COMP9315 25T1: Assignment 2

Multi-attribute Linear Hashed Files

Aims

This assignment aims to give you an understanding of

- how database files are structured and accessed.
- how multi-attribute hashing is implemented.
- how linear hashing is implemented.

The goal is to build a simple implementation of a linear-hashed file structure that uses multi-attribute hashing.

Summary

Deadline: **Friday 20:59:59 25th April (Sydney Time).**

Late Penalty: 5% of the max assessment mark per-day reduction, for up to 5 days.

Marks: This assignment contributes **20 marks** toward your total mark for this course.

Submission: **Moodle** > Assignment > Assignment 2> upload **ass2_ zID.zip.**

The **ass2_ zID.zip** file must contain your Makefile plus all of your *.c and *.h files. Details on how to build the **ass2_ zID.zip** file are given below.

Note: Make sure that you read this assignment specification carefully and completely before starting work on the assignment. Questions which indicate that you haven't done this will simply get the response "Please read the spec". This assignment does not require you to do anything with PostgreSQL.

Introduction

Linear hashed files and multi-attribute hashing are two techniques that can be used together to produce hashed files that grow as needed and which allow all attributes to contribute to the hash value of each tuple. See the course notes and lecture slides for further details on linear hashed files and multi-attribute hashing.

In our context, multi-attribute linear-hashed (MALH) files are file structures that represent one relational table, and can be manipulated by three commands:

❖ Create command

Creates MALH files by accepting four command line arguments:

- the name of the relation
- the number of attributes
- the initial number of data pages (rounded up to nearest 2n)
- the multi-attribute hashing choice vector

This gives you storage for one relation/table, and is analogous to making an SQL data definition like:

```
create table R ( a1 text, a2 text, ... an text );
```

Note that, internally, attributes are indexed 0..n-1 rather than 1..n. The following example of using create makes a table called **abc** with 4 attributes and 8 initial data pages:

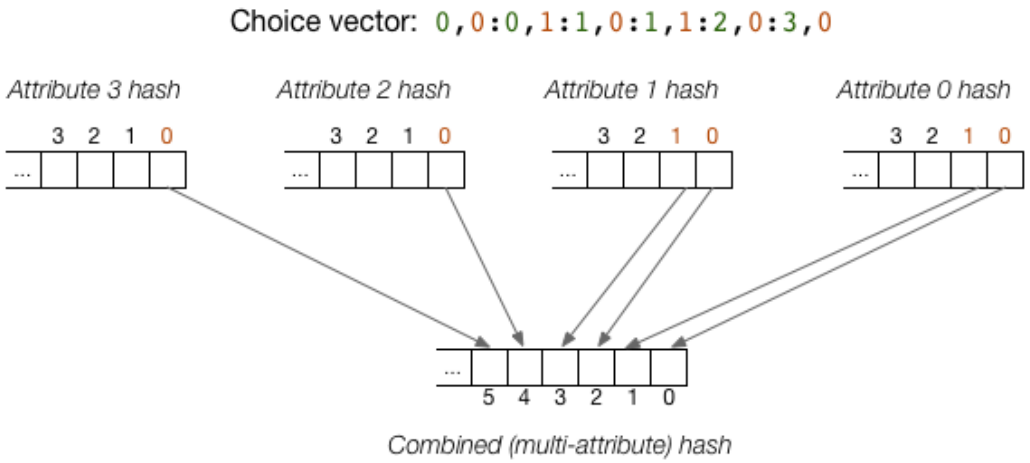
```
$ ./create abc 4 6 "0,0:0,1:1,0:1,1:2,0:3,0"
```

Note that 6 will be rounded up to the nearest 2^n (i.e. to 8). If we'd written 8, we would have gotten the same result.

The choice vector (fourth argument above) indicates that

- bit 0 from attribute 0 produces bit 0 of the MA hash value
- bit 1 from attribute 0 produces bit 1 of the MA hash value
- bit 0 from attribute 1 produces bit 2 of the MA hash value
- bit 1 from attribute 1 produces bit 3 of the MA hash value
- bit 0 from attribute 2 produces bit 4 of the MA hash value
- bit 0 from attribute 3 produces bit 5 of the MA hash value

The following diagram illustrates this scenario:



The above choice vector only specifies 6 bits of the combined hash, but combined hashes contain 32 bits. The remaining 26 entries in the choice vector are automatically generated by cycling through the attributes and taking bits from the high-order hash bits from each of those attributes.

