Algorithms in Secondary Memory

Project Assignment

2020-2021

Assignment

The aim in this assignment is to get real-world experience with the performance of external-memory algorithms. Concretely, you are asked to:

1. Compare the performance of different methods for reading from and writing to secondary memory.
2. Implement an external-memory merge-sort algorithm (such as the one described in Section 15.4.1 of TCB), and examine its performance under different parameters.

Follow the outline of tasks below. You need to document your findings in a report that needs to be handed in together with your implementation. Code may be written in C++, Java, or Python.

You will be asked to evaluate the performance of the programs that you write on the real-world dataset available from

`https://wit.ulb.ac.be/nextcloud/index.php/s/R7Jef6switwL32M`

This dataset, called the *IMDB dataset* in what follows, contains a snapshot of data harvested from the IMDB movie database (`https://www.imdb.com`), which contains information related to films, television programs, etc. The dataset is a gzip-compressed tar file (1.3 GB compressed, 3.7GB uncompressed) containing CSV files. You can think of each CSV file as corresponding to a relational table. The corresponding relational schema is expressed in appendix A, and is also included in the file `schematext.sql` in the dataset.

Note that all files in the dataset are *text files*. For simplicity, we will focus on reading, writing, and sorting text files in this assignment. In contrast, real data management systems typically organize data in *binary files* because this is more efficient in general (i.e., it takes less space). The underlying ideas remain the same, however.

1 Project outline

The project consists of the following tasks.

1.1 Reading and writing

Your external-memory merge sort implementation will need to be able to sort files. It will hence need to read data from, and write data to disk. A first step, therefore, is to develop *stream* classes that can read and write text files. You will need two types of streams: *input streams* and *output streams*. The input stream should support the following operations:

- `open`: open an existing file for reading
- `readln`: read the next line from the stream,
- `seek(pos)`: move the file cursor to `pos` so that a subsequent `readln` reads from position `pos` to the next end of line,
- `end_of_stream`: a boolean operation that returns true if the end of stream has been reached.

The output stream should support the following operations:

- `create`: create a new file,
- `writeln`: write a string to the stream and terminate this stream with the newline character, and
- `close`: close the stream.

Use the following four distinct I/O mechanisms to obtain four different implementations of the input and output streams' `readln` and `writeln` methods.

1. Implement `readln` and `writeln` by reading / writing one *character* at a time using the `read` and `write` system calls. Here, you continue reading characters (and collecting the characters read so far) until you read the end-of-line symbol. The `read` and `write` calls are available in C/C++ through the `io.h` header (if you're programming on windows in Visual C++) or through the `unistd.h` header (on Unix/Linux). If you are doing the project in Java these system calls can be mimicked. You can mimic `read` by calling `read()` on a `java.io.FileReader`, as illustrated by the following code snippet:

   ```
   FileReader is = new FileReader( new File("input.data" ) );
   is.read(); //read a single character
   ```

   The `write` system call can be mimicked similarly.

   This behavior can be mimicked in python by setting the `buffering` parameter of the `open` call to 0 when opening the file.

2. Implement `readln` and `writeln` by using the `fgets` and `fputs` functions from the C `stdio` library. These functions implements a buffering mechanism. If you are doing the project in Java these functions can be mimicked. You can mimic `fgets` by calling `readLine()` on a `java.io.BufferedReader` that itself is wrapped around a `java.io.FileReader`, as illustrated by the following code snippet:

   ```
   FileReader is = new FileReader( new File("input.data" ) );
   BufferedReader br = new BufferedReader( is );
   br.readLine();
   ```

   The `fputs` function can be mimicked similarly. In python, `fgets` can be mimicked by opening the file without specifying a value for the `buffering` parameter, and then iterating over the file, as in the following code snippet.

```
file = open("input.data")
for line in file:
   # do stuff with the line
```

3. Read and write are implemented as in step (1), except that now you equip your streams with a buffer of size B in internal memory. Whenever the buffer becomes empty/full the next B characters are read/written from/to the file.

4. Read and write is performed by mapping and unmapping B characters of the file into internal memory through *memory mapping*. Whenever you need to read/write outside of the mapped portion, the next B element portion of the file is mapped.

   Memory mapping is available in C/C++ through the mmap/munmap functions of the `<sys/mman.h>` header (in Linux) and the `CreateFileMapping` function of `<windows.h>` (in Windows). Alternatively, the `Boost` C++ library[1] provides platform-independent interfaces to memory-mapped files.[2] In Java, you should use the `map` method of the `java.nio.channels.FileChannel` class. In python, use the `mmap` module.

