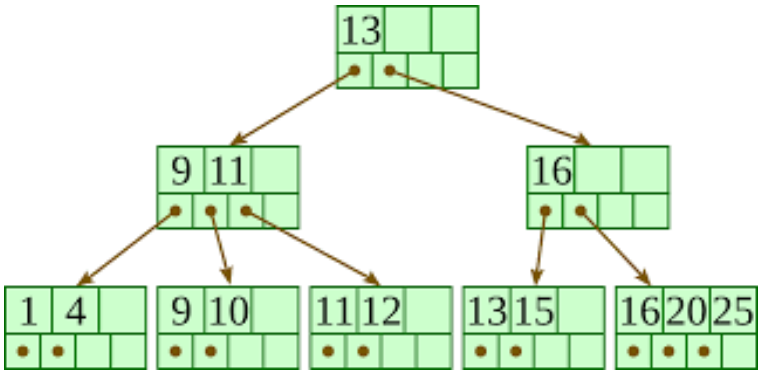
How to Optimise Read and Writes in Database

In a database we use **B+ tree** data structure which requires:-



Insertion time - O(log N)
Searching time - O(log N)

A server sends a request to the database and the database acknowledges that it has received the request. This can cause a lot of unnecessary data overhead.

To scale the database we need to reduce unnecessary exchange of data like acknowledgements and headers and also reduce I/O calls which helps in freeing resources and reducing response time.

In order to do it we can do the following:-

1. Condense data queries into single query:

    - Advantage:
      We can take multiple queries and process them in batches so as to reduce I/O operations (Number of Acknowledgements reduce).
    - Disadvantage:
      Additional memory is utilised by the server to condense it.

2. Use linked list:

    - Advantage:
      Linked list has an insertion time of O(1). So using linked lists in the ₹database can help increase write operations.
    - Disadvantage:
      Read operations are slow as search time in the linked list takes O(N) time.

3. Use Sorted array together with linked list:

    - Advantage:
      Search time becomes O(log N) and insertion time becomes O(1) hence improving read and write time giving optimal solution.

We need to sort the data before inserting it in the database so the data is sorted first and then persisted in the database. Data is processed in batches and these batches are called Sorted Chunks. We can apply binary search on them to search for the data required. Now searching in these chunks will still be slow as we need to apply binary search to each and every chunk.

So, we use a hybrid approach - Merging the sorted chunks in a sorted manner until the sort time is reduced. This would help to reduce the number of sorted chunks and hence improve the search time. But we can have chunks of different sizes.
So, we can also use bloom filters to reduce search time even more.

Bloom filters - A Bloom filter is a probabilistic data structure used to test whether an element is a member of a set or not. It offers an efficient way of checking for the existence of an item in a large set, while using a small amount of memory. The key idea behind a Bloom filter is to use multiple hash functions to map an element to several positions, and set those positions to 1.
However, due to the probabilistic nature of Bloom filters, **false positives** (reporting an element as being in the set when it is not) can occur.