❖ Insert command

Reads tuples, one per line, from standard input and inserts them into the relation specified on the command line. Tuples all take the form v1,v2,..., vn. The values can be any sequence of characters except ',', '%' and '?'.

The bucket where the tuple is placed is determined by the appropriate number of bits of the combined hash value. If the relation has 2^d data pages, then d bits are used. If the specified data page is full, then the tuple is inserted into an overflow page of that data page.

❖ Query command

The query command allows you to run selection and projection queries over a given relation. It supports wildcard and pattern matching, finds all tuples in either the data pages or overflow pages that match the query, as well as flexible attribute projection without **distinct**.

The general usage is:

```
$ ./query [-v] 'a1,a2,...' from RelName where 'v1,v2,...'
```

- 'a1,a2,...' (or '*'): a sequence of 1-based attribute indexes used for projection, can be '*' to indicate all attributes. The minimal 'a' value is '0'.
- 'v1,v2,...': a sequence of attribute values used for selection.

Note that: The projection and selection strings are wrapped in quotes to prevent the shell from misinterpreting characters as wildcards or splitting the values on commas, these quotes are handled automatically by the shell. Your code **does not** need to perform any extra parsing or stripping of quotes.

Each value vi in the selection tuple can be:

- Literal value: A specific value that must match exactly in the corresponding attribute position. (e.g., 'abc' matches 'abc', '10' matches '10')
- Single question mark '?': Matches any literal value in the corresponding attribute position. (e.g., '?' matches 'abc', '?' matches '10')
- Pattern string containing '%': A string that includes one or more '%', where each '%' matches zero or more characters. Enables flexible pattern-based matching. (e.g., 'ab%' matches any literal value starting with 'ab', such as 'abc', 'ab123')

Some example query commands, and their interpretation are given below.

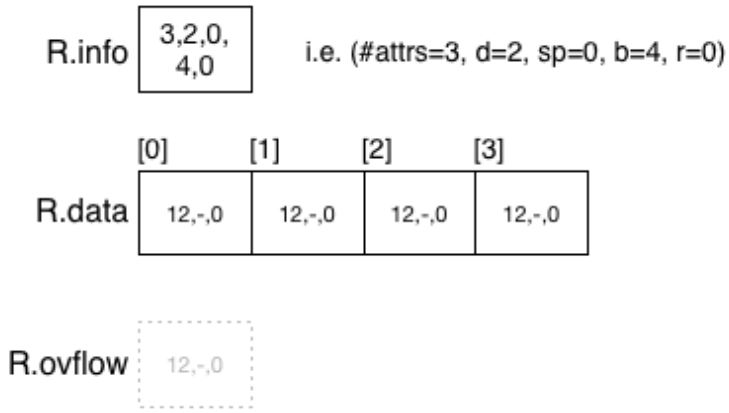
```
$ ./query '*' from R where '?,?,?'
        # matches any tuple in the relation R

$ ./query '3,1' from R where '10,?,?'
        # projects attributes 1 and 3 (in order) from tuples where the value
of attribute 0 is 10

$ ./query '*' from R where '?,%ab%,?'
        # matches any tuple where attribute 1 contains 'ab'
```

A MALH relation R is represented by three physical files:

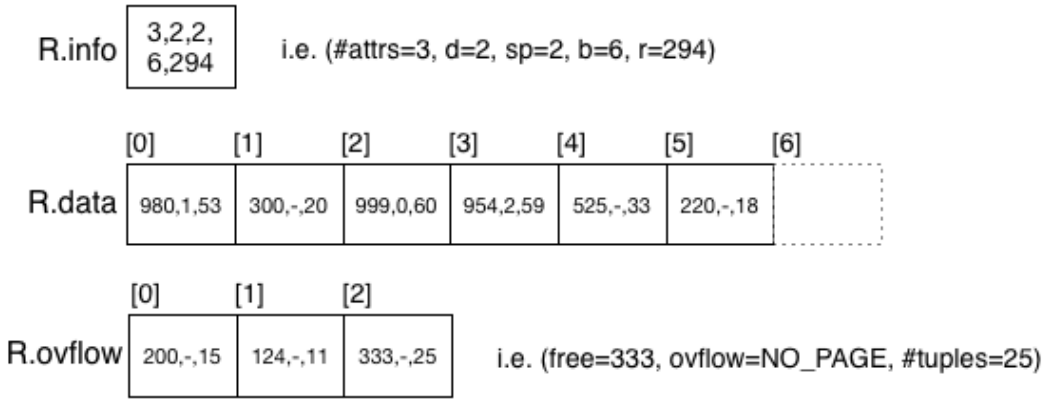
- **R.info** containing global information such as
  - a count of the number of attributes
  - the depth of main data file (d for linear hashing)
  - the page index of the split pointer (sp for linear hashing)
  - a count of the number of main data pages
  - the total number of tuples (in both data and overflow pages)

