic

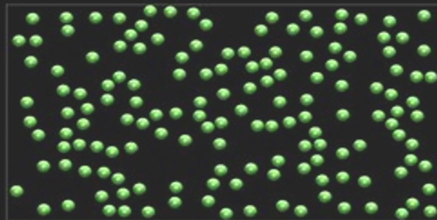
A field should contain only one quantity or thing, not an array of similar quantities, or a tuple of different quantities.

This may not seem to make much sense in some scientific applications, but in fact it makes all the more sense there: It gives you a good idea of what kind of data should *not* be in your database!

Use Case: Parameter sweep MD simulations

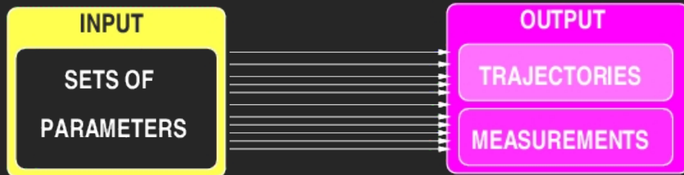
Parameter sweep MD simulations

SciNet



- We are to perform a set of simulations of Argon, each to give one measured value of the pressure.
- We want to determine the pressure as a function of the density or temperature.
- We also want to save a record of intermediate measurements.
- Varying container size, #particles, temperature $\Rightarrow$ lot of data
- Want to be able to select data points, e.g. at a given temperature, or in a range of number of particles, ...

Workflow



Database Design

SciNet

- What should we not store in our database?
- What are the entities?
- What are the attributes of these entities?
- How are the entities linked?

Entities

SciNet

- Parameters definitions
- Range of parameters
- Input parameter sets used
- Jobs
- Measured quantities

Step A: Parameter Definitions

SciNet

Design a table to store parameter definitions and ranges:

$$0.01 \text{ mol/l} \leq \text{density} \leq 10 \text{ mol/l}$$
$$80 \text{ K} \leq \text{temperature} \leq 340 \text{ K}$$

Other parameters are:

$$\text{Substance} = \text{Ar}$$
$$\text{N} = 343$$
$$\text{runtime} = 15 \text{ ps}$$
$$\text{timestep} = 0.025 \text{ ps}$$
$$\text{seed} = 13$$
$$\text{equil} = 5 \text{ ps}$$

Keep the possibility open that the table structure should allow these parameters should be allowed to vary as well, although we'll keep them fixed here.

Think about storing the units as well.

Create a schema/design, then implementing the creation of the database and table in Python.

Parameter Definition Table

SciNet

A_parameter_ranges.py

```
import sqlite3
db = sqlite3.connect("ljproject.sqlite3")
dbc = db.cursor()
dbc.execute("""CREATE TABLE parameter (
                parameter_id INTEGER PRIMARY KEY,
                name         TEXT,
                units        TEXT,
                lower_limit  REAL,
                upper_limit  REAL)""")
dbc.executemany("INSERT INTO parameter VALUES (NULL,?,?,?,?)",
               [('temperature',   'K',        80,   340),
                ('density',       'mol/l', 0.01,  10.0),
                ('number',        '1',       343,   343),
                ('runtime',       'ps',     15.0,  15.0),
                ('timestep',      'ps',    0.025, 0.025),
                ('seed',          '1',        13,    13)])
db.commit()
db.close()
```

Parameter Definitions - Check in sqlite3

```
$ sqlite3 ljproject.sqlite3
```

```
SQLite version 3.7.15.2 2013-01-09 11:53:05
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .mode column
sqlite> .header on
```

```
sqlite> select * from parameter;
```

```
  parameter_id  name          units       lower_limit  upper_limit
  ------------  ------------  ----------  -----------  -----------
  1             temperature   K           80.0         340.0
  2             density       mol/l       0.01         0.05
  3             number        1           343.0        343.0
  4             runtime       ps          80.0         80.0
  5             timestep      ps          0.025        0.025
  6             seed          1           13.0         13.0
```

```
sqlite> .exit
```

Check in sqlitebrowser

SciNet

```
$ sqlitebrowser ljproject.sqlite3
```

SQLite Database Browser - ljproject.sq3

File Edit View Help

Database Structure | Browse Data | Execute SQL

Table: parameters

New Record | Delete Record

	parameter id	name	units	lower limit	upper limit
1	1	temperature	K	80.0	340.0
2	2	density	mol/l	0.01	0.05
3	3	number	1	343.0	
4	4	runtime	ps	15.0	
5	5	timestep	ps	0.005	
6	6	seed	1	13.0	

Step B: Parameter Sets

SciNet

To go from the specified range to actual input parameters, we need Python's help.
(SQL does not generate ranges.)

