DBMS Tutorial 28.11.2018

Topics

Indexes : Primary, Secondary ; Dense, Sparse

Indexes in database management systems can be categorized into primary and secondary indexes, as well as dense and sparse indexes.

Primary indexes are associated with sorted data files based on the primary key and control the physical placement of records.

Secondary indexes provide additional access paths to data and are not tied to the physical organization of data files.

Dense indexes contain an entry for every record, offering direct access based on key values, while sparse indexes have entries for only some records, requiring less storage space.

Overall, indexes play a critical role in enhancing query performance by facilitating efficient access to data records based on specified attributes.

Select * From R

If we scatter the records that represent tuples of the relation among various blocks. Then we would have to examine every

block in the storage system, and we would have to rely on there being :

1. Enough information in block headers to identify where in the block records begin.
2. Enough information in record headers to tell what relation the record belongs to.

Better Idea : reserve some blocks, perhaps several whole cylinders, for a given relation. All blocks in those cylinders may be assumed to hold records that represent tuples of our relation. Now, at least we can find the tuples of the relation without scanning the entire data store.

Select * From R Where name = 'xyz';

Would require us to scan all the blocks on which R tuples could be found.

Better Idea: To facilitate queries such as this one, we often create one or more indexes on a relation on the fields which are likely to be query parameters. So that we can find the record with that parameter quickly.

The field(s) on whose values the index is based is called the search key, or just "key" if the index is understood.

Keys

Sort Key : the attribute(s) on which a file of records is sorted.

Search Key : the attribute(s) for which we are given values and asked to search through an index for tuples with matching values. Eg. Select * From R Where **name** = 'xyz';

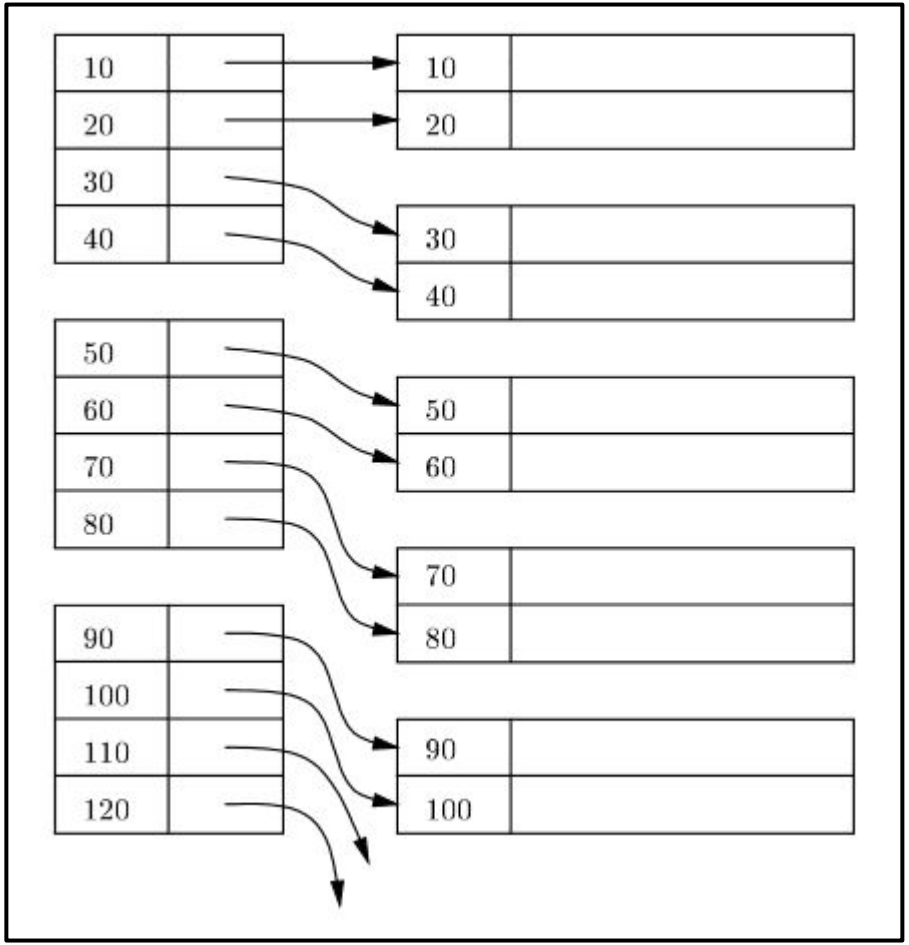
Primary Key : "*primary key of a relation*." These keys are declared in SQL and require that the relation not have two tuples that agree on the attribute of the primary key. ***Sometimes*** the sort and/or search key is the primary key as well.

Index

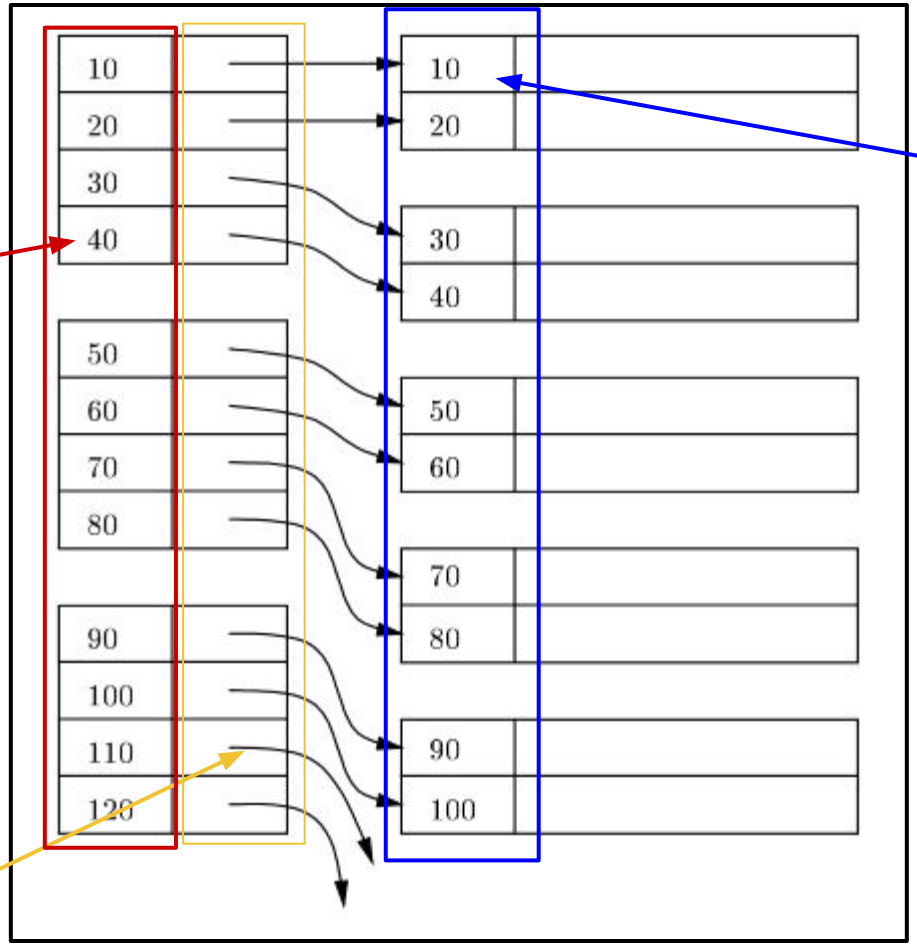
An index is a data structure that supports efficient access to data. *Add indirection for the sake for efficiency.*

In particular, an index lets us find a records without having to look at more than a small fraction of all possible records.

