CS W186
Fall 2019

Sorting

In CS61B, you learned about many different sorting algorithms. Why are we learning yet another new one in this class? All of the traditional sorting algorithms (i.e. quick sort, insertion sort, etc.) rely on us being able to store all of the data in memory. This is a luxury we do not have when developing a database. In fact, most of the time our data will be an order of magnitude larger than the memory available to us.

1 I/O Review

Remember that an I/O is any time we either write a page from memory to disk or read a page from disk into memory. Because of how time consuming it is to go to disk, we only look at the number of I/Os an algorithm incurs when analyzing its performance. We pretty much ignore traditional measures of algorithmic complexity like big-O. Therefore, when developing our sorting algorithm we will attempt to minimize the number of I/Os it will incur. One last thing to note when counting IOs is that we ignore any potential caching done by the buffer manager. This implies that once we unpin the page and say that we are done using it, the next time we attempt to access the page it will always cost 1 IO.

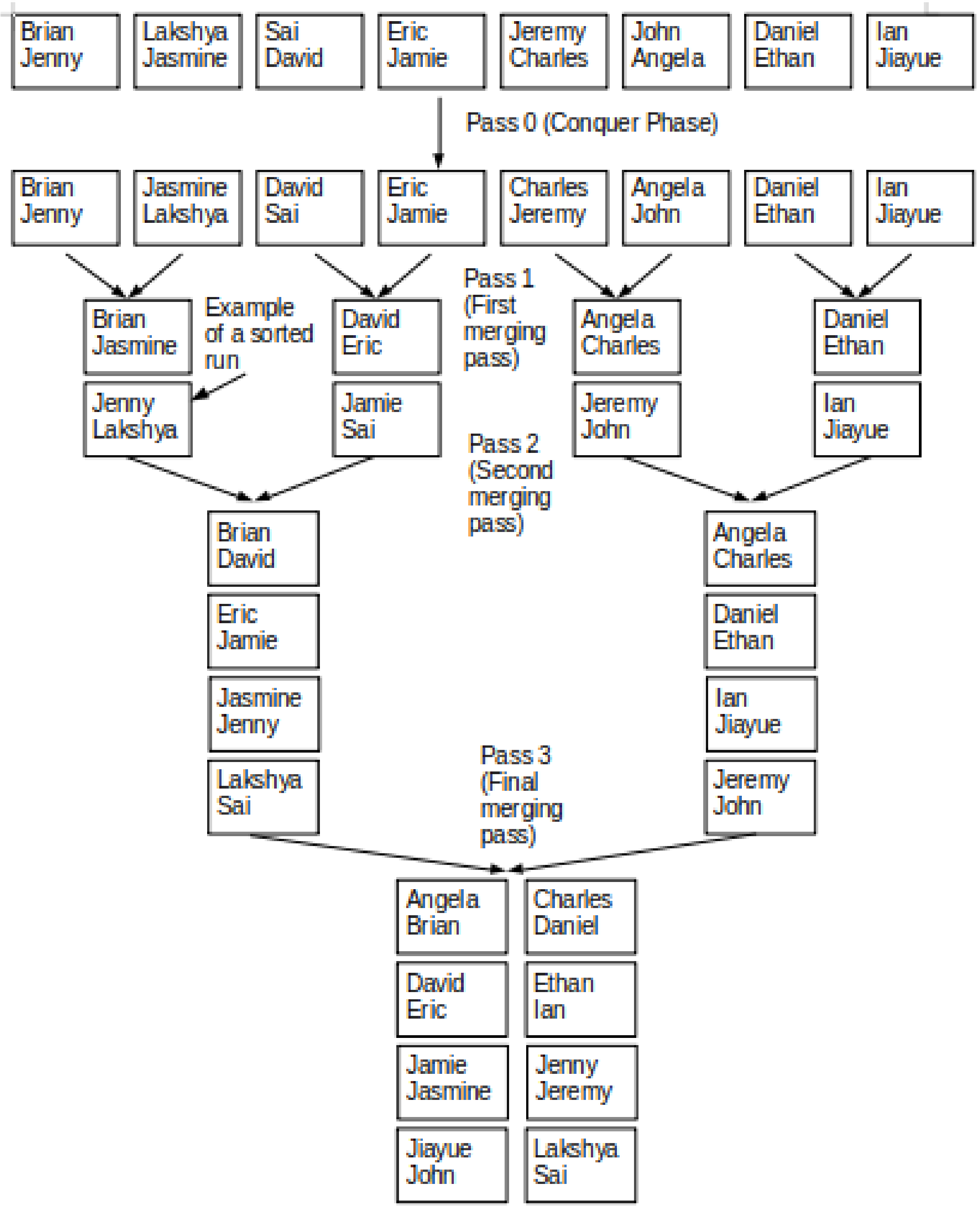
2 Two Way External Merge Sort

Let's start by developing a sorting algorithm that works but is not as good as possible. Because we cannot keep all of our data in memory at one time, we know that we are going to sort different pieces of it separately and then merge it together.