   For this particular implementation you are required to do a little research on what *memory mapping* actually is; you are expected to explain this concept in your report.

1.2 Experiment 1.1: sequential reading

Conduct the following experiment to determine which stream implementations perform best for *sequential* reading. Write a program called *length* that takes as input the name f of one of the csv files in the IMDB dataset. The program initializes a counter *sum* to zero and then reads f line by line. For each line, it adds the length of the line to the counter. When it has read the entire file, the value of *sum* is output.[3]

Create different versions of *length*, one version for each input stream implementation. The goal is to determine which version works the best, and to clearly document the process of this discovery in the report. What are the limitations and advantages of each implementation? In particular:

- Try the different versions on files of different sizes in the IMDB dataset.
- For versions that include implementations (3) and (4), experiment with different values of B (including very large ones). Identify the optimal values.

It may not be feasible to run some versions on large files (because it takes excessive time to do so). Therefore, start with running on files of small size. Be sure to document which versions of distribute you did not try.

Before you start running the actual experiments, reflect on the possible performance of each of the aforementioned implementations. What do you expect? Define, for each of the 4 implementations, a cost formula that estimates the total number of I/Os that needs to be

[1] `http://www.boost.org/`

[2] It actually provides 2 such interfaces, one in the `iostreams` sublibrary, and one in the `interprocess` sublibrary.

[3] It is important that your program does output *sum* as written, especially if you are doing the project in Java or Python. Otherwise the compiler/interpreter may actually infer that you are reading the file without actually doing anything with it, and subsequently optimize your program by not reading the file at all.

done, in function of N, the size of f. Do your experimental results conform to this theoretical model?

Don't forget to describe, in your report, the environment on which your experiments have been performed (machine type, hard disk type, operating system, total memory available, programming language used, and libraries used).

1.3 Experiment 1.2: random reading

After conducting experiment 1.1. you will have a good idea of which stream implementation(s) are best for reading sequentially. The goal in the next experiment is to see if the situation changes when we do *random reading* instead.

Write a program called *randjump* that takes as input two parameters: f and j, where f is the name of one of the csv files in the IMDB dataset, and j is a positive integer. The program *randjump* opens *filename*, initializes a counter *sum* to 0, and repeats the following j times:

- compute a random number p between 0 and the length of the input file;
- `seek` to position p;
- add l to *sum* where l is the the length of the string returned by call `readline`.

At the end of the iteration, it outputs *sum*. In other words, *randjump* makes j random jumps in the file. For each jump, it computes the length until the next end-of-line symbol. It outputs the sum of all such lengths.

Create different versions of *randjump*, one version for each possible input stream implementation. The goal is to determine which version works the best, and to clearly document the process of this discovery in the report. What are the limitations and advantages of each implementation? In particular,

- Try the versions on files of different sizes in the IMDB dataset.
- Try the versions on different values of j, from small to large ones.
- For versions that include implementations (3) and (4), experiment with different values of B (including very large ones). Identify the optimal values.

Again *before* you start running the actual experiments, write down what you expect to happen? Do you expect a difference with experiment 1.1 ? What do you observe ? Can you explain why ?

1.4 Experiment 1.3: combined reading and writing

Write a program called *rrmerge* that takes as input a variable number of parameters $f_1, \ldots, f_k$ where each f_i is the name of one of the csv files in the IMDB dataset. The program *rrmerge* creates one new file that merges the contents of $f_1, \ldots, f_k$. It does so by copying one line of each f_i in a round robin fashion, until all files have been processed.

Create different versions of *rrmerge* for each pair (x, y) of input stream implementation x and output stream implementation y. You do not need to consider all possible input stream implementations, however: for x it suffices to consider only the best implementation as identified in experiment 1.1, and the best implementation as identified in experiment 1.2. You should consider all possible implementations for y.

The goal is to determine which version works the best, and to clearly document the process of this discovery in the report. What are the limitations and advantages of each implementation? In particular:

- Try with different values the number of input files k.
- Try with different sizes of the input files.
- For versions that include implementations (3) and (4), experiment with different values of B for the output stream implementation (including very large ones). Identify the optimal values.