Since keys and pointers presumably take much less space than complete records, we expect to use many fewer blocks for the Index than for the file itself.



Index



Search Key

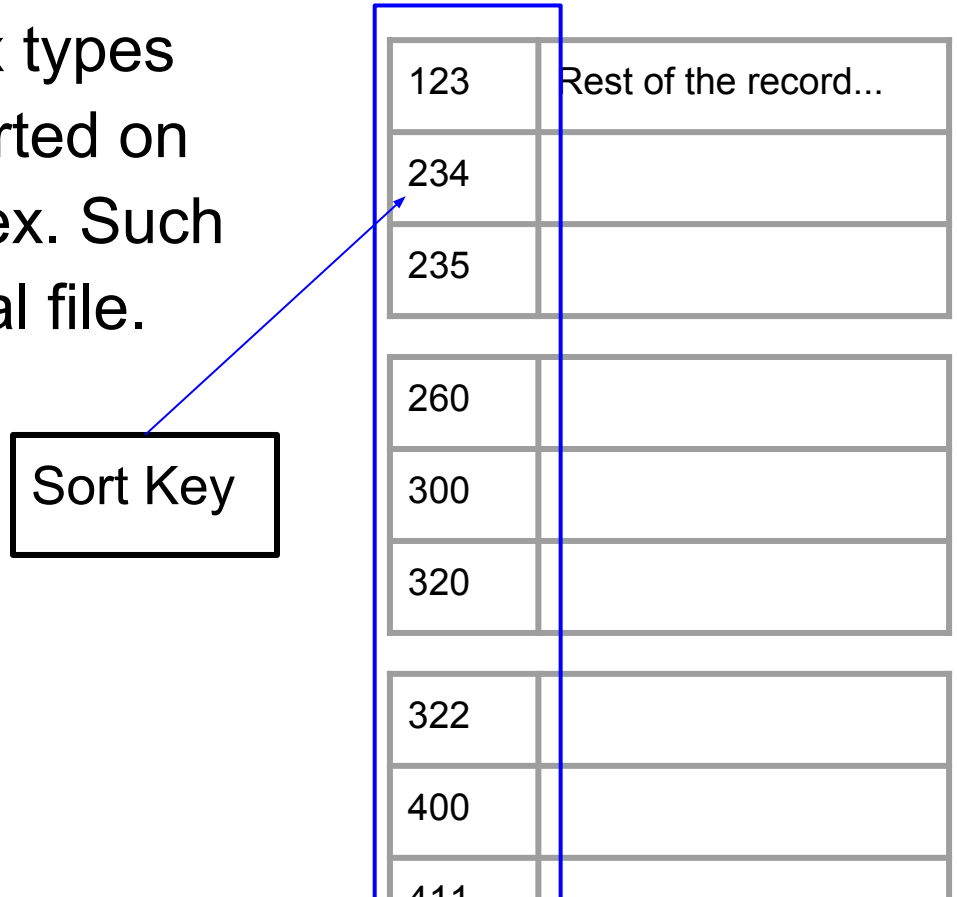
Sort Key

Pointers to records

Sequential Files

One of the simplest index types relies on the file being sorted on the attribute(s) of the index. Such a file is called a sequential file.

Tuples are stored in a chronological order based on the sort key.

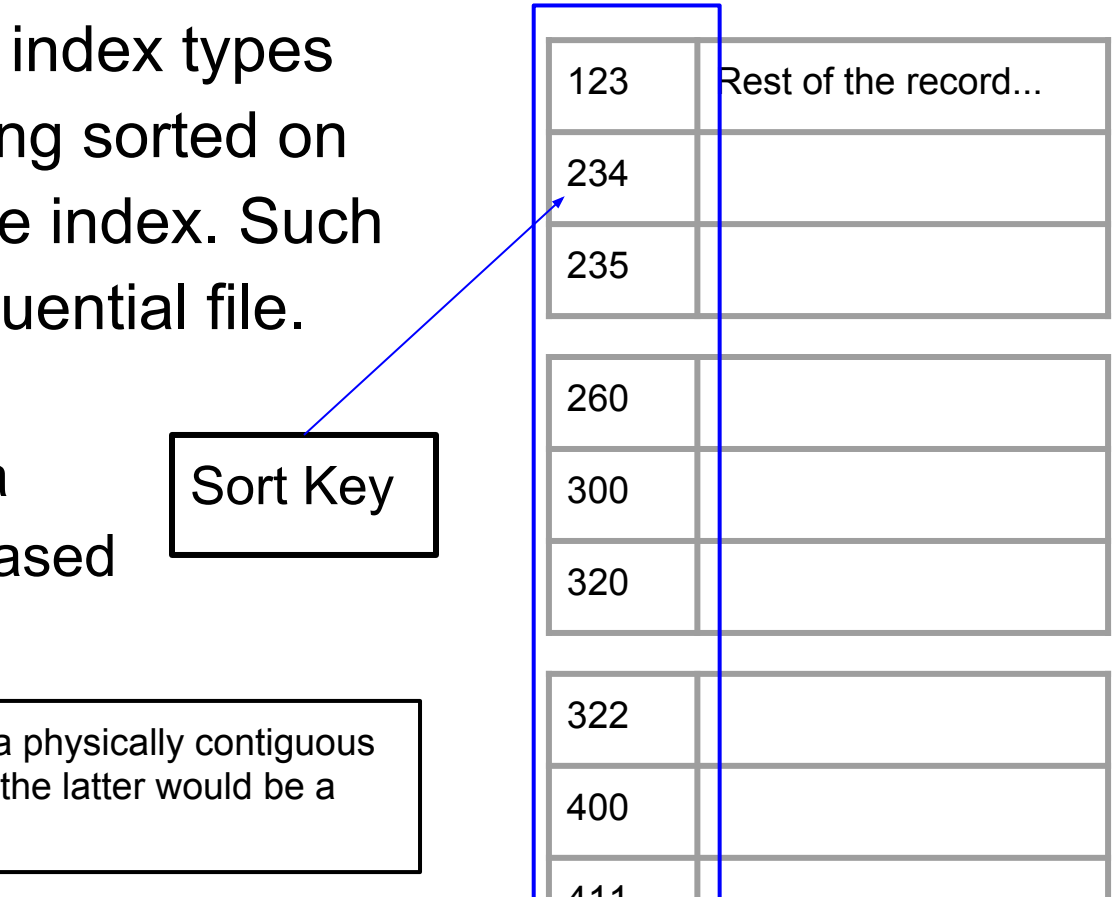


Sequential Files

One of the simplest index types relies on the file being sorted on the attribute(s) of the index. Such a file is called a sequential file.