In order to merge two lists together efficiently, they must be sorted first. This is a hint that the first step of our sorting algorithm should be to sort the records on each individual page. We'll call this first phase the "conquer" phase because we are conquering individual pages.

After this, let's start merging the pages together using the merge algorithm from merge sort. We'll call the result of these merges **sorted runs**. A sorted run is any sequence of pages that is sorted.

The rest of the algorithm will simply be to continue merging these sorted runs until we have only one sorted run remaining. One sorted run implies that our data is fully sorted! See the image on the next page for a diagram of the algorithm run to completion.


Brian Jenny
Lakshya Jasmine
Sai David
Eric Jamie
Jeremy Charles
John Angela
Daniel Ethan
Ian Jiayue
Pass 0 (Conquer Phase)
Brian Jenny
Jasmine Lakshya
David Sai
Eric Jamie
Charles Jeremy
Angela John
Daniel Ethan
Ian Jiayue
Pass 1 (First merging pass)
Brian Jasmine
Jenny Lakshya
Example of a sorted run
David Eric
Jamie Sai
Angela Charles
Jeremy John
Daniel Ethan
Ian Jiayue
Pass 2 (Second merging pass)
Brian David
Eric Jamie
Jasmine Jenny
Lakshya Sai
Angela Charles
Daniel Ethan
Ian Jiayue
Jeremy John
Pass 3 (Final merging pass)
Angela Brian
David Eric
Jamie Jasmine
Jiayue John
Charles Daniel
Ethan Ian
Jenny Jeremy
Lakshya Sai

3 Analysis of Two Way Merge

When analyzing a database algorithm, the most important metric is the number of I/Os the algorithm takes, so let's start there. First, notice that each pass over the data will take $2 * N$ I/Os where N is the number of data pages. This is because for each pass, we need to read in every page, and write back every page after modifying it.

The only thing left to do is to figure out how many passes we need to sort the table. We always need to do that initial "conquer" pass, so we always have at least one. Now, how many merging passes are required? Each pass, we cut the amount of sorted runs we have left in half. Dividing the data each time should scream out *log* to you, and because we're diving it by 2, the base of our log will be 2. Therefore we need $\lceil \log_2(N) \rceil$ merging passes, and $1 + \lceil \log_2(N) \rceil$ passes in total. This leads to our final formula of $2N * (1 + \lceil \log_2(N) \rceil)$ I/Os.

Now let's analyze how many buffer pages we need to execute this algorithm. Remember that a **buffer page**, or **buffer frame**, is a slot for a page to be stored in memory.