It may not be feasible to run some versions on large files and/or large values of k (because it takes excessive time to do so). Therefore, start with running on files of small size and small values of k. Be sure to document which versions of distribute you did not try.

Again *before* you start running the actual experiments, write down what you expect to happen? Which pair (x, y) do you assume to perform best? Why ? Is your expectation confirmed by the experiments ? Why (not)?

1.5 Multi-way merge

Finally, implement a program *extsort* that takes as input a file f from the IMDB dataset, a positive integer k, a positive integer M, and a positive integer d. The program *extsort* will create a new file that contains the same contents as f, but now sorted on the k-th column of f. (It is required that k is at most the number of columns in f). This sorting happens according to a variant of external-memory merge-sort algorithm (as the one described in Section 15.4.1 of TCB):

1. Read the (single) input file line by line and store the lines read in an in-memory buffer until the size (in bytes) of all buffered lines exceeds M. Each line corresponds to a record. Sort the buffered records on the k-th column using an internal memory sorting algorithm. (You can re-use the sort algorithm(s) that comes with your programming language.) Write the sorted records to a new output file.

   Repeat the procedure above until you have approximately $\lceil N/M \rceil$ sorted files, each of size approximately M. Collectively, these $\lceil N/M \rceil$ file contain the same contents as f.

   Note that you need to take care that you have only (approximately) M bytes of f in your memory at any given time.

2. Store the references to the $\lceil N/M \rceil$ files in a queue. Again, take care to only store the references to the files/streams in the queue. The content of the streams themselves should not be loaded into memory until the stream is treated by the following step.

3. Repeatedly, until only one stream remains, merge the d first streams in the queue, and put the resulting stream at the end of the

   queue (if less than d streams remain in the queue, merge those).

Here, the *merge* procedure is a procedure that is given d (references to) input streams (all sorted on column k) and creates a single output stream all containing the elements from the input stream in sorted order (on column k). The merging should use a priority queue

(e.g., a heap) to obtain the next element to be output at all times. In particular, apart from the buffering that is done by your input/output stream implementations, you should have at most $O(d)$ lines in memory (one line for each input stream).

Experiment with your merge-sort implementation using the best combination for reading and writing identified in Experiment 1.3. Experiment with CSV files of different sizes, different versions of M, and of d. Identify optimal values for M and d. Does it make a difference whether the input file is already sorted on k or not ?

Again *before* you start running the actual experiments, write down what you expect to happen. In particular, devise a cost formula that estimates the number of I/Os in function of N (the size of f in bytes), M, and d. What do you expect that will happen given this cost formula ? Do you experimental results confirm this ? Why (not)?

2 Tips

- Run your programs a suitable number of times for each combination of input parameters and use the average running time. This should level out fluctuations due to other processes.
- There are quite a number of Unix commands to time the execution of the program. Among these are `time` and `timex`. In some shells (e.g. csh), there are also built-in versions of time. There are also corresponding unix routines called `times`, `getrusage`, and `gettimeofday`. Their availability may differ from machine to machine. The Windows API also provides `QueryPerformanceCounter` and `QueryFrequencyCounter` that can also be used to time the execution. Both the Boost library and the C++11 STL also provides a `chrono` class that provides timing in a platform independent way.
- Alternatively there are a number of *benchmmarking* frameworks available for both C++ and Java that allow you to specify the code that needs to be benchmarked, and that takes care of running this benchmark a number of times and reporting running time averages, standard deviations, etc.
- When comparing different implementations, make sure to run them on the *same input parameters* in order to get meaningful results. For experiment 1.2 (random reading) you may want to initialize your random number generator with the same seed to compare different implementations, in order to ensure that they do the exact same work.

3 Report

Your report should document the progress of your project, your findings, as well as try to relate your experimental observations to what we have discussed during the theory lectures. (In particular: the cost formulas from Lecture 6.) Here is a suggested table of contents.

1. Introduction and Environment.

   Briefly summarize the project goals. Document the machine environment on which the experiments were run (mentioning disk type, amount of memory, etc); the programming language used; and the external libraries used (if any).

2. Observations on reading and writing

(a) Your implementation

For each of the 4 implementations, briefly explain the implementation in your own words. Give details on how you programmed the implementation if necessary.

(b) Experimental observations regarding experiment 1.1.

Discuss the expected behavior, cost formulas, and what you observe during your experiments. Use graphs and plots to compare the implementations. Explain what you observe.

(c) Experimental observations regarding experiment 1.2.