Tuples are stored in a chronological order based on the sort key.

Sort Key



A sequential file may be stored in a physically contiguous manner or in an arbitrary manner (the latter would be a linked list style.)

Dense Index

Dense index is a sequence of blocks holding only the keys of the records and pointers to the records themselves.

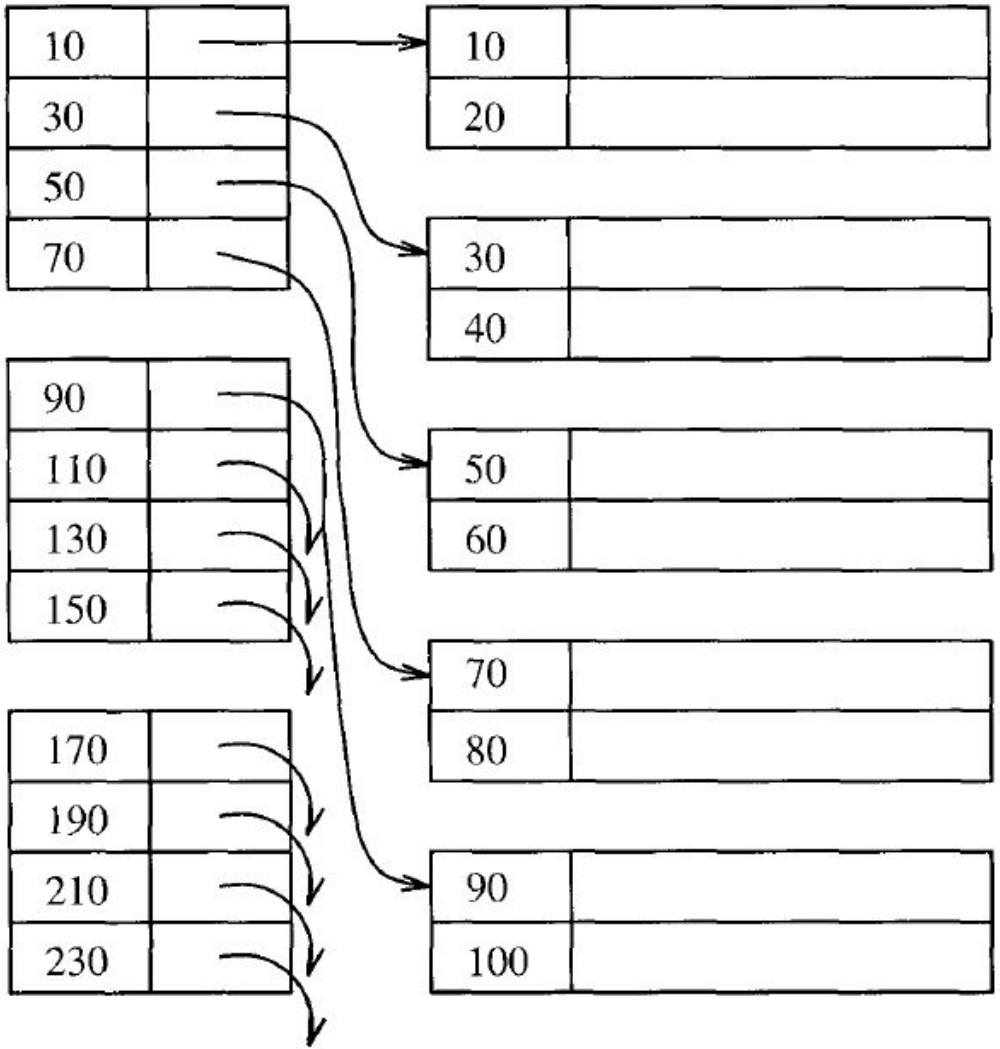
The index is called "dense" because every key from the data file is represented in the index.

The index is especially advantageous when it, but not the data file, can fit in main memory. Then, by using the index, we can find any record given its search key, with only one disk I/O per lookup.

Sparse Index

Sparse indexes normally keep only one key per data block in the index.

A sparse index, as seen here, holds only one key-pointer per data block. The key is for the first record on the data block.



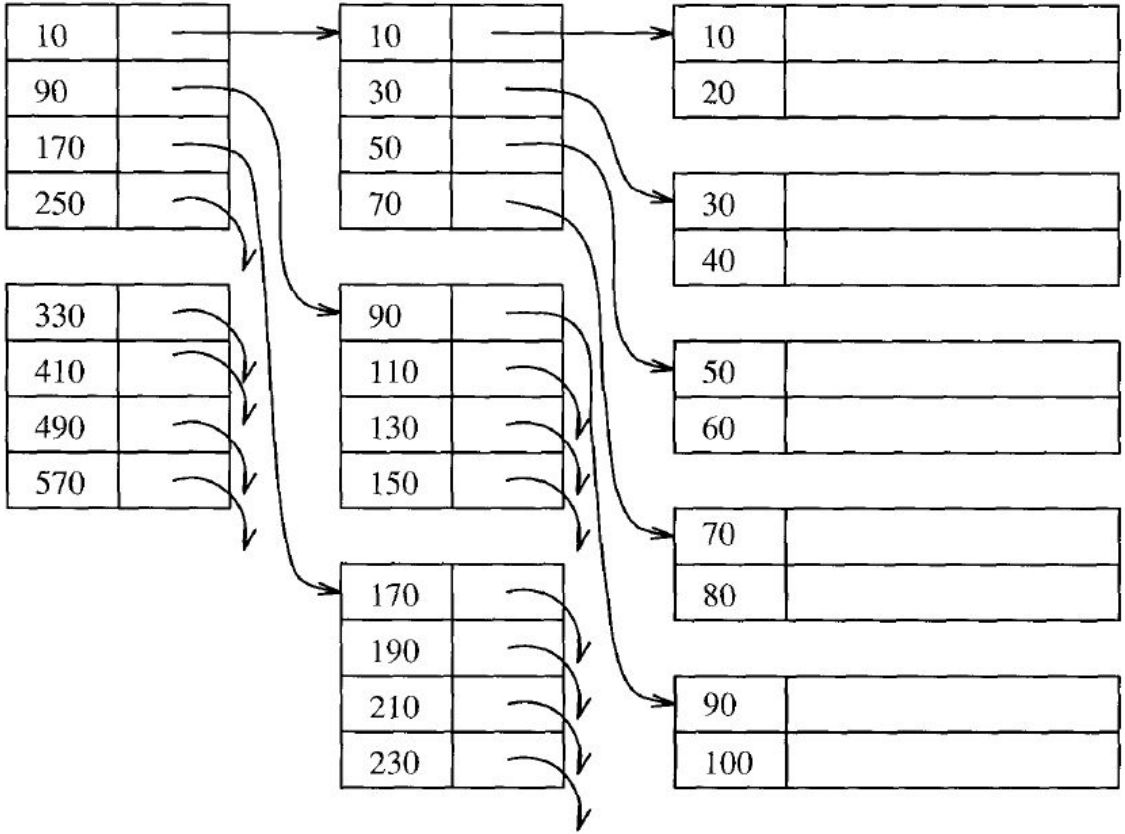
Dense vs Sparse

Dense : You can find out if a record exists or not by just looking at the index!