    - the choice vector (cv for multi-attribute hashing)
- **R.data** containing data pages, where each data page contains
    - offset of start of free space
    - overflow page index (or NO_PAGE if none)
    - a count of the number of tuples in that page
    - the tuples (as comma-separated C strings)
- **R.ovflow** containing overflow pages, which have the same structure as data pages

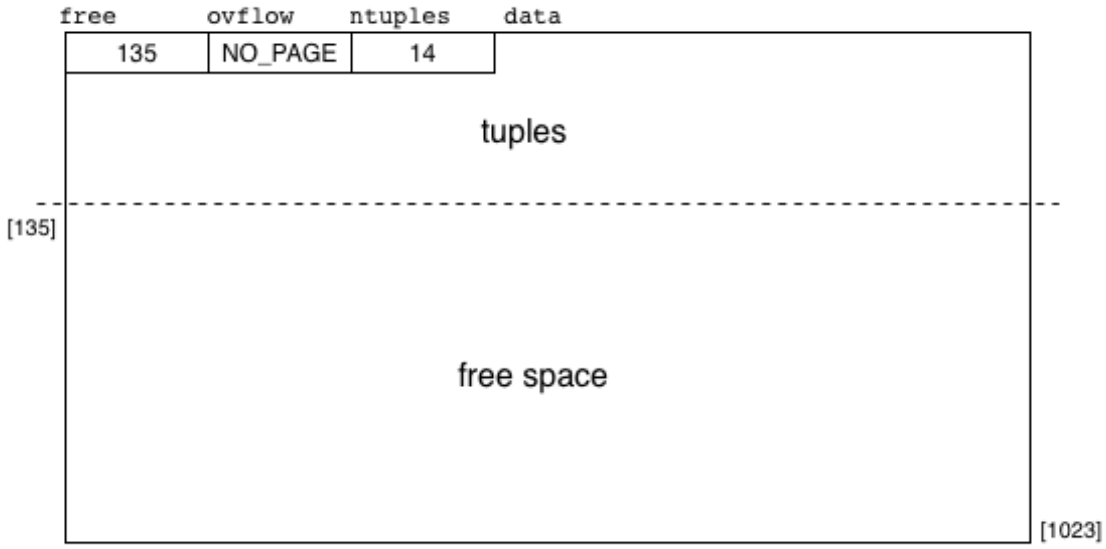
When a MALH relation is first created, it is set to contain a 2^n pages, with depth d=n and split pointer sp=0. The overflow file is initially empty. The following diagram shows an MALH file **R** with initial state with n=2.



After 294 tuples have been inserted, the file might have the following state (depending on field value distributions, tuple sizes, etc):



Pages in MALH files have the following structure: a header with three unsigned integers, strings for all of the tuple data, free space containing no tuple data. The following diagram gives an example of this:



We have developed some infrastructure for you to use in implementing multi-attribute linear-hashed (MALH) files. You may use this infrastructure or replace parts of it (or all of it) with your own, but your MALH files implementation must conform to the conventions used in our code. In particular, you should **PRESERVE ALL EXISTING INTERFACES** to the supplied modules (e.g. Reln, Page, Selection, Projection, Tuple).

DO NOT MODIFY OR DELETE any existing interfaces-you may only add new ones in exceptional cases. Ensure that your submitted ADTs work with the supplied code in the create, insert, and query commands.

Setting Up

You should make a working directory for this assignment and put the supplied code there. Read the supplied code to make sure that you understand all of the data types and operations used in the system.

```
$ mkdir Your/ass2/Directory

$ cd Your/ass2/Directory

$ unzip /web/cs9315/25T1/assignments/ass2/ass2.zip
```

You should see the following files in the directory:

- create.c ... a main program that creates a new MALH relation
- dump.c ... a main program that lists all tuples in an MALH relation
- insert.c ... a main program that reads tuples and insert them
- query.c ... a main program that finds tuples matching a PMR query and projects them onto specified attributes
- stats.c ... a main program that prints info about an MAH relation
- gendata.c ... a main program to generate random tuples
- bits.h, bits.c ... an ADT for bit-strings
- chvec.h, chvec.c ... an ADT for choice vectors
- defs.h ... defines global constants and types
- hash.h, hash.c ... the PostgreSQL hash function
- page.h, page.c ... an ADT for data/overflow pages
- select.h, select.c ... an ADT for selection scanners (incomplete)
- project.h, project.c ... an ADT for projection operators (incomplete)
- reln.h, reln.c ... an ADT for relations (partly complete)
- tuple.h, tuple.c ... an ADT for tuples (partly complete)
- util.h, util.c ... utility functions

This gives you a partial implementation of MALH files; you need to complete the code so that it provides the functionality described below.

The supplied code actually produces executables that work somewhat, but are missing a working query scanner implementation (from select.c), a proper MA hash function (from tuple.c), and splitting and data file increase (from reln.c). Effectively, they give a static hash file structure with overflows.

To build the executables from the supplied code, do the following:

```
$ make
    gcc -Wall -Werror -g -std=c99   -c -o create.o create.c
    …
    gcc   gendata.o select.o project.o page.o reln.o tuple.o util.o chvec.o    hash.o    bits.o
/usr/lib/x86_64-linux-gnu/libm.so   -o gendata
```

This should not produce any errors on the CSE servers.

Once you have the executables, you could build a sample database as follows:

```
$ ./create  R  3  4  "0,0:0,1:0,2:1,0:1,1:2,0"

    cv[0] is (0,0)
    cv[1] is (0,1)
    cv[2] is (0,2)
    ...
    cv[31] is (1,23)
```

This command creates a new table called R with 3 attributes. It will be stored in files called R.info, R.data and R.ovflow. The data file initially has 4 pages (so depth d=2). The overflow file is initially empty. The lower-order 6 bits of the choice vector are given on the command line; the remaining bits are auto-generated. Given the file size (4 pages), only two of the hash bits are actually needed.

You could check the status of the files for table R via the stats command:

```
$ ./stats  R

    Global Info:
    #attrs:3  #pages:4  #tuples:0  d:2  sp:0
    Choice vector
    0,0:0,1:0,2:1,0:1,1:2,0:0,31:1,31:2,31:0,30:1,30:2,30:0,29:1,29:2,29:0,28:1,2
8:2,28:0,27:1,27:2,27:0,26:1,26:2,26:0,25:1,25:2,25:0,24:1,24:2,24:0,23:1,23
    Bucket Info:
    #Info on pages in bucket
    (pageID,#tuples,freebytes,ovflow)
    [ 0]  (d0,0,1012,-1)
    [ 1]  (d1,0,1012,-1)
    [ 2]  (d2,0,1012,-1)
    [ 3]  (d3,0,1012,-1)
```

Since the file is size 2^d, the split pointer sp = 0. The rest of the global information should be self explanatory, as should the choice vector. The bucket info shows a quadruple for each page; since there are no overflow pages (yet), only data pages appear. The pageID value in each quad consists of the character 'd' (indicating a data file), plus the page index. Each page is 1024 bytes long, which includes a small header, plus 1012 bytes of free space for tuples. There are currently zero tuples in any of the pages. The overflow page IDs are all -1 (for NO_PAGE) to indicate that no data page has an overflow page.

You can insert data into the table using the insert command. This command reads tuple from its standard input and inserts them into the named table. For example, the command below inserts a single tuple into the R MALH files:

```
$ echo "100,abc,xyz" | ./insert R

    hash(100) = 00011100 00101000 10100111 11101100
```

The **insert** command prints the hash value for the tuple (based on just the first attribute), and then inserts it into the file. Since the table is currently empty, this tuple will be inserted into page 0. Why page 0? You should be able to answer this by knowing the depth and the hash value. If you then check with the **stats** command you will see that there is a single tuple in the files, and it's in page 0.

Typing many individual tuples is tedious, so we have provided a command, gendata, which can generate tuples appropriate for a given table. It takes four comand line arguments, only two of which are compulsory: the number of tuples to generate, and the number of attributes in each tuple. a sample usage:

```
$ ./gendata  5  3

    1,triangle,pith
    2,comet,signature
    3,aeroplane,mum
    4,dog,win
    5,finger,desk
```

This generates five tuples, each with three attributes. The first attribute is a unique ID value; the other attributes are random words. You can modify the starting ID value and the seed for the random number generator from the command line.