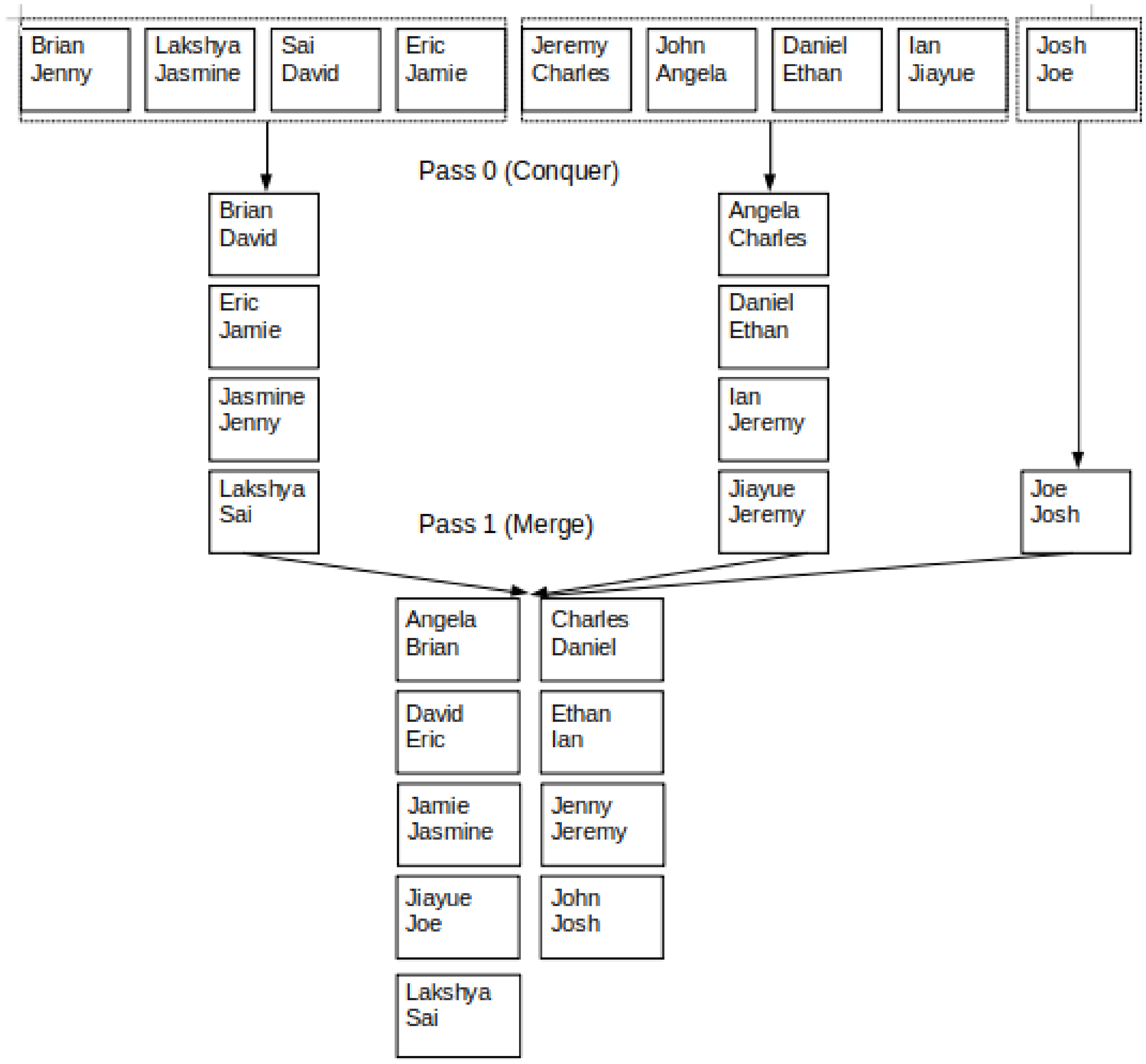
The first pass, the "conquering pass", sorts each page individually. This means we only ever need 1 buffer page for this pass, to hold the page that we are sorting!

Now let's analyze the merging passes. Recall how merging works in merge sort. We only compare the first value for the two lists that we are merging. This means that we only need to store the first page of each sorted run in memory, rather than the entire sorted runs. When we have used all of the records from the original page, we simply get rid of that page from memory and load in the next page of the sorted run. So far, we need 2 buffer pages (1 for each sorted run). We will call the buffer frame used to store the front of each sorted run the **input buffer**. The only thing we're missing now is a place to store our output. We need to write out records somewhere, so we need 1 more page, called the **output buffer**. Once this page has filled up, we flush it to disk and start constructing the next page. In total, we have two input buffers and 1 output buffer for a total of 3 pages required. This does not take advantage of all the memory that we have. Let's construct a better algorithm that uses all of our memory.

4 Full External Sort

Let's assume we have B buffer pages available to us. The first optimization we will make is in the initial "conquer pass." Rather than just sorting individual pages, let's load B pages and sort them all at once. This way we will produce fewer and longer sorted runs after the first pass.

The second optimization is to merge more than 2 sorted runs together at a time. We have B buffer frames available to us, but we need 1 for the output buffer. This means that we can have B-1 input buffers and can thus merge together B-1 sorted runs at a time. See the next page for a diagram of this sort assuming we have 4 buffer frames available to us.



Now we take in 4 pages at a time during the conquering phase and output a sorted run of length 4. In the merging pass, we can merge all three sorted runs produced during the conquering pass at once. This cut the number of passes (and thus our I/Os) in half!

5 Analysis of Full External Merge Sort

Let's now figure out how many I/Os our improved sort takes using the same process we did for Two-Way Merge. The conquering pass produces only $\lceil N/B \rceil$ sorted runs now, so we have fewer runs to merge. During merging, we are dividing the number of sorted runs by $B-1$ instead of 2, so the base of our log needs to change to $B-1$. This makes our overall sort take $1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$ passes, and thus $2N * (1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$ I/Os.