Databases in research computing

Ramses van Zon

Dec 6, 8, 10, 2021

About this workshop

About this workshop

SciNet

Aim

The aim of this workshop 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

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

SciNet

- 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 measurements M JOIN parameters 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

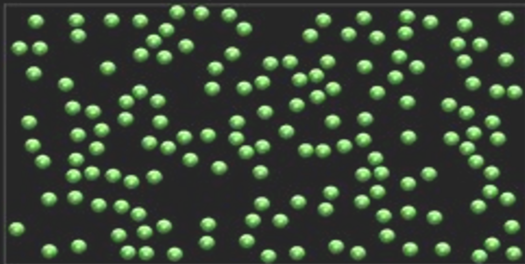
You can combine relational databases with more traditional approaches:

- 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 ($\to$ 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

```
$ ssh USER@niagara.scinet.utoronto.ca
$ cd $SCRATCH
$ cp -r /scinet/course/db21
$ cd db21
$ source setup
$ make
```

- The simulation program is called `lj` (for the *Lennard-Jones* force field).
- To run a simulation Argon 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. This output can be omitted by changing the last parameter 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.

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

SciNet

name	type
Missy	cat
Puppy	dog
Bingo	dog
Kitty	cat

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

Python/SQL mix

SciNet

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

- Outer part (`db.execute(...)`) is Python.
- Inner part is SQL.
- SQL the same in all ‘host’ languages.

SQL overview

SQL

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

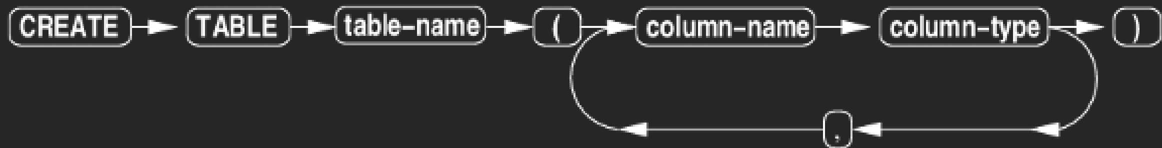
- This is a bit of SQL
- SQL=Structured Query Language
  - A 'fourth generation' programming language: verbose, expresses intention more than implementation
  - Case insensitive
  - Despite ANSI standard, every implementation differs in details
- Common commands

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

CREATE

Creates an empty table given a schema.

Partial syntax:



Example

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

Creates a table `pets` 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 pets
```

Removes the table ‘pets’ from the current database.

ALTER

Adds a column to an existing table.

(Can also rename a table)

Partial syntax:

Example