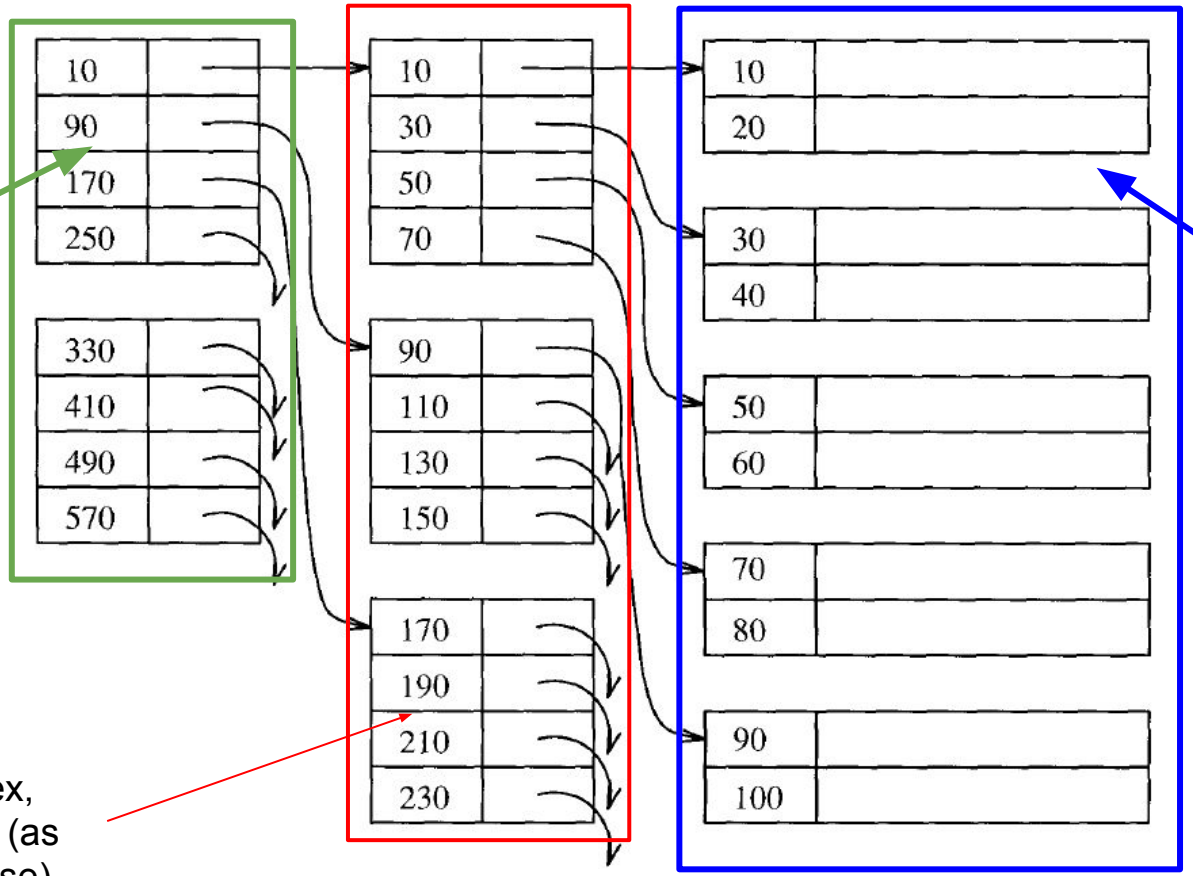
Sparse : You need to make an IO to find out if a record doesn't exist. (find greatest key less or equal to K)

We can keep small indexes in memory. Since sparse index takes much less space, it will be easier to keep it in memory (maybe as a higher level index).

Multi-level Indexes



Multi-level Indexes



Second level index **always sparse**!

First level Index, can be sparse (as shown) or dense)

Relation or File

Multi-level Indexes

By putting an index on the index, we can make the use of the first level of index more efficient.

Note : A second-level *dense* index thus introduces additional structure for no advantage. The reason is that a dense index on an index would have exactly as many key-pointer pairs as the first-level index, and therefore would take exactly as much space as the first-level index.

Indexes : Primary and Secondary

Primary Index : is on the sort key, can be dense or spare. **Decides how the file is placed on disk.** There can be only one primary index.

Secondary Index: We might want an index on something other than the sort key this would be Secondary Index. Always dense. There can be multiple secondary indices for the same relation. **If its multi-level, higher levels can be sparse, but the first level always dense.**

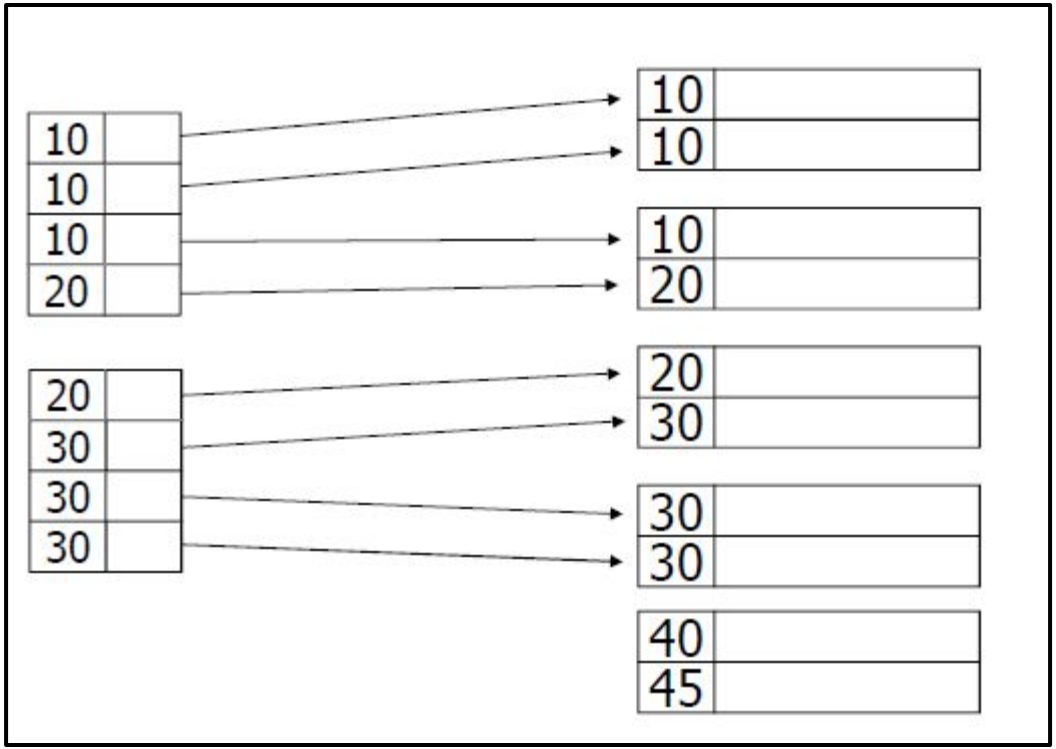
All index files are always sorted on the search key!