- Let's design a database table `parameter_set` to hold values picked from the ranges in the parameters range table.
- Let's write a python script which generates 10 values within each of the parameters (within specified ranges), and which writes those to the database.

Parameter Sets Generation in Python

SciNet

```
import sqlite3, numpy
db    = sqlite3.connect("ljproject.sqlite3")
cur   = db.execute("SELECT lower_limit, upper_limit FROM parameters WHERE name='temperature'")
tvals = numpy.linspace(*cur.fetchone(), num=10)
cur   = db.execute("SELECT lower_limit, upper_limit FROM parameters WHERE name='density'")
dvals = numpy.linspace(*cur.fetchone(), num=10)
N     = db.execute("SELECT lower_limit FROM parameters WHERE name='number'").fetchone()[0]
t     = db.execute("SELECT lower_limit FROM parameters WHERE name='runtime'").fetchone()[0]
dt    = db.execute("SELECT lower_limit FROM parameters WHERE name='timestep'").fetchone()[0]
s     = db.execute("SELECT lower_limit FROM parameters WHERE name='seed'").fetchone()[0]
db.execute("""CREATE TABLE parameter_set (
              Seq                INTEGER PRIMARY KEY,
              temperature        REAL,
              density            REAL,
              number             INTEGER,
              runtime            REAL,
              timestep           REAL,
              seed               INTEGER )""")
settings = [(T,d,N,t,dt,s) for T in tvals for d in dvals]
db.executemany("""INSERT INTO parameter_set VALUES (NULL,?,?,?,?,?,?)""", settings)
db.commit()
db.close()
```

Parameter Sets - Check in sqlitebrowser

SciNet

SQLite Database Browser - ljproject.sq3

File Edit View Help

Database Structure | Browse Data | Execute SQL

Table: parameter_sets

New Record | Delete Record

	parameter_se	temperature	density	number	runtime	timestep	seed
1	1	80.0	0.01	343	15.0	0.005	13
2	2	80.0	1052631578947	343	15.0	0.005	13
3	3	80.0	2105263157895	343	15.0	0.005	13
4	4	80.0	3157894736842	343	15.0	0.005	13
5	5	80.0	4210526315789	343	15.0	0.005	13
6	6	80.0	5263157894737	343	15.0	0.005	13
7	7	80.0	6315789473684	343	15.0	0.005	13
8	8	80.0	7368421052632	343	15.0	0.005	13
9	9	80.0	8421052631579	343	15.0	0.005	13
10	10	80.0	9473684210526	343	15.0	0.005	13
11	11	80.0	0526315789474	343	15.0	0.005	13
12	12	80.0	1578947368421	343	15.0	0.005	13
13	13	80.0	2631578947368	343	15.0	0.005	13
14	14	80.0	3684210526316	343	15.0	0.005	13
15	15	80.0	4736842105263	343	15.0	0.005	13
16	16	80.0	5789473684211	343	15.0	0.005	13
17	17	80.0	6842105263158	343	15.0	0.005	13

< | 1 - 400 of 400 | > | Go to: 0

Step C: Input file and job generation

SciNet

To go from the input parameters to input files (and job scripts), we need Python's help again.

We can write another python script that reads the input parameters, and generate the input files and job script(s) in different directories.

Step B+C: Parameter set and job creation

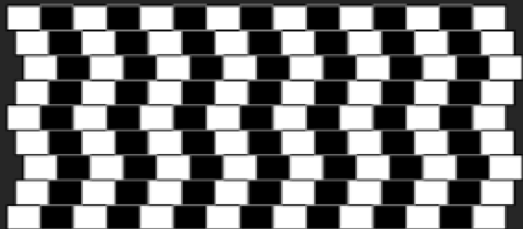
- These are a bunch of little serial jobs. Taking a whole 40-core Niagara node and using a batch job for each would be very inefficient.
- When you've got a bunch of serial jobs to do, you should look into:

  GNU Parallel

- O. Tange (2018): GNU Parallel 2018, March 2018, https://doi.org/10.5281/zenodo.1146014
- http://www.gnu.org/software/parallel/parallel_tutorial.html

Managing subjobs of different durations.

SciNet



- Surprisingly versatile (perl) script, especially for text input.
- Gets your many cases assigned to different cores and on different nodes without much hassle.
- Invoked using the `parallel` command, after doing:

```
module load gnu-parallel
```


- O. Tange, *GNU Parallel - The Command-Line Power Tool* ;login: **36** (1), 42-47 (2011)
- http://www.gnu.org/software/parallel/parallel_tutorial.html

GNU Parallel example

- Load the gnu-parallel module in your script.
- The "-j ..." flag indicates you wish GNU parallel to run 16 subjobs at a time.
- The "--nodes" parameter is important here to make sure all allocated cores are on the same node.