```
ALTER TABLE pets ADD COLUMN age INTEGER
```

Adds a column called 'age' that will contain integers to the table 'pets'.

INSERT

Inserts a row into a table.

Partial syntax:

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

Example

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

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

UPDATE

SciNet

Updates an existing row in a table.

Partial syntax:

Example

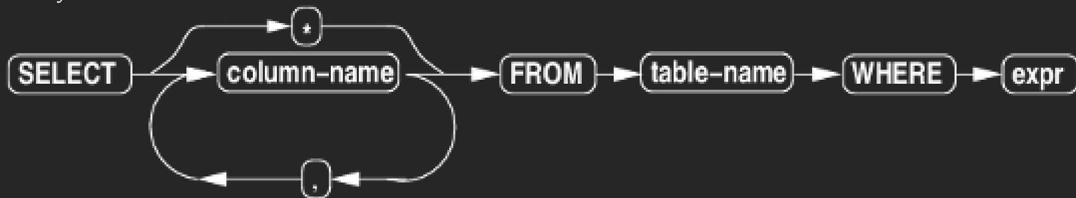
```
UPDATE pets SET type='cat' WHERE name='Nala'
```

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

SELECT

Looks up specific rows in a table.

Partial syntax:



Example

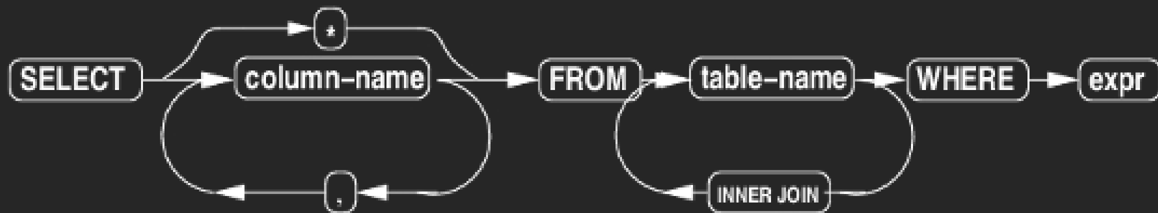
```
SELECT name,type,age FROM pets WHERE type='cat'
SELECT * FROM pets WHERE type='cat'
```

SELECT with JOIN

Looks up specific rows in combination of tables.

Instead of a table, you can specify joined tables.

Partial syntax:



Example

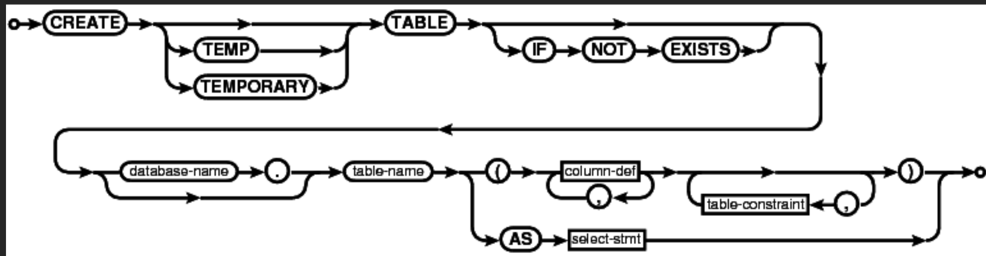
```
SELECT A.*, B.directory
    FROM simulations_attributes A INNER JOIN simulation_paths B
    WHERE A.simulation_id=B.simulation_id
```

Full SQL Syntax

Full Syntax Reference of These Commands:

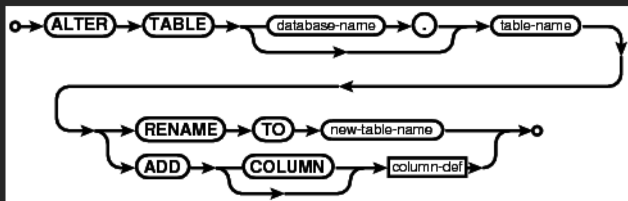
[www.sqlite.org]([www.sqlite.org]

SQLite Reference: CREATE

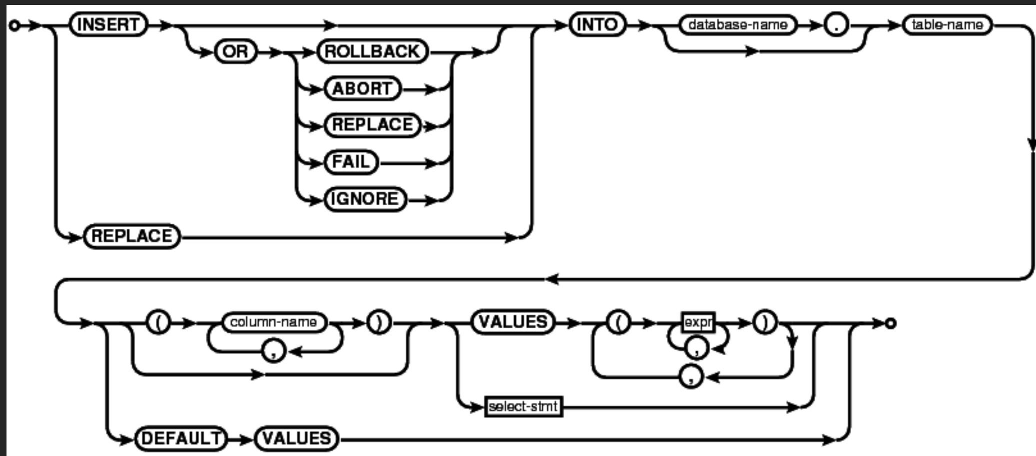


SQLite Reference: DROP

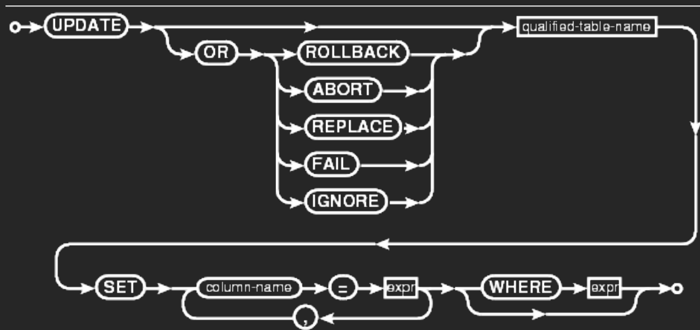
SQLite Reference: ALTER



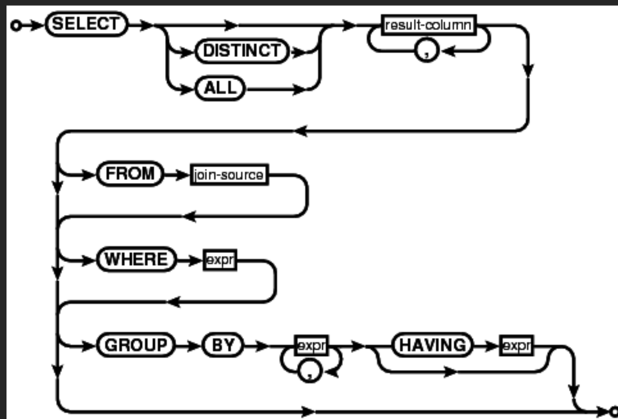
SQLite Reference: INSERT



SQLite Reference: UPDATE



SQLite Reference: SELECT



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`.

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

- You are then supposed to get the cursor

