Summary of Advanced Hashing and Sorting

Key k is placed into a bucket using the simplest approach: the bucket number is the remainder of k divided by the total number of buckets, N. The number of buckets is fixed from the outset; if a bucket is full, the entire structure must be rebuilt. If two keys collide, they can be linked together using a linked list, or they can be searched one by one to find an empty space.

Another approach is to avoid hard coding the number of buckets: buckets can be added or subtracted at any time as the amount of data increases, eliminating the need for a complete reshuffle and restart. This is how scalable hashing works, used by file systems like ZFS, GFS, and Spad FS. It still finds the data in one step and can grow almost as large as desired.

If the data is too large to fit in memory, an external sort is used. The simplest way to regress and merge versions uses only three caches, resulting in a total read and write count of approximately 2 × number of pages × (1 + logarithm of number of pages to base 2). Every page is read and written once during each pass through the file.

If the data already has a B^+-tree index based on the sort key, there's no need for a dedicated sort. When the index is "in order," the data pages are already largely sorted. Simply scanning the leaf level and retrieving the pages in order can help save a lot of extra reads and writes. Conversely, if the index is "random," with data pages scattered all over the place, retrieving records in key order may require a random jump for each record, which is prohibitively expensive and generally not cost-effective.

Expected Deliverables:

- Python code with clear documentation.
- Evidence of how the merge logic handles limited memory (buffer) constraints.
- Optional: log or print statements to visualize each merge pass.

Buffer Usage Limits:

The buffer size is n pages, but only n-1 pages can be used to read input runs. The last buffer page must be reserved for writing output. This limitation forces us to merge at most n-1 runs at a time.

Code: merge_degree = self.buffer_size - 1

Why use n-1 pages for reading:

If the buffer size is 3, only 2 runs can be merged at a time. The third buffer is needed to temporarily store the merged output before it is flushed to disk. This avoids data loss and simulates real disk I/O.

If all n pages were used for reading, there would be no space to store the merged results, resulting in data loss or overwriting of previously written data.

Merge Phase Logic:

Each merge merges multiple groups of runs using a priority queue. This process repeats multiple times until all data is merged into a single sorted run. Each merge adheres to the buffer limit, grouping at most n-1 runs at a time.

Eg. Degree = buffer_size –1 =2

First round: 100 -> 50

Second round: 50 -> 25

Third round: 25 -> 13

Forth round: 13 -> 7

Fifth round: 7 -> 4

Sixth round: 4 -> 2

Seventh round: 2 -> 1

I/O Count: The system simulates I/O operations (simulate_read and simulate_write) to simulate real disk costs, which is crucial for external sorting.

Code:

```
def simulate_read(self):
    self.io_count += 1  # Simulated disk read
def simulate_write(self):
    self.io_count += 1  # Simulated disk write
```