  (Running GNU Parallel across nodes is quite possible, but requires extra flags.)
- If you can't fit as many subjobs onto a node as there are cores due to memory constraints, specify a different value for the "-j" flag.
- Put all the commands for a given subjob onto a single line.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=40 # Niagara nodes
#SBATCH --time=1:00:00
#SBATCH --job-name=gnuparallell

module load gnu-parallel

# Run the code in parallel
parallel -j $SLURM_TASKS_PER_NODE <<EOF
cd jobdir1; ../app 1; echo "job 1 done"
cd jobdir2; ../app 2; echo "job 2 done"
...
cd jobdir200; ../app 200; echo "job 200 done"
EOF
```

What does GNU parallel do?

- GNU parallel assigns subjobs to the processors.
  - As subjobs finish it assigns new subjobs to the free processors.
  - It continues to assign subjobs until all subjobs in the subjob list are assigned.
- Consequently there is built-in load balancing!
- You can use GNU parallel across multiple nodes as well.
- It can also log a record of each subjob, including information about subjob duration, exit status, *etc.*

GNU Parallel syntax

SciNet

Some commonly used arguments for GNU parallel:

- `--jobs NUM`, sets the number of simultaneous subjobs.

  By default, parallel uses the maximum number of cores (16/80 on Teach/Niagara nodes).
  Same as `-j N`.

- `--joblog LOGFILE`, causes parallel to output a record for each completed subjob. The records contain information about subjob duration, exit status, and other goodies.

- `--resume`, when combined with `--joblog`, continues a GNU parallel job that was killed prematurely or did not finish all subjobs.

- `--pipe`, splits stdin into chunks given to the stdin of each subjob.

GNU Parallel - replacement strings

SciNet

- In the previous example, the commands GNU Parallel is to run were read from standard input.
- A lot of those commands contain the same parts.
- Instead, we can specify on the command line that part of the commands that is the same, with values for placeholders to be read as from standard input.

For instance, this:

```
$ parallel <<EOF
cd jobdir1;../app;echo "job 1 done"
cd jobdir2;../app;echo "job 2 done"
...
cd jobdir200;../app;echo "job 200 done"
EOF
```

is equivalent to

```
$ parallel 'cd jobdir{};../app {};echo "job{} done"'<<EOF
1
2
...
200
EOF
```

The replacement string here is {}.
There are other replacement strings that can remove extensions, paths, etc.

GNU Parallel - command line argument data

SciNet

We can also give the arguments on the command line instead of as standard input.

```
$ parallel 'cd jobdir{};../app {};echo "job{} done"' <<EOF
1
2
...
200
EOF
```

is equivalent to

```
$ parallel 'cd jobdir{};../app {} ;echo "job{} done"' ::: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

But bash has sequence expressions, so we can generate a list of 200 lines with `{1..200}`, and write:

```
$ parallel 'cd jobdir{};../app {};echo "job{} done"' ::: {1..200}
```

Multiple placeholders are also possible using this technique, e.g.

```
$ parallel 'echo {2} {1}' ::: {1..10} ::: {0..200..50}
```

which prints out all combinations of the elements of each set.

GNU Parallel and SQLite

SciNet

- GNU parallel can create entries for each subjob in a database.
- Then it can run those, filling in the job particularities.

```
$ parallel --sqlmaster sqlite3:///ljproject.sqlite/jobsexample echo ::: 1 2 ::: A B ::: ! ?
```

This stores the jobs in the file `ljproject.sqlite`, in table `jobexample`.

```
$ parallel --sqlworker sqlite3:///ljproject.sqlite/jobsexample
```

This executes the jobs in the database.

Keeps track of runtime, completion, parameters.

Use case: create jobs

- Using the database `ljproject.sqlite3` from step A, let's write a python script using GNU Parallel to generate a table of jobs in the database.
- It will also genereate a bash script (job script) that invokes GNU Parallel to run the jobs in the database.

