DBMS Indexing

We know that information in the DBMS files is stored in form of records. Every record is equipped with some key field, which helps it to be recognized uniquely.

Indexing is a data structure technique to efficiently retrieve records from database files based on some attributes on which the indexing has been done. Indexing in database systems is similar to the one we see in books.

Indexing is defined based on its indexing attributes. Indexing can be one of the following types:

- **Primary Index:** If index is built on ordering 'key-field' of file it is called Primary Index. Generally it is the primary key of the relation.
- **Secondary Index:** If index is built on non-ordering field of file it is called Secondary Index.
- **Clustering Index:** If index is built on ordering non-key field of file it is called Clustering Index.

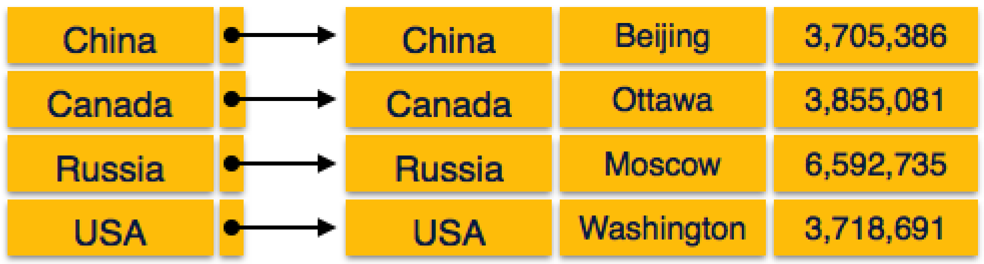
Ordering field is the field on which the records of file are ordered. It can be different from primary or candidate key of a file.

Ordered Indexing is of two types:

- Dense Index
- Sparse Index

Dense Index

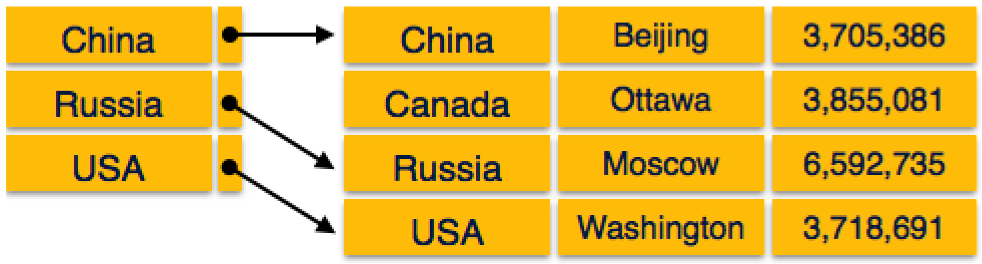
In dense index, there is an index record for every search key value in the database. This makes searching faster but requires more space to store index records itself. Index record contains search key value and a pointer to the actual record on the disk.



[*Image: Dense Index*]

Sparse Index

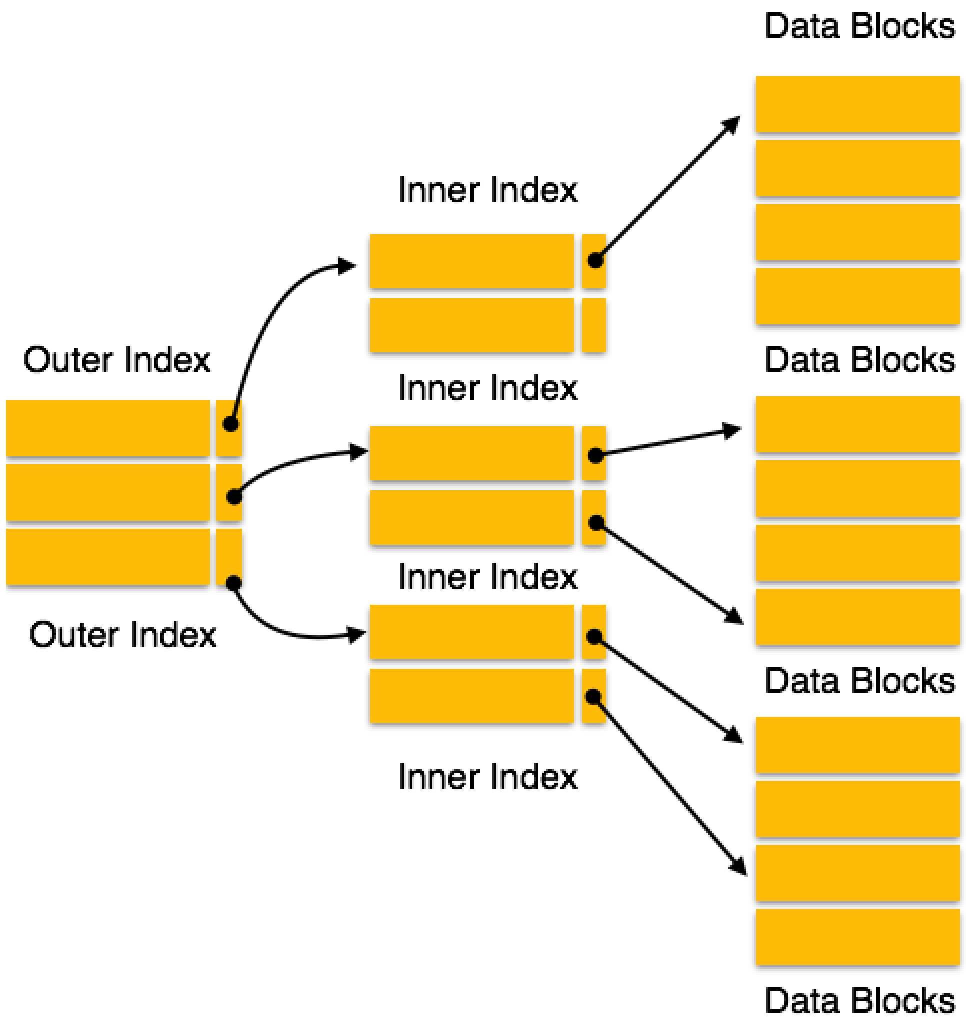
In sparse index, index records are not created for every search key. An index record here contains search key and actual pointer to the data on the disk. To search a record we first proceed by index record and reach at the actual location of the data. If the data we are looking for is not where we directly reach by following index, the system starts sequential search until the desired data is found.