Duplicates with Dense Index 1

Since the search/sort key is not always the primary key, there can be duplicates.

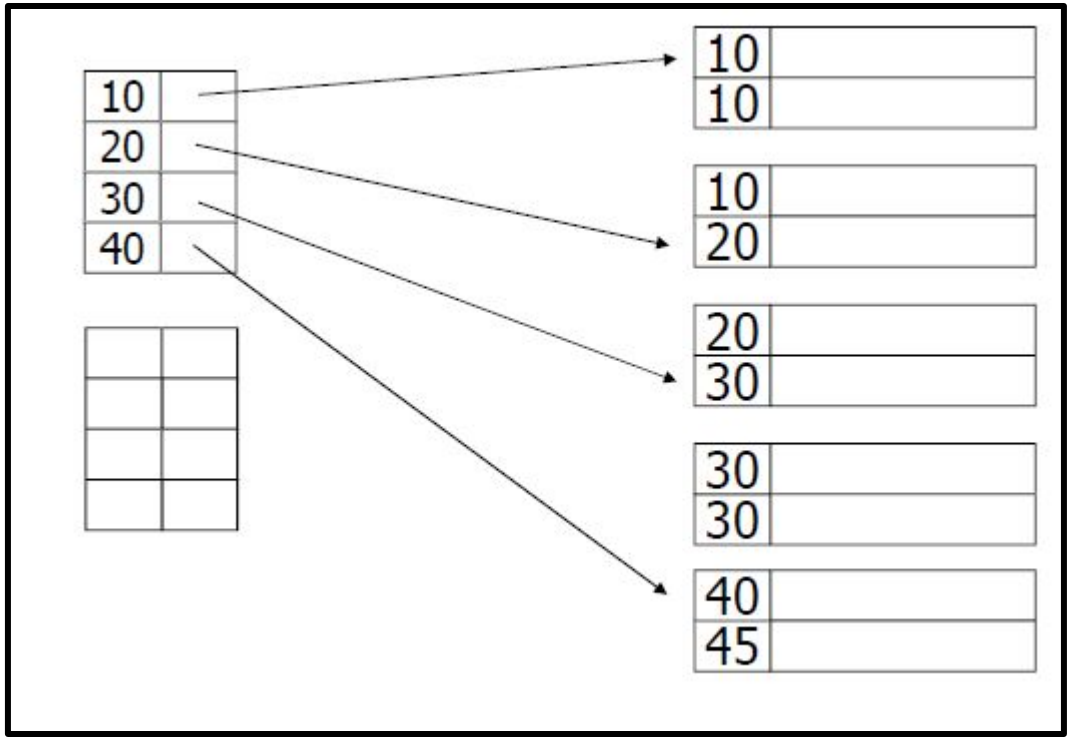
The way simplest way to deal with duplicates would be to have pointers to all records.

Look up: follow all relevant pointers.



Duplicates with Dense Index 2

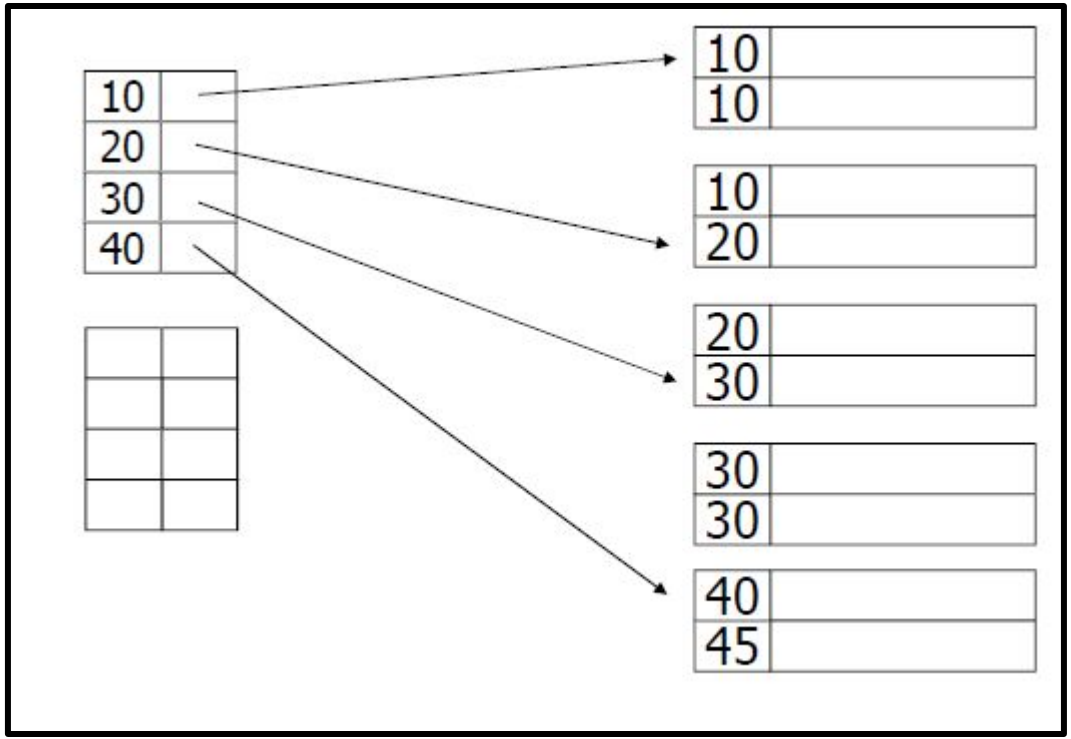
A slightly more efficient approach is to have only one record in the dense index for each search key K. This key is associated with a pointer to the first of the records with K.



Duplicates with Dense Index 2

Look up: To find the others, move forward in the data file to find any additional records with K; these must follow immediately in the sorted order of the data file.