Discuss the expected behavior, and what you observe during your experiments, answering the questions posed in experiment 1.2. Use graphs and plots to compare the implementations. Explain what you observe.

(d) Experimental observations regarding experiment 1.3.

Discuss the expected behavior, and what you observe during your experiments, answering the questions posed in experiment 1.3. Use graphs and plots to compare the implementations. Explain what you observe.

3. Observations on multi-way merge sort.

(a) Expected behavior.

Explain the implementation in your own words. Give details on how you programmed the implementation if necessary. Explain what kind of performance behavior you expect (before running any experiment) from the implementation. Hereto, define a cost function that estimates the total number of I/Os that need to be done, in function of N, d, and M

(b) Experimental observations.

Discuss what you observe during your experiments. Use graphs and plots to compare the implementations. In each experiment, vary only one parameter (e.g., N) while keeping the other parameters (e.g., M,d) fixed.

(c) Discussion of expected behavior vs experimental observations. Identify what good choices are of the parameters for varying input sizes.

4. Overall conclusion.

Summarize what you have learned in this project.

Careful! Do not simply copy-and-paste text from the internet to include in your report. If you copy-and-paste existing text/graphics without duly citing the source from which this was copied this constitutes fraud, which will result on 0/20 on the course and possibly an exclusion from other exams!

4 Modalities

The assignment has the following modalities:

1. This project assignment contributes 10/20 to the overall grade; the written exam contributes the remaining 10/20 points.

2. The assignment will be graded on (1) the implementation itself and (2) the report that you need to write describing both the implementation, the expected behavior, and the experimental work.

3. The assignment should be solved in groups of 3 (if the total number of students in the course is not divisible by 3, at most two groups of 2 students will be allowed). You are asked to register your group members through the Virtual University, by October 30 at the latest. If you cannot find a partner, please indicate so by sending an email to prof. Vansummeren (`svsummer@ulb.ac.be`), who will hook you up with a partner.

4. This assignment is mandatory. If you do not make the assignment, you cannot pass the course in the first exam session.

5. You are asked to submit your solution (report in PDF format + code) to the activity "Submission algorithms in secondary memory project" on the *UV* **no later than 3 January 2021**. It is strongly recommended, however, to finish & submit your project before the Christmas break (i.e., well before this deadline).

6. As stated above, the project will need to use a *priority queue* and use a main-memory sorting algorithm. You are only allowed to use existing code and existing libraries for these two aspects; all other code must be written by you for the project, unless explicitly stated otherwise in the assignment above. For example, you can use the `std::priority_queue` class from the C++ STL to implement priority queue (if you do the project in C++), or the `Java java.util.PriorityQueue` (if you do it in `Java`).

   (Note that, if you decide to use a benchmarking framework, you are free to use that as a library in your code.)

7. Sharing of code or reports between groups is not allowed. (Groups may, however, verbally discuss ideas on how to tackle the project).

8. Plagiarism, in the sense of copy-pasting from existing reports or copy-pasting existing external memory merge-sort code is not allowed. In case plagiarism is detected, students risk being punished according to article 34 of the exam regulations shown in Figure 1.

9. As stated above, this project assignment is done in groups of 3. All members of the group receive the same grade on the project! Therefore, in case that a group member feels that another group member does not do his/her share of the agreed work, please contact prof. Vansummeren immediately.

A Relational schema of the dataset

```
CREATE TABLE aka_name (
    id integer NOT NULL PRIMARY KEY,
    person_id integer NOT NULL,
    name character varying,
    imdb_index character varying(3),
    name_pcode_cf character varying(11),
    name_pcode_nf character varying(11),
    surname_pcode character varying(11),
```

Art.34 In case of fraud or plagiarism during an examination or during a test at an interim date during the academic year, or in relation with the preparation of written reports or papers, the course professor reports the case in writing prior to the jury deliberation to the relevant academic authority levels for disciplinary matters. A copy of that fraud report is addressed to the jury chairmen. The student can ask to be heard by a jury chairperson prior to the jury deliberation, in presence of the related course professor. Without prejudice to the disciplinary processes at the University Faculty level, in case of fraud the student points for the related course are brought down to 0/20. The jury further can:

- *decide to cancel the examination session;*
- *decide to refuse the student access to both examination sessions of that academic year.*

