Section 4: Data Processing
Grouping Key/Value pairs
Part 2

Suppose that the data was extremely large, and did not fit in memory. What problems would this cause?

We know that by adding more memory to a system, we can increase the performance. If the input data is too large to be fit in main memory, the system tries to accommodate the data by setting up the Virtual Memory file. So, the CPU reserves space on the disk to simulate a sense of additional RAM. This is known as Swapping and it slows down and degrades performance considerably.

To explain it more, let us suppose we are attempting to load a program that does not fit in RAM memory. We expect that the Operating System will show us a message called "Insufficient Memory" or "Too little memory to load program". But in reality, this does not happen because of Virtual Memory. Due to this feature, the processor stores a file know as Swap File in the disk which contains RAM Memory data. So, overtime we want to land a program that does not fit in main memory, the Operating System sends to the Swap File in disk, parts of the programs that are currently stored in Ram but are not used or accessed. Thus it frees up space in Ram and we may now load the program we want to run. Similarly, when we want to access those programs which the system has written in the Swap File in disk, we write programs currently not in use from Ram inside Swap File in disk and transfer contents of Swap File in memory.

The real problem arises due to the fact that Disk is a mechanical device and not an electronic device. Therefore, obviously, the data transfer between disk and RAM is much slower than the data transfer between RAM and processor. The data transfer between RAM and processor is typically around 800MB/second while that between RAM and disk is around 33MB/second or 66MB/second depending upon the technology.

So every time we load a program too large for main memory, transfer of data takes place between RAM and disk and the whole system slows down. Upgrading or adding more RAM is not always the answer because not all machines are compatible with slots to accommodate external RAM. We must use external Algorithms and Data Structures which interact with Disk and RAM to solve our problems.

How could these problems be solved?

- We can keep the data in Binary Format like IGV that fits into memory.
- We can use Zram which swaps to a compressed RAM disk instead of swapping to disk because compression to Ram is much faster than disk I/O.
- We can preorder the Input Data and then process it in chunks that will fit in memory. Since the data is too large to be fitted in the main memory completely, the data must stay in the disk. This data must be brought into the main memory selectively for processing our algorithms. So, we must use Data Structures and Algorithms for these disk-based application. we have to minimize the number of disk access because access to data on disk is much much slower than access to data on memory. We need to use efficient Data Structures or File Structures which can arrange the data in such a way that fewer disk access is needed. We can cache some useful data by making predictions or guessing such that we can minimize the possibility of future disk accesses. For example, we can look into External Mergesort algorithm which is a Disk Based Algorithm. This algorithm sorts chunks of data that each fits well inside the RAM and then it merges the sorted chunks together. Hence, we can sort 900MB of data using only a 100MB RAM. The steps which we can take are:

1. Read 100MB of data in main memory and then sort it by using Quicksort.
2. Write back the sorted chunk of data in disk.
3. Repeat steps 1 and 2 until all the data is in chunks of 100MB and sorted. We can now do a 9-way (900MB/100MB=9 chunks) merge into a single output file.

4. Since the RAM is of 100MB, we can read 10MB of data from each sorted chunk kept in the disk into the main memory. So, we have used 90MB and the remaining 10MB is allocated to an output buffer.
5. Now we can perform a 9-way merge. Whenever we see that the output buffer is filled, we can write the buffer into the output file and flush the buffer. Whenever we see any of the 9 input buffers is empty, we can transfer the next 10MB of data from that chunk it its respective input buffer.
6. So, we can see that when we have data too large for memory, we can just load sequential chunks of it in main memory, process our algorithm and then fetch more data from disk into memory when needed.

- So, in this particular problem, if input data cannot fit in memory, we can use a similar sort of technique. If we have a RAM of 100MB, we can load a chunk of 100MB data from disk to main memory in an input buffer and start processing it with our algorithm. We can create an output buffer hold the results. Once the output buffer is full, we process the buffer and for every new value of Key that we find, we create a new output file for that Key and we write the corresponding Value inside that file of that Key. If the output file for a Key is already existing, we append the corresponding Value at then end of its file. After this, we flush the output buffer. Similarly, once our 100MB of input buffer is done processing and we reach the end of the buffer, we clear out the buffer and fetch the next 100MB of data as a chunk from the disk into the main memory.Finally, when we have processed the entire data from disk, we can merge all the intermediate output files intone single file which contains all the Values grouped by Keys, like we want.