RMIT
School of Computer Science & IT

COSC2406/2407 Database Systems

Assignment 2: Query Evaluation
Due: Friday, 22 May 2009, 5:30pm
Marks: This assignment is worth 20% of your overall mark
This is an individual assignment

1 Introduction

In this assignment, you will implement equality join algorithms, and test your algorithms on files of words.

The "Database Systems" blackboard contains further announcements and a list of frequently asked questions. You are expected to check the discussion board on daily basis. Login through `http://www.rmit.edu.au/learninghub`.

Have a look at the files `characters` and `guilds` in `/scratch/DatabaseSystems/DATA` on `yallara`. The first file contains a relation storing data about characters in an online game. The second contains the guilds that are available in the game. The last field in `characters` is a foreign key, referencing the `guilds` guild entries.

Your task is to join the files pairwise using different algorithms, to conduct experiments on the relative efficiency of the approaches, and to present your findings in a report.

Plagiarism

All assignments will be checked with plagiarism-detection software; any student found to have plagiarised will be subject to disciplinary action as described in the course guide. Plagiarism includes submitting code that is not your own or submitting text that is not your own. Submitting one comment in your C code or a sentence from someone else's report is plagiarism, and plagiarism includes submitting work from previous years. Allowing others to copy your work is also plagiarism. All plagiarism will be penalised; there are no exceptions and no excuses. For further information, please see: `http://www.cs.rmit.edu.au/students/integrity/`. **You have been warned.**

General Requirements

This section contains information about the general requirements that your assignment must meet. *Please read all requirements carefully before you start.*

- You must implement your programs in C. Your programs must be well written, using good coding style and including appropriate use of comments. Your markers will look at your source code. Coding style will form part of the assessment of this assignment.

- Your programs may be developed on any machine, but must compile *and* run on `yallara`.
- Any code you submit must be able to be built using a single `Makefile` with the command `make all` . If your marker cannot compile your programs due to the absence of a `Makefile`, you risk yielding zero marks for the coding component of your assignment. Your Makefile must use the `gcc` compiler.
- Paths must not be hard-coded.
- Diagnostic messages must be output to `stderr`.
- Your assignments will be submitted using the turnin command on `yallara`.
- Parts of this assignment will ask you to analyse your results, and to write about your conclusions in a report. Your report must be a *plain text* file. It may not be word processor documents or a text file with markup (such as LaTeX or XML). In addition, text must be wrapped to a maximum of 80 characters per line. Files that do not meet this requirement may not be marked.
- Please ensure that your submission obeys the file naming rules specified in the tasks below. File names are case sensitive, i.e. if it is specified that the file name is `gryphon`, then that is exactly the file name you must submit; `Gryphon`, `GRYPHON`, `griffin`, and anything else but `gryphon` will be rejected. If you do not obey the file naming rules, you risk yielding zero marks for the corresponding task.
- For some tasks, you need to generate large output files. If you do not have enough space in your own account, you can create a directory under `/scratch/DatabaseSystems` on `yallara`. The directory that you create should have the same name as your username. Please bear in mind that this directory should *only* be used to store temporary output files from your programs. You must not store source code or other files that you wish to keep here, as the lifespan of these directories is only short-term. Also, please clean up your subdirectories every day (i.e. delete them before you log out of `yallara`) to free up disk space.
- For the sections that require you to write C code and involve timing experiments, you must use the Solaris library call `gethrtime()`. Take care to include the correct sections of code in the timings, so that you can make fair comparisons.
- **Important:** You must run all your experiments on `yallara`, because the disk that hosts the file is local to this machine.

File Descriptors

Parts of this assignment may require that you open many files simultaneously. You should be aware of the following issues when designing your programs:

- When your shell session is initialised on yallara, the maximum number of file descriptors is subject to a soft limit of 256. You can check the current setting using the `limit` command. You should increase this number to 1003 by running the command `limit descriptors`

1003. (Note: this new limit will only apply for the current shell session. When you next log in, the usual soft limit will be in place again. You could make the change automatic by adding the new `limit` command as the last line of your `.cshrc` file.)

- The Solaris implementation of the standard C library functions for files, including `fopen()`, `fclose()`, `fread()`, `fwrite()` and `fseek()`, impose a hard upper limit of 256 file descriptors. As a result you will need to use the Solaris specific library functions when manipulating files in this assignment: `open()`, `close()`, `read()`, `write()` and `lseek()`. While these behave similarly to the corresponding standard C functions, you should read the `man` pages of the functions carefully to check for differences.

Programming Tasks

Number of Buffers

Your join algorithms will run with a specified *number of buffers* that can be used for processing. (Note: a buffer is equivalent to a page.) Assume that a single buffer can store:

- 2 records from the `characters` relation; OR
- 6 records from the `guilds` relation; OR
- 1 joined record (i.e. an output tuple composed of the unique fields of a `characters` and a `guilds` tuple).

For example, when the specified number of available buffers is 2, the following are all examples of what could be held in memory at a single point in time: 4 records from `characters`; or, 12 records from `guilds`; or, 2 record from `characters` and 6 records from `guilds`; or other suitable combinations.

You need to make sensible assumptions about requirements for input and output buffers for each of the join algorithms. If these are required, they contribute to the total number of buffers that are available for your join to use. Be sure to state these assumptions in your report.

Relation Schemas

You should use the following schemas for the records in the data files:

`characters` — `name:char[38], race:int, class:int, id:int, guildID:int`

`guilds` — `guildID:int, name:char[14]`

You may use the native C integer type to represent `int` specifications from the schemas.