BC_prepare_jobs.py

SciNet

```
import sqlite3,subprocess
from numpy import linspace
db        = sqlite3.connect("ljproject.sqlite3")
dbc       = db.cursor()
cur       = dbc.execute("SELECT lower_limit, upper_limit FROM parameters WHERE name='temperature'")
temperature_array = linspace(*cur.fetchone(), num=10)
cur       = dbc.execute("SELECT lower_limit, upper_limit FROM parameters WHERE name='density'")
density_array = linspace(*cur.fetchone(), num=10)
N         = dbc.execute("SELECT lower_limit FROM parameters WHERE name='number'").fetchone()[0]
seed      = dbc.execute("SELECT lower_limit FROM parameters WHERE name='seed'").fetchone()[0]
runtime   = dbc.execute("SELECT lower_limit FROM parameters WHERE name='runtime'").fetchone()[0]
timestep = dbc.execute("SELECT lower_limit FROM parameters WHERE name='timestep'").fetchone()[0]
dbc.execute("CREATE TABLE IF NOT EXISTS parameter_set (   \
              Seq                INTEGER PRIMARY KEY, \
              temperature        REAL,                \
              density            REAL,                \
              number             INTEGER,             \
              runtime            REAL,                \
              timestep           REAL,                \
              seed               INTEGER               )")
settings = [(T, d, N, runtime, timestep, seed) for T in temperature_array for d in density_array]
dbc.executemany("INSERT INTO parameter_set VALUES (NULL,?,?,?,?,?,?)", settings); db.commit(); db.close()
```

BC_prepare_jobs.py, continued

SciNet

```
gnu_parallel_prepare_command=" ".join(["parallel",
                                        "--sqlmaster", "sqlite3:///ljproject.sqlite3/jobs",
                                        "lj","{#}","Ar","{1}","{2}","{3}","{4}","{5}","{6}","{7}",
                                        "::: " + " ".join(map(str,temperature_array)),
                                        "::: " + " ".join(map(str,density_array)),
                                        "::: " + str(N),
                                        "::: " + str(runtime),
                                        "::: " + str(timestep),
                                        "::: " + str(seed),
                                        "::: " + str(runtime/2)])
gnu_parallel_runjobs_command=" ".join(["parallel",
                                        "-j", "${SLURM_TASKS_PER_NODE}",
                                        "--sqlworker", "sqlite3:///ljproject.sqlite3/jobs"])
subprocess.Popen(gnu_parallel_prepare_command, shell=True).wait()
f=open("D_runjobs.sh","w")
f.write("""#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=40
#SBATCH --time 1:00:00
#SBATCH --job-name C_runjobs
#SBATCH --output %x_%j.out
source setup""" + gnu_parallel_runjobs_command + "\n")
f.close()
```

Step D: run the jobs

SciNet

D_runjobs.sh

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=40
#SBATCH --time 1:00:00
#SBATCH --job-name C_runjobs
#SBATCH --output %x_%j.out

source setup

parallel -j ${SLURM_TASKS_PER_NODE:-4} --sqlworker sqlite3:///ljproject.sqlite3/jobs
```

```
$ sbatch D_runjobs.sh
```

Step E: Collecting Measurements

SciNet

- Assume all has run and output is in the directories.
- Let's design a database table to hold the measurements.
- And write a python script that fills the values from the json output files using Python's json module.

E_load_results.py

SciNet

```
import sqlite3
import json
db = sqlite3.connect("ljproject.sqlite3")
dbc = db.cursor()
dbc.execute(
"""CREATE TABLE IF NOT EXISTS measurement (
           Seq                INTEGER UNIQUE,
           substance          TEXT,
           total_energy       REAL,
           potential_energy   REAL,
           kinetic_energy     REAL,
           temperature        REAL,
           pressure           REAL
)""")
result_array = []
sql = "SELECT Seq from jobs"
```

```
for Seq, in dbc.execute(sql).fetchall():
  with open(str(Seq) + "/report.json") as f:
      data = json.load(f)
  result_array.append(
  [Seq,
   data["parameters"]["substance"],
   data["measurements"]["energy"]["total"]["value"],
   data["measurements"]["energy"]["potential"]["value"],
   data["measurements"]["energy"]["kinetic"]["value"],
   data["measurements"]["temperature"]["value"],
   data["measurements"]["pressure"]["value"]])
dbc.executemany(
 "INSERT INTO measurement VALUES (?,?,?,?,?,?,?)",
 result_array)
db.commit()
db.close()
```

Step F: Analysis

SciNet

- Select all data from the lowest temperature, and plot the pressure vs. density.

F_extract_results.py

```
import sqlite3
db = sqlite3.connect("ljproject.sqlite3")
cur=db.execute("SELECT P.temperature,P.density,M.pressure FROM parameter_set P"
    + " INNER JOIN measurement M on P.Seq=M.Seq")
print ("# Temperature  Density  Pressure")
oldT=0
for T,rho,P in cur.fetchall():
    if T!=oldT:
        print ("\n")
        oldT=T
    print ("%f %f %f"%(T,rho,P))
db.close()
```

Conclusion

Summary

SciNet

- RDBMS can organize data, both input and output.
- Independence of physical implementation.
- Easy data retrieval using SQL.
- Access to database through a host language: Python, C, ...
- You need to understand what your database is supposed to be able to do.
- Design before you write SQL.
- Make a small mock-up of your workflow before scaling out.
- Not everything in a database:
  Raw/binary data often lives outside of database.