```
cur = con.cursor()
```

Much of what you'd do with cur, you can do with con.
You should know that this is non-standard, but we'll use it here anyway, because it is clearer.

Sqlite-Python interface: Query the Database

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

```
cur = con.execute("CREATE TABLE MY_TABLE ...")
cur = con.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/databasefilename.sqlite3')
c=db.execute("SELECT name,type FROM pets WHERE name='Missy'")
r=c.fetchone()
print(r)
db.close()
```

output

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

Sqlite/Python interface - Database Entry

SciNet

- Executing many inserts or updates can get tedious:

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

- `executemany` helps:

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

The argument should be a list of tuples.

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:

```
con.commit()
```

- You can then close the connection as well:

```
con.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

SciNet

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

How about our simpler pet database?

pets

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

SciNet

Let every record be unique

E.g. not:

pets

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")
db.execute("""CREATE TABLE pets( id INTEGER PRIMARY KEY,
                                name TEXT, type TEXT, age INTEGER)""")
db.executemany("INSERT INTO pets VALUES (NULL,?,?,?)",
    [('Missy','cat',4), ('Kitty','cat',2), ('Puppy','dog',3), ('Bingo','dog',2) ])
db.commit()
```

Database design principle #3

Let records be entities

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

The `pets` 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.

Do not duplicate data

pets

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

→ Use a *foreign key* →

pets

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:

pets2

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

How To Foreign Keys

SciNet

- You can promiss yourself that you will only fill `type_id` in `pets` with values from `pets2`.
- 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 pets2(type_id)
```

Foreign Keys Example

SciNet

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

db.execute("CREATE TABLE pets2  (type_id INTEGER PRIMARY KEY, name TEXT)")
db.execute("""CREATE TABLE pets (id INTEGER PRIMARY KEY,
                                 name TEXT,
                                 type_id INTEGER,
                                 age INTEGER,
                                 FOREIGN KEY(type_id) REFERENCES pets2(type_id))""")

db.executemany("INSERT INTO pets2 VALUES (?,?)",[(1,'cat'),(2,'dog')])
db.executemany("INSERT INTO pets VALUES (NULL,?,?,?)",
            [('Missy',1,4), ('Puppy',2,3), ('Bingo',2,2), ('Kitty',1,2) ])

db.commit()
```

Construct the original table

```
db.execute("SELECT * FROM pets JOIN pets2 ON pets.type_id=pets2.type_id").fetchall()
```

Database design principle #5

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

pets2 ⟶ pet_types

age ⟶ age_in_years

CPP0311 ⟶ cplusplus_03_or_11_compliance

Column values should be atomic

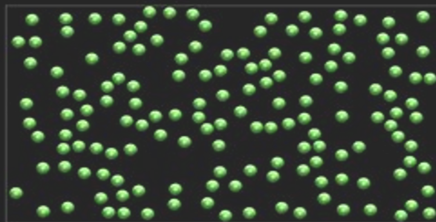
A field should contain only one quantity or thing, not an array of similar quantities, or a tuple of different quantities.

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

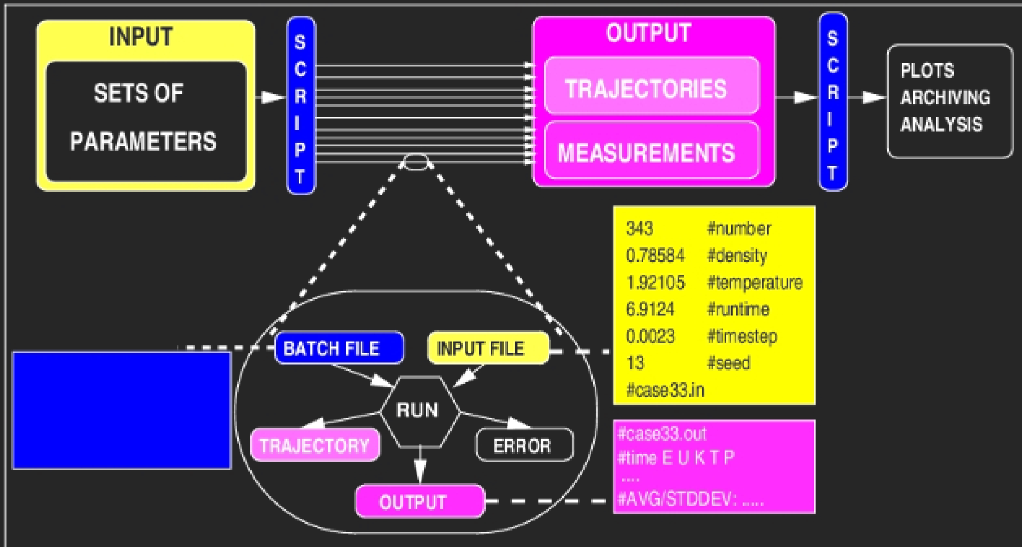
Parameter sweep MD simulations

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

Assignment 1: 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.

Assignment 2: Parameter generation

SciNet

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

- Design a database table to hold values picked from the ranges in the parameters range table.
- Write a python script which generates 10 values within each of the parameters (within specified ranges), and which writes those to the database.