1.1 Nested Loops Join (35%)

Implement a nested loops join in C that takes as input the total number of available buffers, and the names of the two files to join. The executable must be called `NestedLoopJoin`, and must be run as follows:

```
./NestedLoopJoin [-d] <buffers> <path>/characters <path>/guilds
```

As output, when the program is run *without* the optional flag `-d`, the program must display only the total number of matching tuples and the time that it took to join the files to standard out, as follows:

```
Number of tuples: X
Time: Y
```

If the program is run *with* the optional flag `-d`, then instead of showing the information above, all matching tuples must be output to standard out, printing one matching tuple per line, for example:

```
tuple1
tuple2
...
```

Allowing for input and output buffers, your algorithm should allocate remaining buffers (if any) to read in a *block* of data from the *first* file specified when your program is invoked. (Note: here, block is used in the context of the block nested loops join algorithm, not a disk block.)

This block of data must be hashed into an in-memory hash table for fast lookups. It is up to you to design a hash table, and to use a hash function that is appropriate for hashing the type of data that you are dealing with in the supplied files. You will be asked to explain and justify your choice of hash function in your report. Your hashing code must be entirely your work.

1.2 Hash Join (35%)

Implement a hash join that takes as input the total number of available buffers, and the names of the two files to join. The executable must be called `HashJoin` and must be run as follows:

```
./HashJoin [-d] <buffers> <path>/characters <path>/guilds
```

As output, when the program is run *without* the optional flag `-d`, the program must display only the total number of matching tuples and the time that it took to join the files to standard out, as follows:

```
Number of tuples: X
Time: Y
```

If the program is run *with* the optional flag `-d`, then instead of showing the information above, all matching tuples must be output to standard out, printing one matching tuple per line, for example:

```
tuple1
tuple2
...
```

Your hash join algorithm needs to consist of a distinct *partitioning* and *matching* phase. You must clearly indicate these phases, using appropriate comments in your source code. For this question, you need to implement two hash functions: one for the partitioning and one for the matching phase. The hash functions are expected to work well for hashing integers.

For the matching phase, you will need to make use of an in-memory hash table. It is up to you to design a hash table; you may re-use your hash table code from Section 1.1. Your hashing code must be entirely your work. Marks will be awarded for good implementations, and for suitable choices of hash functions.

Experiments and Report (30%)

Create a file called `report.txt`. In this file, record your answers to the following questions.

Nested Loops Join

Create a heading called "1: Nested Loops Join" in your report file. Using your `NestedLoopJoin` program, carry out the following exercises:

1. Average the time to join the `characters` and `guilds` files over twenty runs, with each of the following number of available buffers: 100, 200, 500, 1000. In this experiment, the `characters` file must be the second parameter for your program, and the `guilds` file must be the third parameter.

   Report the average times with each different number of buffers. Include the date and time of each run.

2. Average the time to join the `characters` and `guilds` files over twenty runs with each of the following number of available buffers: 100, 200, 500, 1000. In this experiment, the `guilds` file must be the second parameter, and the `characters` file must be the third parameter.

   Report the average times with each different number of buffers. Include the date and time of each run.

3. Discuss your results. Is the data as you expected? What number of buffers would you recommend for the join process? Which order of file parameters works best?

   In answering these questions, you should demonstrate understanding of the theoretical properties of the nested loop join. Restrict your answer to a maximum of half a page.

4. In up to ten sentences, explain how you implemented your hash table. As part of your discussion, describe your hash function, and explain why it is suitable for hashing this type of data.

Hash Join

Create a heading called "2: Hash Join" in your report file. Using your `HashJoin` program, carry out the following exercises:

1. Average the time to join the `characters` and `guilds` files over twenty runs, with each of the following number of buffers: 100, 200, 500, 1000. In this experiment, the `characters` file must be the second parameter for your program, and the `guilds` file must be the third parameter.

   Report the average times with each different number of buffers. Include the date and time of each run.

2. Average the time to join the `characters` and `guilds` files over twenty runs with each of the following number of buffers: 100, 200, 500, 1000. In this experiment, the `guilds` file must be the second parameter, and the `characters` file must be the third parameter.

   Report the average times with each different number of buffers. Include the date and time of each run.

3. Discuss your results. Is the data as you expected? What number of buffers would you recommend for the join process? Which order of file parameters works best?

   In answering these questions, you should demonstrate understanding of the theoretical properties of the hash join. Restrict your answer to a maximum of half a page.

4. In up to ten sentences, describe your two hash functions, and explain your choice. Remember to cite any sources that you use, if appropriate.

Discussion Questions

Create a heading called "3: Discussion" in your report file. You may answer the following questions using a list of bullet-points or as paragraph-style answers.

1. In a few sentences, summarise your overall results. Which algorithm works best for joining the files?

2. Would the algorithm that you found to work best on the supplied data files work as well for very large files? If not, which algorithm would you expect to work best and why?

3. Would the relative performance of your algorithms be the same when joining other small data files of different relative sizes? Why? In your answer, give examples of cases where performance is likely to be the same or different. You could run additional experiments to verify your findings.

What to Submit, When, and How

The assignment is due on Friday, 22 May 2009, 5:30pm.

You should submit your source code, reports, script and Makefile using `turnin`. (Do not submit executables, object files, or data.)

Use the following process:

1. Put all your submission files in a directory
2. Run `turnin` using (*for example*):
   `turnin -c ds -p Assignment2 *.h *.c Makefile *.txt`
3. You can check your submission using:
   `turnin -v -c ds -p Assignment2`
   More information about `turnin` can be found by typing `man turnin` on `yallara`.

Your turnin submission must have a timestamp showing Friday, 22 May 2009, 5:30pm (or earlier). Late submissions should be submitted using the same turnin procedure, but will be penalised according to the course guide.