Date: 04/09/2021 DBMS 16. clustered Index Abhishek Sharma Notes. By GFG placement course.

* Now we will understand How clustered index are implemented by DBMS servers:

DBMS server Maintain an Extra Index file.

That file has entries in this form:

Search_key	Pointer

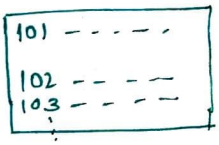
→ stores reference to disk block containing given key.

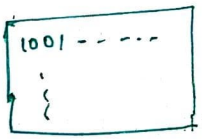
Example:

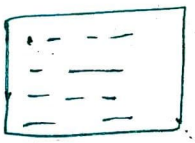
Table

Order ID	order Date	cost	CostumerID
101	– –	–	–
102	–	–	–
103	–	–	–
104	–	–	–
1001	–	–	–

Disk block







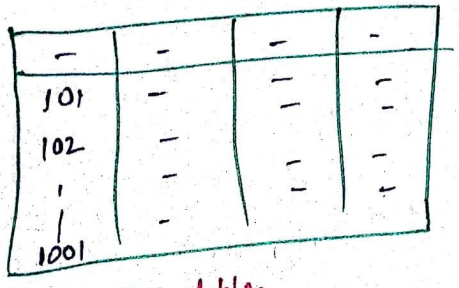
index file.

It contains search key along with the reference (Address of that block).

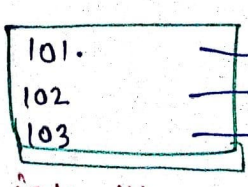
Let's say,

If we find Key 101

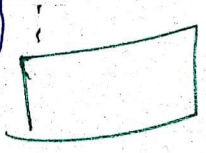
then it will come to the index file.



our table

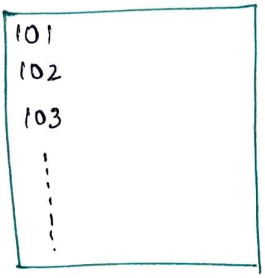


index file.



This is how we implement clustered key in our DBMS.

we store search key and we store disk block address where the search key items are present.



index file.

⇒ This index files are called Ordered index file. Or Sequencial index file organisation

↓

in this type of index file all the keys are present in Sequencial or ordered mamner.

like 101 - 102 - 103 then 104 and so on.

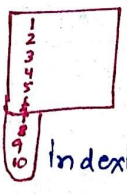
* Ordered index file or sequencial index file may be sparse or Dense.

Dense indexing: Every key has an entry in the index file and for every key value you have the disk Address.

So, index file might be big in dense index file.

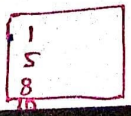
bcz every key value will be present.

Ex. (if table में order id में 1 से 10 तक Number है तो index file में भी 1 से 10 तक key होगी और उसका corresponding Address).



(Dense index file).

Sparse indexing: we can skip some Indexes in the index file like if we have 10 items (1 to 10) are in table then we can add in the index file like that



(Sparse index file)

Ex. (Only we can add 1, 5, 8 10)

In sparse indexing the index file is small as compared to Dense indexing.

bcz. in Dense indexing we have to add all the keys but in sparse indexing we can skip some keys (why we skip some key ??)

⇓

bcz all are in ~~sorted~~ increasing order.

if 1 comes then 2 will come after 1. that why we can skip some keys.

* Which is fast ?? sparse or Dense index ?? in searching a key

Ans :- Dense index is fast in searching a key bcz. All the keys are present there and we can search easily from there.

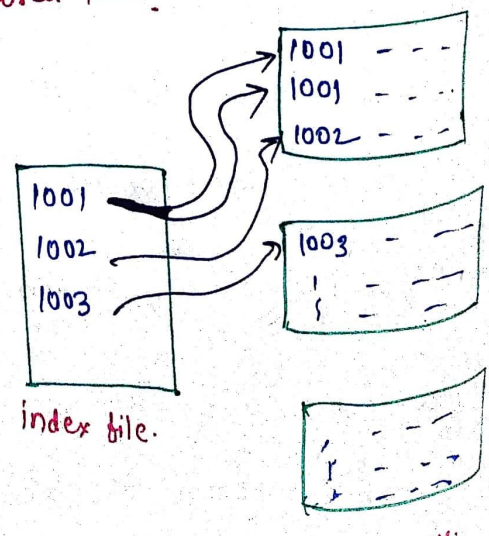
sparse index as it contain only some keys so it might be slow compare to Dense index.