You could use gendata to generate large numbers of tuples, and insert them as follows:

```
$ ./gendata 250 3 101 | ./insert R
    hash(101) = 11110100 01100100 11010000 00110000
    hash(102) = 00100101 10100110 10100001 11100100
    ...
    hash(349) = 01101101 01100101 00011111 10100111
    hash(350) = 10011011 01100101 01111001 11001000
```

This will insert 250 tuples into the table, with ID values starting at 101. You can check the final state of the database using the **stats** command. It should look something like:

```
$ ./stats R
    Global Info:
    #attrs:3  #pages:4  #tuples:251  d:2  sp:0
    Choice vector
    0,0:0,1:0,2:1,0:1,1:2,0:0,31:1,31:2,31:0,30:1,30:2,30:0,29:1,29:2,29:0,28:1,2
8:2,28:0,27:1,27:2,27:0,26:1,26:2,26:0,25:1,25:2,25:0,24:1,24:2,24:0,23:1,23
    Bucket Info:
    #Info on pages in bucket
    (pageID,#tuples,freebytes,ovflow)
    [ 0]  (d0,56,4,0) -> (ov0,15,737,-1)
    [ 1]  (d1,57,2,3) -> (ov3,2,981,-1)
    [ 2]  (d2,59,1,2) -> (ov2,2,976,-1)
    [ 3]  (d3,54,7,1) -> (ov1,6,905,-1)
```

This shows that each data page has one overflow page, and that each data page has roughly the same number of tuples. The bucket starting at data page 0 has a few more tuples than th other buckets, because it has more tuples (15) in the overflow page. Note that page IDs in the overflow pages are distinguished by starting with "ov". Note also that e.g. the data page at position 3 in the data file has an overflow page at position 1 in the overflow file; this is because page 3 filled up before pages 1 and 2.

One other thing to notice here is that the file has not expanded. It still has the 4 original data pages. Even if you added thousands of tuples, it would still have only 4 data pages. This is because linear hashing is not yet implemented. Implementing it is one of your tasks.

You could then use the **query** command to search for tuples using a command like:

```
$ ./query '2,1' from R where '101,?,?'
```

This aims to find any tuple with 101 as the ID value (the first attribute), and projects the result on attributes 2 and 1 (in that order); there will be exactly one such tuple, since ID values are unique. This returns no solutions because query scanning is not yet implemented. Implementing it is another of your tasks.

Task 1: Multi-attribute Hashing

The current hash function does not use the choice vector to produce a combined hash value. It simply uses the hash value of the first attribute (the ID value) to generate a hash for the tuple. Your first task is to modify the tupleHash() function to use the relevant bits from each attribute hash value to form a composite hash. The choice vector determines the "relevant" bits. You can find more details on how a multi-attribute hash value is produced in the lecture slides and notes.

Task 2: Querying (Selection and Projection)

The selection (scan) data type is defined in select.c and select.h, and the projection data type is defined in project.c and project.h. These data types are used exclusively within query.c. Currently, both data types are incomplete. Your task is to design appropriate data structures for selection and projection, and implement the necessary operations on them.

Specifically, in select.c, you are required to implement n-dimensional partial-match retrieval (n-d pmr).This includes support for pattern matching, which can be implemented directly in select.c or by calling a function defined in .c files of other data types. In project.c, your task is to implement projection without **distinct**, which involves selecting and possibly reordering attributes from tuples.

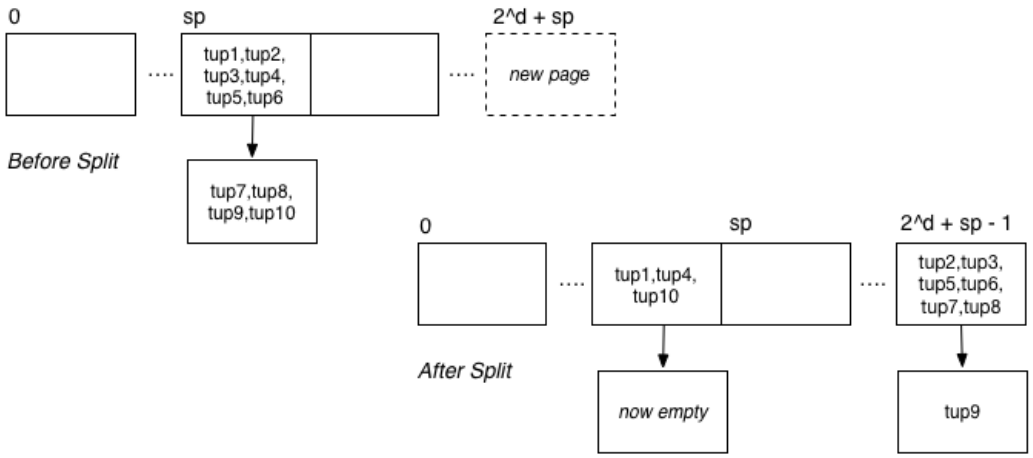
The functions currently provided in select.c and project.c contain rough approximations to the algorithms you will need to build; you can find more details in the lecture slides and course notes. Most of the helper functions you'll need are in other data types, but you can add any others that you find necessary.