Figure 1: Excerpt of the Exam Regulations concerning fraud.

```
    md5sum character varying(65)
);

CREATE TABLE aka_title (
    id integer NOT NULL PRIMARY KEY,
    movie_id integer NOT NULL,
    title character varying,
    imdb_index character varying(4),
    kind_id integer NOT NULL,
    production_year integer,
    phonetic_code character varying(5),
    episode_of_id integer,
    season_nr integer,
    episode_nr integer,
    note character varying(72),
    md5sum character varying(32)
);

CREATE TABLE cast_info (
    id integer NOT NULL PRIMARY KEY,
    person_id integer NOT NULL,
    movie_id integer NOT NULL,
    person_role_id integer,
    note character varying,
    nr_order integer,
    role_id integer NOT NULL
);

CREATE TABLE char_name (
    id integer NOT NULL PRIMARY KEY,
    name character varying NOT NULL,
```

```
    imdb_index character varying(2),
    imdb_id integer,
    name_pcode_nf character varying(5),
    surname_pcode character varying(5),
    md5sum character varying(32)
);

CREATE TABLE comp_cast_type (
    id integer NOT NULL PRIMARY KEY,
    kind character varying(32) NOT NULL
);

CREATE TABLE company_name (
    id integer NOT NULL PRIMARY KEY,
    name character varying NOT NULL,
    country_code character varying(6),
    imdb_id integer,
    name_pcode_nf character varying(5),
    name_pcode_sf character varying(5),
    md5sum character varying(32)
);

CREATE TABLE company_type (
    id integer NOT NULL PRIMARY KEY,
    kind character varying(32)
);

CREATE TABLE complete_cast (
    id integer NOT NULL PRIMARY KEY,
    movie_id integer,
    subject_id integer NOT NULL,
    status_id integer NOT NULL
);

CREATE TABLE info_type (
    id integer NOT NULL PRIMARY KEY,
    info character varying(32) NOT NULL
);

CREATE TABLE keyword (
    id integer NOT NULL PRIMARY KEY,
    keyword character varying NOT NULL,
    phonetic_code character varying(5)
);

CREATE TABLE kind_type (
    id integer NOT NULL PRIMARY KEY,
```

```
    kind character varying(15)
);

CREATE TABLE link_type (
    id integer NOT NULL PRIMARY KEY,
    link character varying(32) NOT NULL
);

CREATE TABLE movie_companies (
    id integer NOT NULL PRIMARY KEY,
    movie_id integer NOT NULL,
    company_id integer NOT NULL,
    company_type_id integer NOT NULL,
    note character varying
);

CREATE TABLE movie_info_idx (
    id integer NOT NULL PRIMARY KEY,
    movie_id integer NOT NULL,
    info_type_id integer NOT NULL,
    info character varying NOT NULL,
    note character varying(1)
);

CREATE TABLE movie_keyword (
    id integer NOT NULL PRIMARY KEY,
    movie_id integer NOT NULL,
    keyword_id integer NOT NULL
);

CREATE TABLE movie_link (
    id integer NOT NULL PRIMARY KEY,
    movie_id integer NOT NULL,
    linked_movie_id integer NOT NULL,
    link_type_id integer NOT NULL
);

CREATE TABLE name (
    id integer NOT NULL PRIMARY KEY,
    name character varying NOT NULL,
    imdb_index character varying(9),
    imdb_id integer,
    gender character varying(1),
    name_pcode_cf character varying(5),
    name_pcode_nf character varying(5),
    surname_pcode character varying(5),
    md5sum character varying(32)
```

```
);

CREATE TABLE role_type (
    id integer NOT NULL PRIMARY KEY,
    role character varying(32) NOT NULL
);

CREATE TABLE title (
    id integer NOT NULL PRIMARY KEY,
    title character varying NOT NULL,
    imdb_index character varying(5),
    kind_id integer NOT NULL,
    production_year integer,
    imdb_id integer,
    phonetic_code character varying(5),
    episode_of_id integer,
    season_nr integer,
    episode_nr integer,
    series_years character varying(49),
    md5sum character varying(32)
);

CREATE TABLE movie_info (
    id integer NOT NULL PRIMARY KEY,
    movie_id integer NOT NULL,
    info_type_id integer NOT NULL,
    info character varying NOT NULL,
    note character varying
);

CREATE TABLE person_info (
    id integer NOT NULL PRIMARY KEY,
    person_id integer NOT NULL,
    info_type_id integer NOT NULL,
    info character varying NOT NULL,
    note character varying
);
```