

## Part B

1.

To optimize data retrieval and system performance for the database on Part A, we propose a hybrid indexing strategy utilizing Ordered and Hashing Indices.

Ordered Indices:

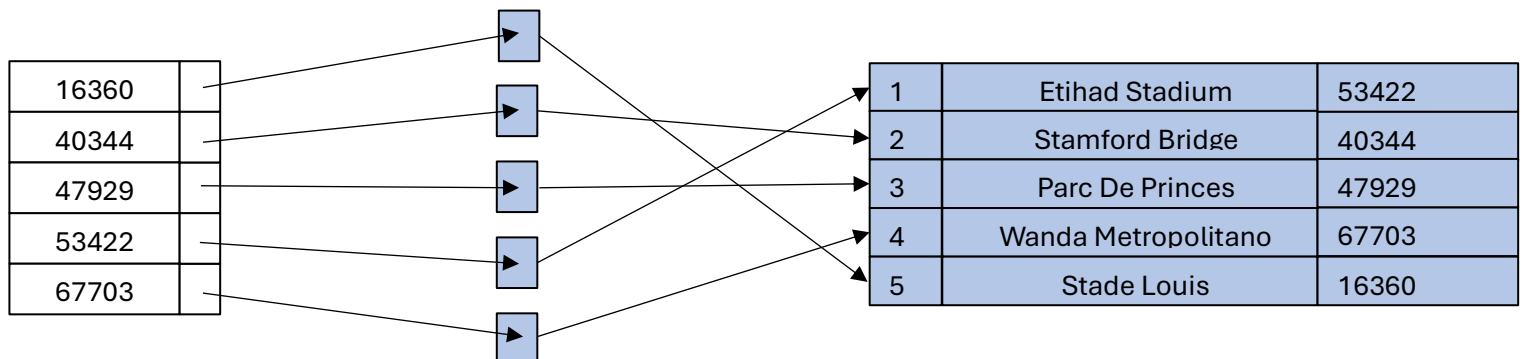
- Primary Index (or Clustering index):

A primary index is established where the search key defines the sequential physical order of the file. Typically, this corresponds to the primary key of the relation. For example, in our database the st\_id, tm\_id, ft\_id, sp\_id which serve as the primary key on their representing relation table will be the search keys for primary index.

- Secondary Index (or Non-Clustering index):

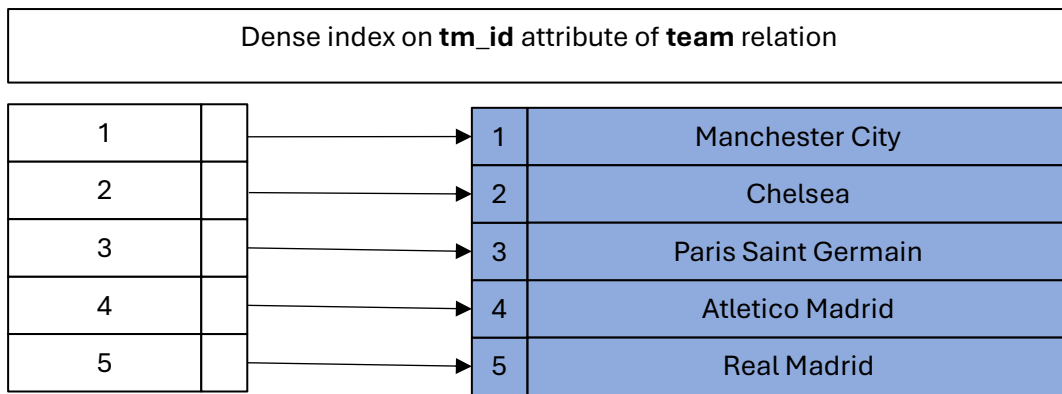
The search key specifies an order different from the sequential order of the file. Such search keys on the database would be st\_name, tm\_name, ft\_surname, sp\_name etc, where they can point to every actual record of their respective relation table (which makes them dense). A secondary index on the capacity attribute of the stadium relation allows for efficient querying of stadiums based on size without altering the physical organization of the data.