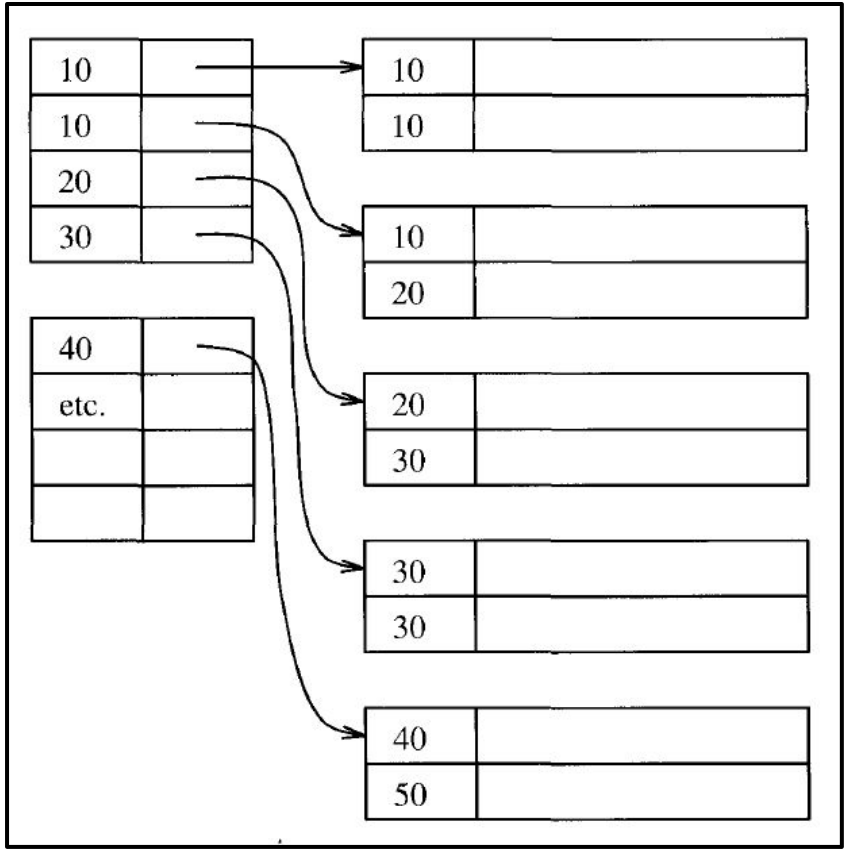
Careful: This looks like a sparse index but its not!



Duplicates with Sparse Index 1

Simplest: Still point to the first record of each block.

Lookup:To find the records with search key K, we find the biggest entry of the index that has a key less than or equal to K (call it E1). We then move towards the front of the index until we either come to the first entry or we come to an entry E2 with a key strictly less than K. E2 could be E1. All the data blocks that might have a record with search key K are pointed to by the index entries from E2 to E1 inclusive.



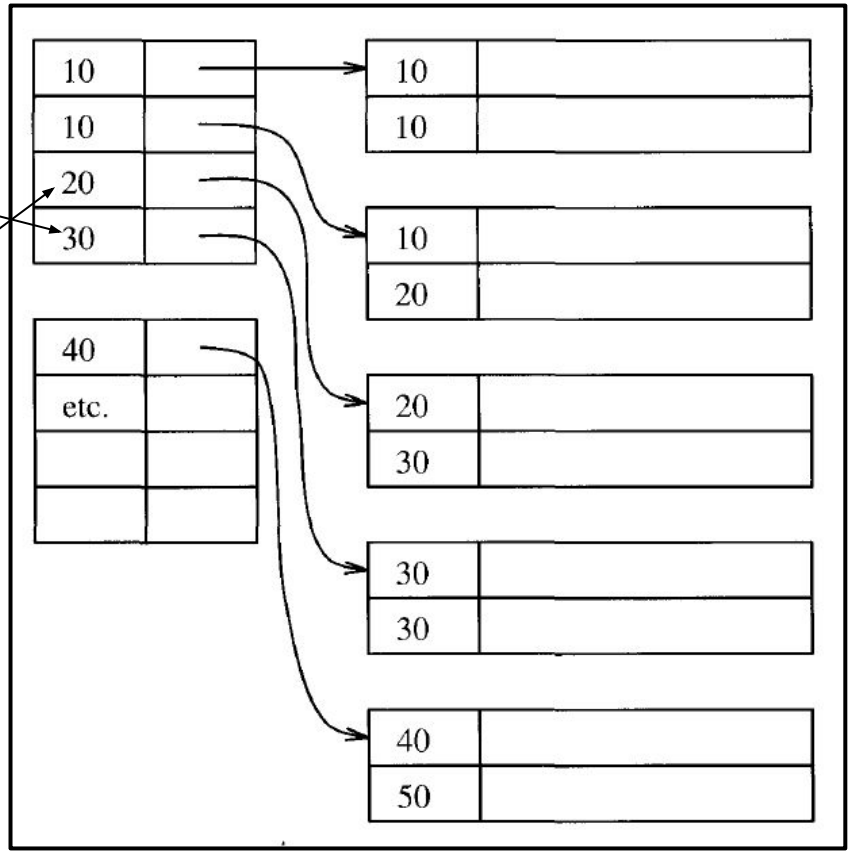
Duplicates with Sparse Index 1

For Example: Looking for 30, first we come to the fourth entry

Then we go up till we arrive at an entry strictly less than 30, which happens to be the previous one (third entry).

All the records with the key 30 will be contained in these two blocks.

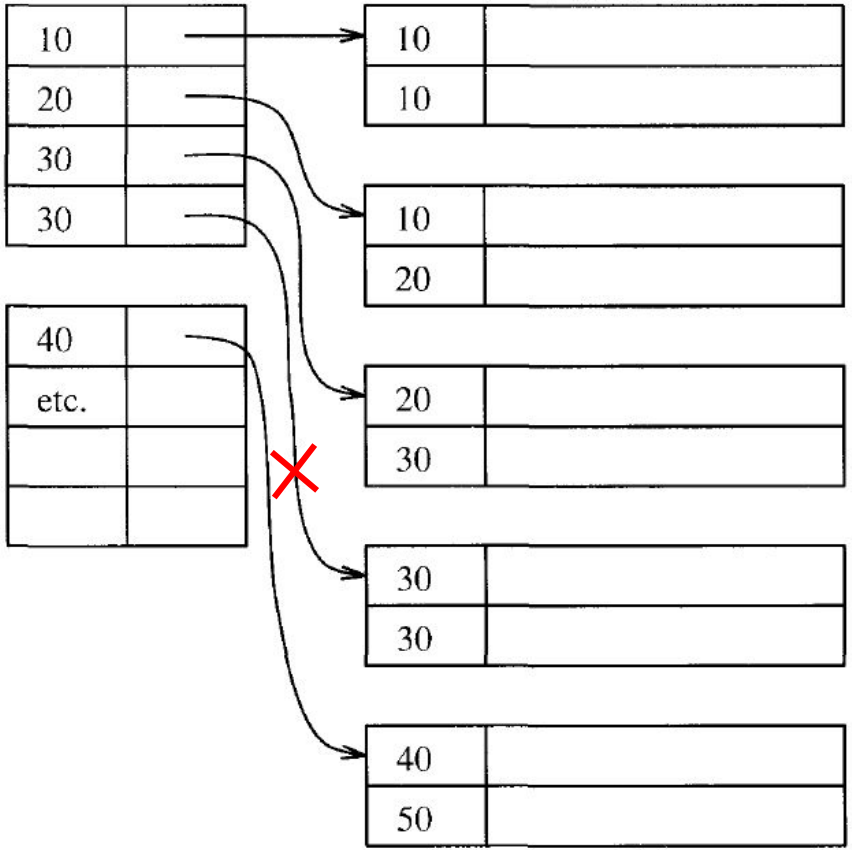
If K = 35, we would still select the same first block (E1) as 35 ≥ 30. But since 35>30, E1=E2 and we have to inspect only one block



Duplicates with Sparse Index 2

The index entry for a data block holds the smallest search key that is new; i.e., it did not appear in a previous block.