[*Image: Sparse Index*]

Multilevel Index

Index records are comprised of search-key value and data pointers. This index itself is stored on the disk along with the actual database files. As the size of database grows so does the size of indices. There is an immense need to keep the index records in the main memory so that the search can speed up. If single level index is used then a large size index cannot be kept in memory as whole and this leads to multiple disk accesses.



[*Image: Multi-level Index*]

Multi-level Index helps breaking down the index into several smaller indices in order to make the outer most level so small that it can be saved in single disk block which can easily be accommodated anywhere in the main memory.

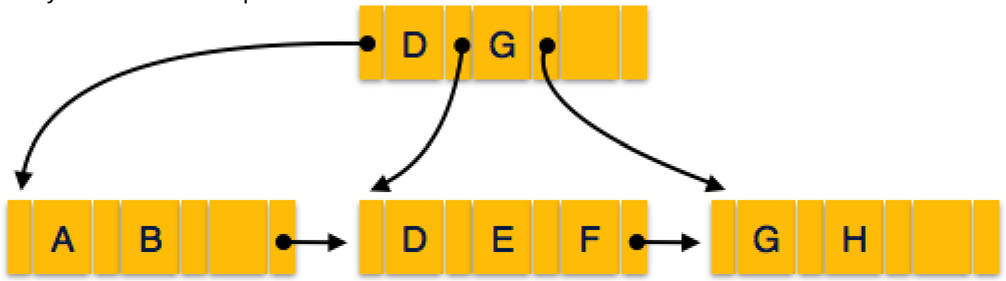
B+ Tree

B<sup+< sup=""> tree is multi-level index format, which is balanced binary search trees. As mentioned earlier single level index records becomes large as the database size grows, which also degrades performance.</sup+<>

All leaf nodes of B^+ tree denote actual data pointers. B^+ tree ensures that all leaf nodes remain at the same height, thus balanced. Additionally, all leaf nodes are linked using link list, which makes B^+ tree to support random access as well as sequential access.

STRUCTURE OF B+ TREE

Every leaf node is at equal distance from the root node. A B^+ tree is of order n where n is fixed for every B^+ tree.



[Image: B+ tree]

Internal nodes:

- Internal (non-leaf) nodes contains at least ⌈n/2⌉ pointers, except the root node.
- At most, internal nodes contain n pointers.

Leaf nodes:

- Leaf nodes contain at least ⌈n/2⌉ record pointers and ⌈n/2⌉ key values
- At most, leaf nodes contain n record pointers and n key values
- Every leaf node contains one block pointer P to point to next leaf node and forms a linked list.

B+ TREE INSERTION

- B^+ tree are filled from bottom. And each node is inserted at leaf node.
- **If leaf node overflows:**
  - Split node into two parts
  - Partition at i = ⌊(m+1)/2⌋
  - First i entries are stored in one node
  - Rest of the entries (i+1 onwards) are moved to a new node
  - i*th* key is duplicated in the parent of the leaf
- **If non-leaf node overflows:**
  - Split node into two parts

- Partition the node at i = $\lceil (m+1)_{/2} \rceil$
- Entries upto i are kept in one node
- Rest of the entries are moved to a new node

B$^+$ TREE DELETION

- B$^+$ tree entries are deleted leaf nodes.
- The target entry is searched and deleted.
   - If it is in internal node, delete and replace with the entry from the left position.
- After deletion underflow is tested
   - If underflow occurs
      - Distribute entries from nodes left to it.
   - If distribution from left is not possible
      - Distribute from nodes right to it
   - If distribution from left and right is not possible
      - Merge the node with left and right to it.

Source:

http://www.tutorialspoint.com/dbms/dbms_indexing.htm