Secondary index on **capacity** field of **stadium**



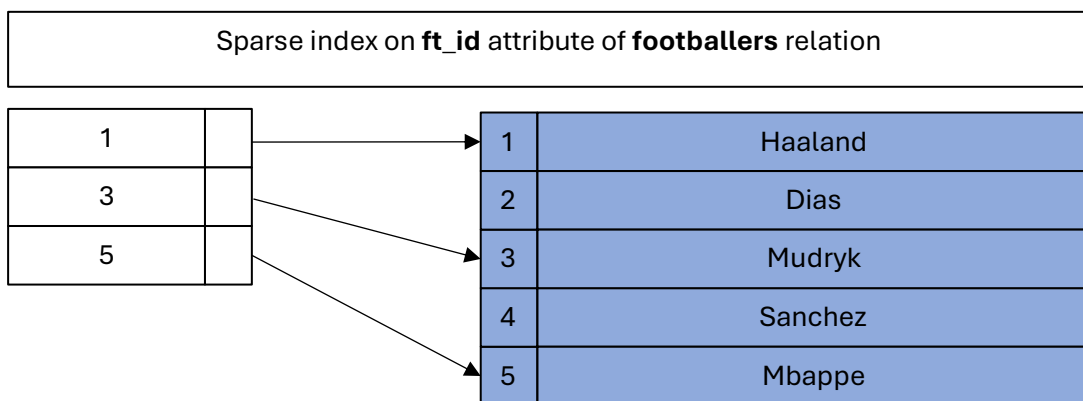
- Dense Index:

A dense index structure maintains an index record for every distinct search-key value. As an example, on the database every team (tm\_name) possesses a unique identifier tm\_id which minimizes access time by providing a direct pointer to the target block for every possible search.



- Sparse Index:

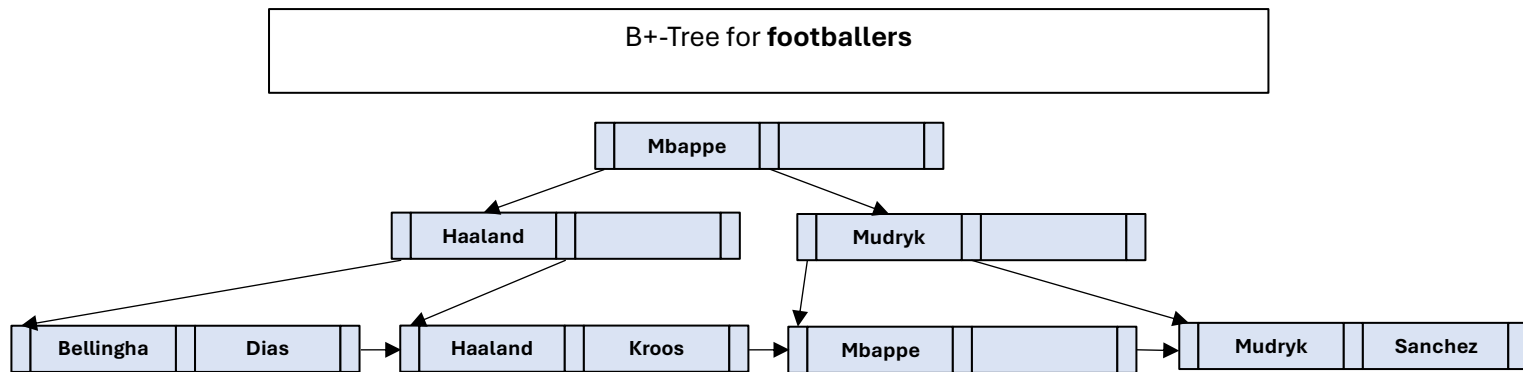
It contains index records for only a subset of search-key values. It is suitable for the **ft\_id** attribute of the **footballers** relation. By storing pointers for only selected keys (e.g., IDs 1, 3, and 5), the system reduces the storage footprint and maintenance overhead during insertions/deletions. This index structure may be slower, but it is easier to maintain and uses less space.