Let's see an Example.

clustered index.

Order ID	Order Date	Cost	Customer ID
1	-	-	1001
2	-	-	1002
3	-	-	1003
4	-	-	1001
.	-	-	.
.	-	-	.
.	-	-	.

Our table.



Explanation: We want that customer ID as a clustered index

So, we found that there are one key is appearing more than one time (multiple time) "(1001)" So, in index file we will Right only once.

* in index file every entry has to be present exactly once. (in Dense file system).

But in disk block we will give the space for both the entry (1001) twice. how many times it may appear. it will be shown in disk block.

But in index file it will be only once.

17. Non clustered Index.

Q. How non-clustered index are implemented by DBMS Server ?

A. we will learn it via example.

clustered index →

Order ID	order date	cost	Consumer ID.
101	15-6-19	2000	1005
102	15-6-19	5000	1001
103	15-6-19	300.	1005
104	15-6-19	1000	1002
105	15-6-19	2000	1003
106	15-6-19	5000	1005
~~109~~			
⋮	⋮	⋮	⋮

→ Non clustered index.

we have a clustered Index. = Order ID.

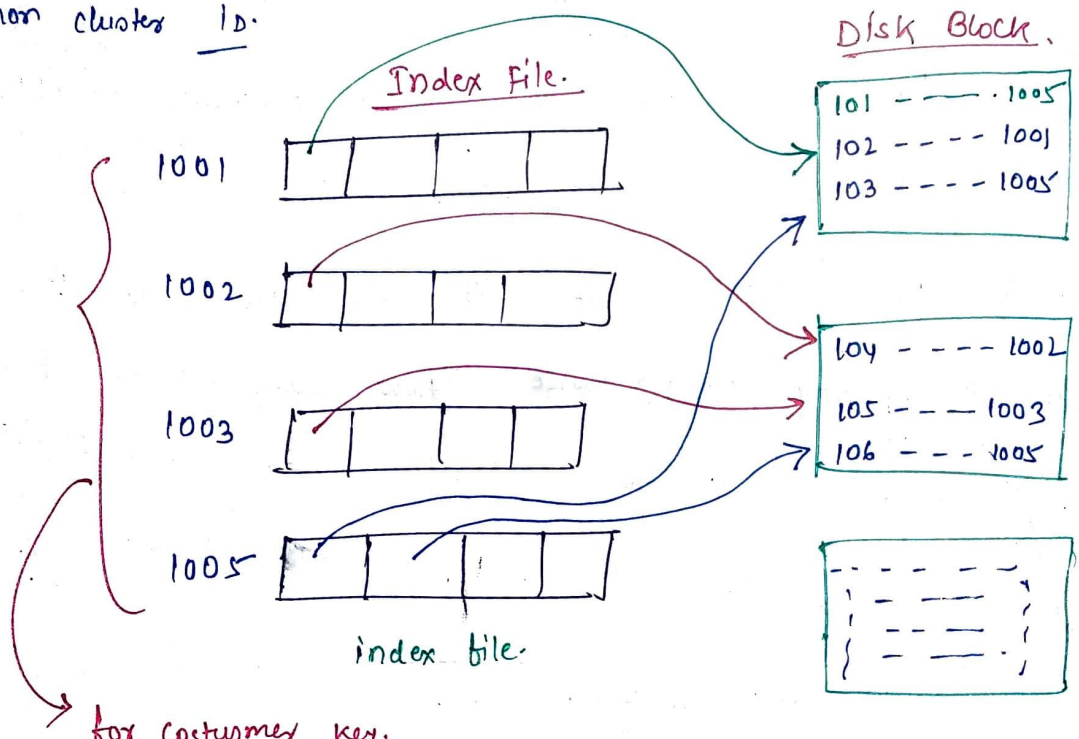
we " " non " " = Consumer ID.

We want to implement in our DBMS.

So DBMS server does the following task.

DBMS server makes an index file & like that for Disk block.

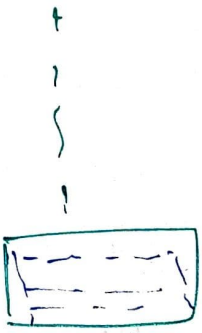
non cluster ID.



for costumer key.

it will be in increasing manner.