Task 3: Linear Hashing

As noted above, the current implementation is essentially a static version of single-attribute hashing. You need to add functionality to ensure that the file expands after every c insertions, where c is the page capcity c = floor(B/R) ≈ 1024/(10*n) where n is the number of attributes. Add one page at the end of the file and distribute the tuples in the "buddy" page (at index 2^d less) between the old and new pages. Determine where each tuple goes by considering d+1 bits of the hash value. This will involve modifying the addToRelation() function, and will most likely require you to add new functions into the reln.c file (and maybe other files).

You can simplify the standard version of linear hashing by not removing overflow pages from the overflow chain of the data page they are attached to. This may result in some data pages having multiple empty overflow pages; this is ok if they are eventually used to hold more tuples.

The following diagram shows an example of what might occur during a page split:



How we Test your Submission

You need to submit a single zip file containing all of the code files that are needed to build the create, dump, insert, query and stats commands.

Note that we will use the original versions of create.c, dump.c, insert.c, query.c, stats.c, and gendata.c for testing your code. This means that any functions you write must use the same interface as defined in the ADT *.h files. **DO NOT MODIFY OR DELETE** any existing interfaces in the ADTs.

When you want to submit your work, make sure to follow the steps below. Failing to do so may result in your code not appearing in the correct directory during testing, and it will fail to compile:

```
$ cd Your/ass2/Directory
$ zip ass2_zID.zip Makefile bits.h chvec.h defs.h hash.h page.h \
select.h  project.h reln.h tuple.h util.h bits.c chvec.c hash.c page.c \
select.c project.c reln.c tuple.c util.c
```

Once you have generated the ass2_zID.zip file, you can submit it via **Moodle**.

We will compile your submission for testing as follows:

```
$ unzip YourAss2.zip
$ tar xf OurMainPrograms.tar
    # extracts our copies of ...
    # create.c dump.c insert.c query.c stats.c
$ make
    # should produce executables ...
    # create dump insert query stats
```

We will then run a range of tests to check that your program meets the requirements given above.

Since we are using the original create.c, etc., your code must work with them. The easiest way to ensure this is to not change these files while you're working on the assignment.

Assignment Submission

Submission

- You need to submit a single zip file containing all of the code files that are needed to build the create, dump, insert, query and stats commands via **Moodle**.
    - Noted, we will use the original versions of create.c, dump.c, insert.c, query.c, stats.c, and gendata.c for testing your code. This means that any functions you write must use the same interface as defined in the ADT *.h files. **DO NOT MODIFY OR DELETE** any existing interfaces in the ADTs. For more details, please refer to the Section 'How we Test your Submission'.
- Please name your ZIP file in the following format to submit: ass2_ **zID**.zip (e.g., ass2_ z5000000.zip).

Note:

1. If you have problems relating to your submission, please email to xingyu.tan@unsw.edu.au.
2. If there are issues with Moodle, send your assignment to the above email with the subject title "<zid> COMP9315 Ass2 Submission".

Late Submission Penalty

- 5% of the max mark (20 marks) will be deducted for each additional day.
- Submissions that are more than five days late will not be marked.

Plagiarism

The work you submit must be your own work. Submission of work partially or completely derived from any other person or jointly written with any other person is not permitted. The penalties for such an offence may include negative marks, automatic failure of the course and possibly other academic discipline.

All submissions will be checked for plagiarism. The university regards plagiarism as a form of academic misconduct and has very strict rules. Not knowing the rules will not be considered a valid excuse when you are caught.

- For UNSW policies, penalties, and information to help avoid plagiarism, please see: https://student.unsw.edu.au/plagiarism.
- For guidelines in the online ELISE tutorials for all new UNSW students: https://subjectguides.library.unsw.edu.au/elise/plagiarism.