- B+ - Tree:

The B+ Tree is the dominant indexing architecture in relational databases, because the advantages outweigh disadvantages. It is designed to reduce disk I/O operations and automatically reorganizes itself with small, local changes in the face of insertions and deletions. The B+ Tree is superior for range queries as the database locates the starting key and sequentially scans the linked leaves, avoiding repeated tree traversals. In the database of part A it can be implemented on **footballers** relation table where the leaf nodes would be the surname of every player on the database. In that way we succeed a dynamic scalability where the B+ Tree can automatically manage node occupancy when we insert or delete a player. The following visualization represents such implementation with  $n = 3$  and it satisfies all the mandatory properties where:

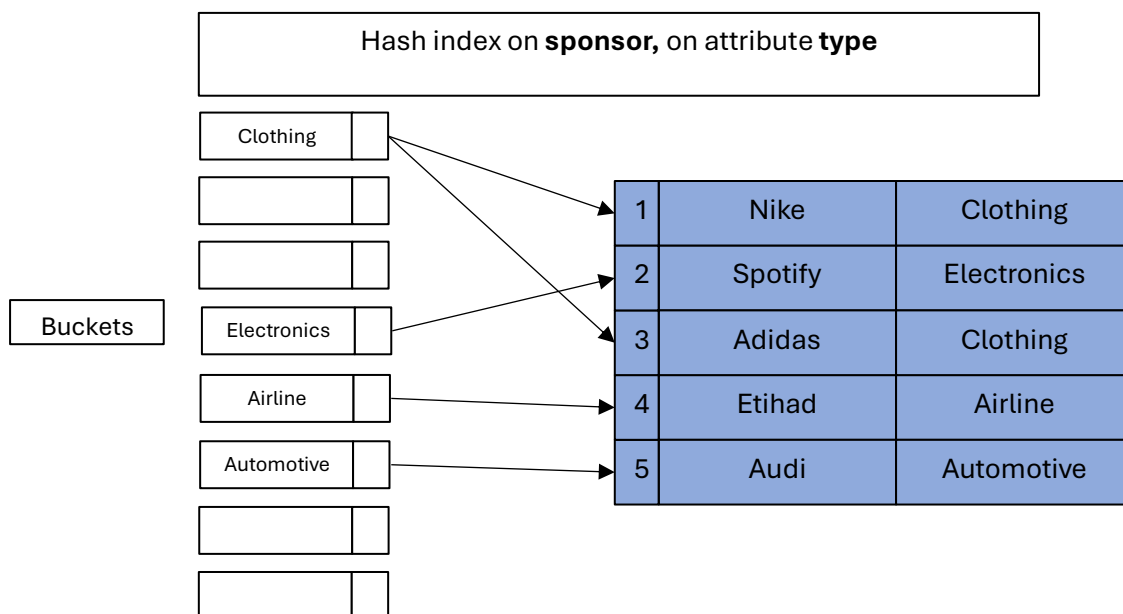
- leaf nodes must have between  $(n - 1)/2$  and  $n-1$  values
- non-leaf nodes other than root must have between  $n/2$  and  $n$  children and
- the root has at least 2 children.



### Hashing Indices:

#### - Static Hash Index:

It organizes search keys into a fixed number of buckets using a deterministic hash function. Such indexing technique can be applied on the database of part A, for example on type attribute of the sponsor relation. The hash function transforms the alphanumeric string of the sponsor type into a discrete bucket address via the modulo operation (  $h(\text{Clothing}) = h(88) = 0$  so the record is stored in bucket 0).



#### - Dynamic Hash Index:

On static hashing the number of bucket addresses is fixed which may lead to problems if a file grows (overflows) or shrinks (storage waste) significantly. Dynamic hashing is the solution where the hash function is modified dynamically. In the above example of the sponsor relation, the binary representation of the different types of sponsors will be the hash function and only a prefix will define the bucket addresses (  $h(\text{sp\_type})$  ).