If there is no new search key in a block, then its index entry holds the lone search key found in that block. (In the profs slides it's suggested that we might skip this entry. The red cross denotes that.)



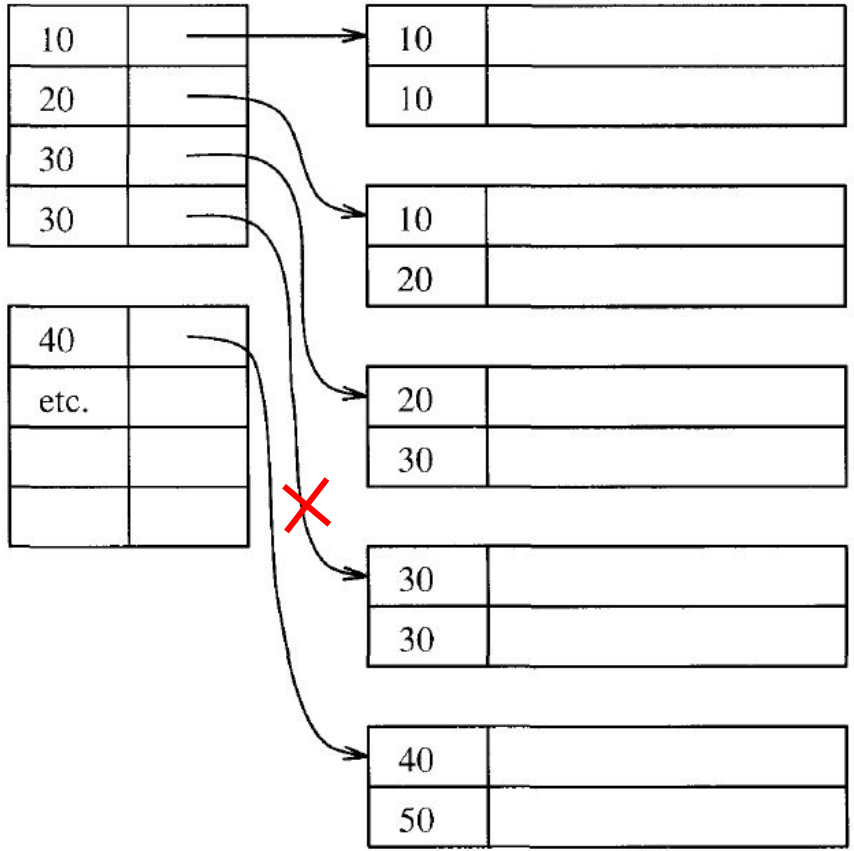
Duplicates with Sparse Index 2

Look up: the records with search key K by looking in the index for the first entry whose key is either

a) Equal to K or

b) Less than K, but the next key is greater than K.

We follow the pointer in this entry, and look for the record with search key K in that block. If that key is the last entry in the block we search forward through additional blocks until higher key is met.

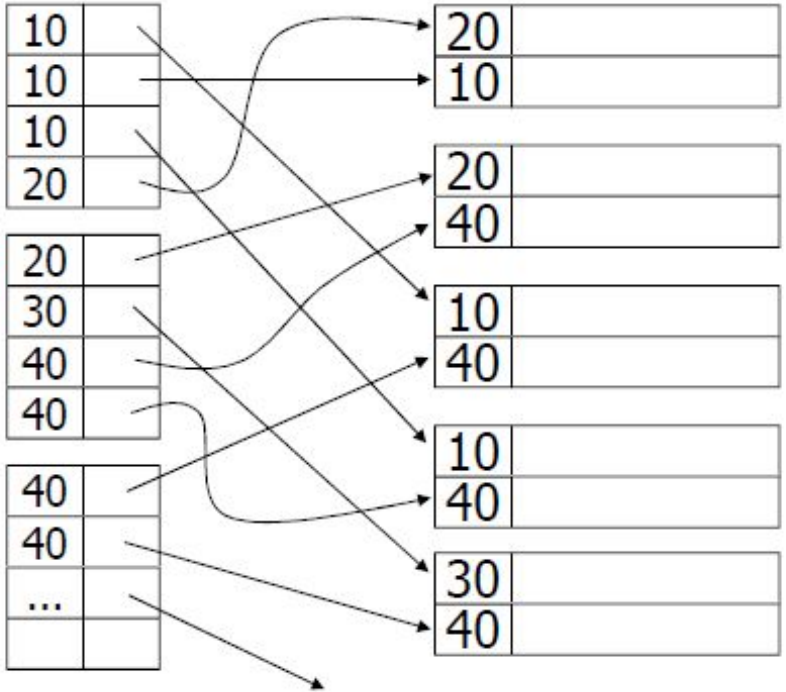


Duplicates with Secondary Index

- one option...

Problem:
excess overhead!

