CS302 --- External Sorting

Review

External Sorting--This term is used to refer to sorting methods that are employed when the data to be sorted is too large to fit in primary memory.

Characteristics of External Sorting

1. During the sort, some of the data must be stored externally. Typically the data will be stored on tape or disk.
2. The cost of accessing data is significantly greater than either bookkeeping or comparison costs.
3. There may be severe restrictions on access. For example, if tape is used, items must be accessed sequentially.

Criteria for Developing an External Sorting Algorithm

1. Minimize number of times an item is accessed.
2. Access items in sequential order

Sort Merge

Sort merge is the strategy of choice for external sorting because it:

1. Accesses records sequentially
2. Minimizes block accesses
3. Gives a stable sort

Sort Merge Strategy

1. Divide the file into runs such that the size of a run is small enough to fit into main memory
2. Sort each run in main memory using a fast in-memory sorting algorithm
3. Merge the resulting runs together into successively bigger runs, until the file is sorted.

Balanced Multiway Merging

Strategy

1. Select an equal number of I/O units (e.g., tapes, disks)
2. Sort Step

- Select half of the I/O units to be output files
- Using an in-memory sorting algorithm, create the initial runs and divide them evenly among the output files

3. Merge Step
   - **(a)** On each merge step, alternate the I/O units so that on one step, a unit is an input file and on the next step it is an output file
   - **(b)**Read one run from each of the input files, merge the runs, and store the resulting run on an output file. Alternate output files so that the runs are evenly distributed on the output files.
   - **(c)**Repeat step **b** until all the runs on the input files have been merged.
   - **(d)**Repeat steps **a-c** until the file has been sorted

Example

Sort the keys `A S O R T I N G A N D M E R G I N G E X A M P L E` assuming that initial run sizes are 3 and that 6 I/O units are used.

Pseudo-Code

The pseudo-code in this section assumes that the user provides an input file and an output file. It also assumes that two parameters for the balanced multi-way algorithm are determined by the user via switches:

1. initial run-size: The size, in number of records, of the initial runs

2. num-ways: The parameter P. In other words, P-way merging will be used.

The Main Procedure

```
main(argc, argv) {

    assign default values for run_size and num_ways
    parse the switches and, if appropriate, assign the user-defined
         values to run-size and num_ways
    open the output file
    create two arrays of scratch files
    read the input file, create the initial runs, and assign the runs
        to the scratch output files (create_initial_runs)
    sort the runs using the sort_merge algorithm (sort_merge)
    close the output file

}
```

Creating the Initial Runs

```
create_initial_runs(input_file_name, run_size, num_ways) {

    allocate a dynamic array, a, large enough to accommodate runs of
        size run_size
    open the input file
    for i = 0 to NUM_WAYS-1 {
        open output_scratch_file i
    }
    more_input = true
    next_output_file = 0
    num_runs_per_output_file = 0
    while (more_input) {
        for i = 1 to run_size {   /* entry 0 in the array is reserved for
                                     a possible sentinel value. If your
```

```
                                    sorting algorithm does not require a
                                    sentinel value, the for loop could
                                    start at i = 0 and go to
                                    (run_size - 1) */
            if (not end_of_input_file)
                read a record into a[i]
            else {
                more_input = false
                break
            }
        }
        sort array a using an in-memory algorithm like quicksort

        /* write the records to the appropriate scratch output file
        for j = 1 to i {  /* can't assume that the loop runs to run_size
                          since the last run's length may be less than
                          run_size */
            write a[i] to scratch_output_file[next_output_file]
        }
        output the sentinel value to scratch_output_file[next_output_file]

        /* everytime we get back to the first output file, increment the
            number of runs per output file by 1 */
        if (next_output_file == 0)
            num_runs_per_output_file = num_runs_per_output_file + 1
        next_output_file = (next_output_file + 1) % num_ways
    }
    /* make sure the same number of runs are assigned to each scratch
       output file */
    if (next_output_file != 0) {
        for i = next_output_file to (num_ways - 1) {
            output the sentinel value to scratch_output_file[i]
        }
    }
    for i = 0 to (num_ways -1)
        close scratch_output_file[i]
    close the input file

    return num_runs_per_output_file
}
```

The Sort-Merge Procedure

```
sortmerge(output_file, num_runs_per_scratch_output_file, num_ways) {

  for (N = num_runs_per_scratch_output_file; N > 1;
        N = ceiling(N / num_ways) /* ceiling is a function that rounds up
                                    to the nearest integer. You need to
                                    write this function yourself */
  {
    open_scratch_files()  /* open input and output scratch files */
    for i = 0 to (N-1) {
      create_run(output_scratch_files[i % num_ways], true, num_ways);
    }
    /* make sure the same number of runs are assigned to each scratch
       output file */
    if ((i % num_ways) != 0) {
        for j = (i % num_ways) to (num_ways - 1) {
            output the sentinel value to scratch_output_file[j]
        }
    }
    close_scratch_files();  /* close input and output scratch files */
  }
  /* make the last run write into the output file */
```

```
  open_scratch_files();
  create_run(output_file, false, num_ways);
  close_scratch_files();
}

create_run(output_file, generate_sentinel_value_flag, num_ways) {

  /* initialize the merge_array */
  for i = 0 to (num_ways -1) {
    read a record from input_scratch_file i to merge_array[i]

  /* create the run */
  while (true) {
    find the minimum key and the index of that minimum key (min_index)
         in merge_array
    if (min == SENTINEL_VALUE)  /* the run is complete if the sentinel value
      break;                          is reached */
    write the record with the minimum key in merge_array to output_file
    read the next record from the input_scratch_file with index min_index
        into merge_array[min_index]
  }
  if (generate_sentinel_value_flag == true)
    write the sentinal value to the output file
}
```

Opening Scratch Files

If the sort is a `num_ways` sort, then your program will need `num_ways` scratch input files and `num_ways` scratch output files. Your program will need to alternate between using a scratch file as an input file and as an output file. On one run it will be an input file, then on the next run an output file, then an input file, etc.

The easiest way to handle this alternation is to maintain a flag that keeps track of which of your two file arrays is currently the input array. For example, you might declare a flag called `input1`, which if true indicates that your first array is the current input array and if false indicates that your second array is the current input array.

As an example of the use of this flag, here is the code for `open_scratch_files`:

```
void open_scratch_files () {
  // if input1 is true, open the first bank of files for input and
  // the second bank for output
  if (input1) {
    for (i = 0; i < num_ways; i++) {
      filearray1[i].Open('i');
      filearray2[i].Open('o');
    }
  }
  // if input1 is false, open the first bank of files for output and
  // the second bank for input
  else {
    for (i = 0; i < num_ways; i++) {
      filearray1[i].Open('o');
      filearray2[i].Open('i');
    }
  }
}
```

Performance

1. If M records can be sorted in-memory and the file consists of N records, then the number of initial runs is N / M.

2. If there are 2P I/O units available, then the number of subsequent passes is ceiling($\log_P(N / M)$) since each pass reduces the number of runs by P. Here ceiling(x) means the smallest integer greater than or equal to x.