This is how secondary indexes are created for every Attribute.



Disk Block.

* non clustered indexing always be Dense.

bcz there have to present all the keys.

* They are not dense in implementation.

* we should not create many secondary indexes we ~~for~~ Should create for that attribute only who is most Searched Attribute.

18. Multi Level Indexing.

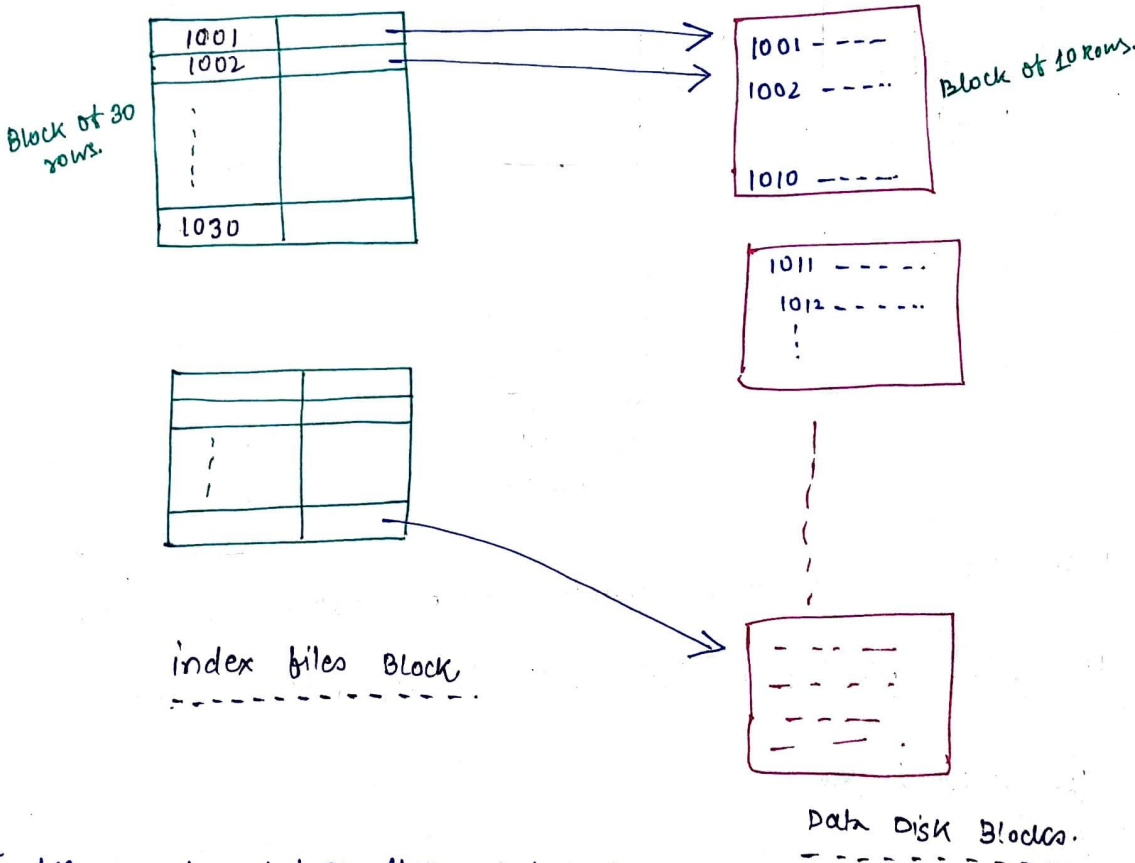
Abhishek sharma Notes.

Multilevel Indexing are used when our Database has very large number of Records.

As the time increases the databases ~~increase~~ grow with the time so the index files also grows with time.

So this index files may not fit in one single block. They might be spreading along multiple disk blocks.

Let's see an example.

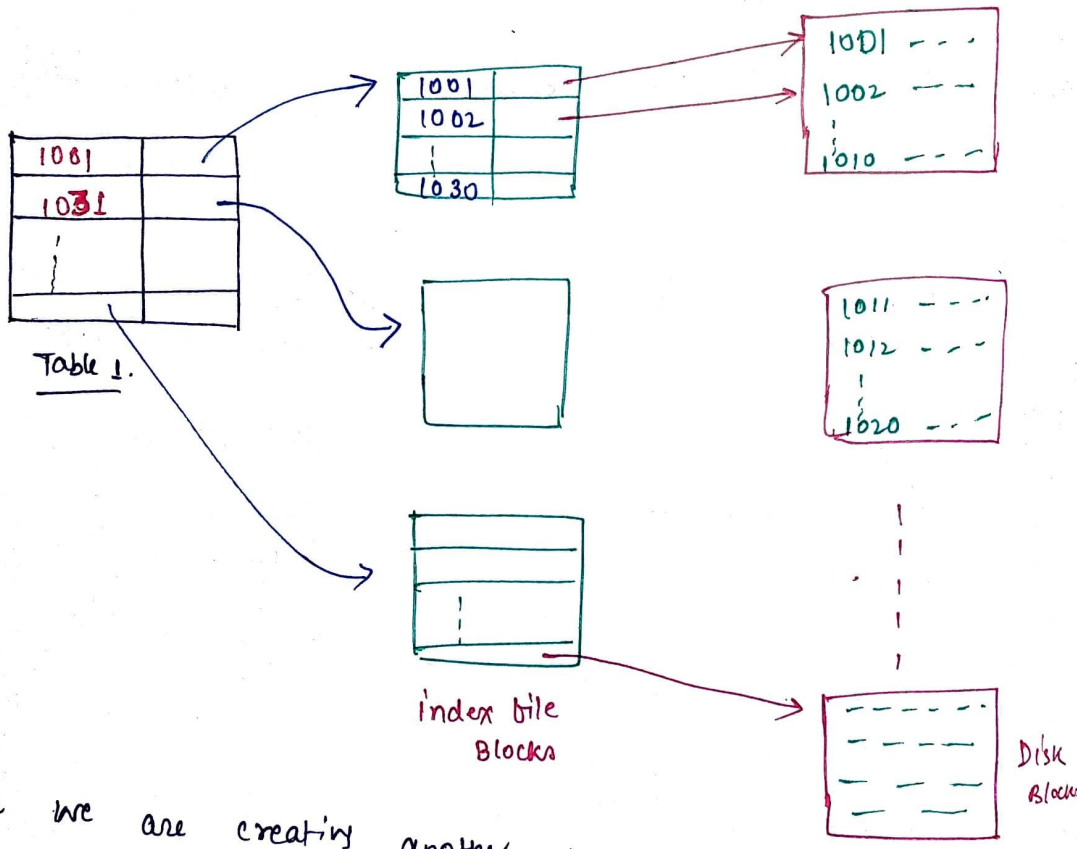


If we go in future then index files grow with time that the cause of Disk Blocks also grow with time in our databases.

So how can we handle this. If we search here any key in index file then it will take O(n) times. at max. (worst case). we have to search from the full index.

So, from get rid of this we will use Multilevel indexes. such that if we search a key then it will take minimum time.

we will Create Multilevel indexes like this:

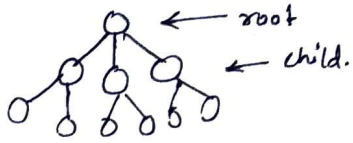


Here we are creating another table which maintain the index file. and index file blocks will maintain the disk blocks.

if we want to search for (1032) then we will find it in 2 comparsion of blocks. (so the time Reduces from O(n).

So, it is more like a tree structure if we rotate it.

The table which we created will act as root node then it's child (i.e. index files) and index files child are disk blocks.



Advantages of Multilevel indexing.

if we want to find last key then we have to go through only 2 blocks. that is Table 1 and index file's block. So we found the last key in minimum no. of blocks. But in earlier we should have to go till the last block in the index file. i.e. (O(n)) time earlier we should take in the worst case. So via using of Multilevel indexing we find the record in minimum no. of times.

* Problems in Multiplelevel indexing.

As per the time grows we need to modify our databases as these are huge no of records. if a person leaves the table then we should have to shrink the table also. So to get rid of these problems the idea of B tree and B^+ Tree. comes.

* How to get rid of problems in Multi level indexing.

using B and B^+ tree. (They modify themselves. grows and shrink when the data coming in and deleted)

These are the Balanced Binary Tree * like Data Structures.