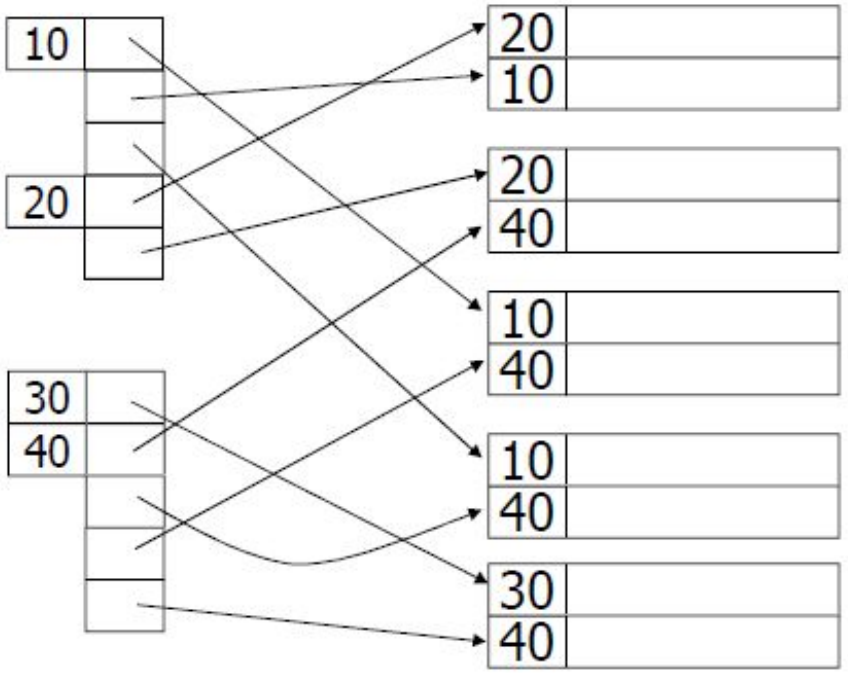
- disk space
- search time



Duplicates with Secondary Index

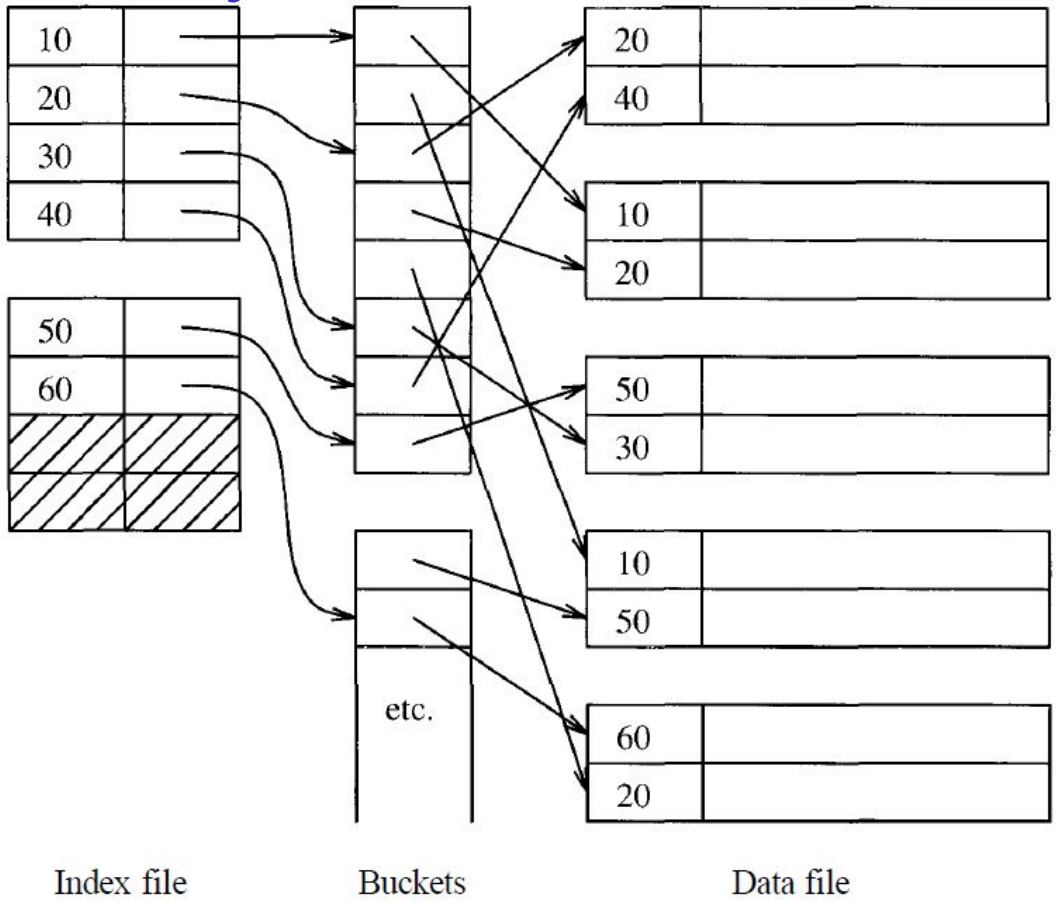
another option...

Problem:
variable size
records in
index!



Duplicates with Secondary Index : Buckets

A convenient way to avoid repeating values is to use a level of indirection, called buckets, between the secondary index file and the data file. There is one pair for each search key K. The pointer of this pair goes to a position in a 'bucket file'. Which holds "bucket" for K. Following this position (until the next position pointed to by the index) are pointers to all the records with search-key value K.



Exercise 13.1.2 DSCB

Exercise 13.1.2:If blocks can hold up to 30 records or 200 key-pointer pairs, but neither data- nor index-blocks are allowed to be more than 80% full. As a function of n, the number of records, how many blocks do we need to hold a data file and:

a) A dense index?

b) A sparse index?

Exercise 13.1.2 DSCB Sol

Exercise 13.1.2:If blocks can hold up to 30 records or 200 key-pointer pairs, but neither data- nor index-blocks are allowed to be more than 80% full. As a function of n, the number of records, how many blocks do we need to hold a data file and:

80% full ⇒ we can fit only 24=(30*0.8) records or 160(200*0.8) KP.

Blocks for Data File = ceil(n/24)

a) A dense index?

Blocks for Dense Index = ceil(n/160) Note: Number of entries or Key-pointer pairs in a dense index = number of records. or key pointer pairs

b) A sparse index?

Blocks for Sparse Index = ceil(ceil(n/24)/160) Note. Number of entries or Key-pointer pairs in a sparse index = number of blocks the data file takes..

Exercise 13.1.3 DSCB New!

Exercise 13.1.3: Repeat Exercise 13.1.2 if we use as many levels of index as is appropriate, until the final level of index has only one block. Assume n to be 10^5.

Exercise 13.1.3 DSCB Ans.

Exercise 13.1.3: Repeat Exercise 13.1.2 if we use as many levels of index as is appropriate, until the final level of index has only one block. Assume n to be 10^5.

Data file = 4167 100,000 / 24

Dense Index (plus 2 higher levels till we reach 1 block)

= 625 + 4 + 1 100,000/160 = 625

Sparse Index (plus 1 higher level till we reach 1 block)

= 27 + 1 ((100,000/24)/160) = 27 => 27 / 160 = 1

Exercise 13.1.4

Exercise 13.1.4: Suppose that blocks hold three records or ten key-pointer pairs but duplicate search keys are possible. To be specific, 1/3 of all search keys in the database appear in one record, 1/3 appear in exactly two records, and 1/3 appear in exactly three records. Suppose we have a dense index, but there is only one key-pointer pair per search-key value to the first of the records that has that key. If no blocks are in memory initially. Compute the average number of disk I/O's needed to find all the records with a given search key K. You may assume that the location of the index block containing key K is known, although it is on disk.

Exercise 13.1.4 Answer

Since the location of the index block containing key K is known, although it is on disk we invest 1 IO to read this block.

Given : ⅓ keys appear only once +

⅓ keys appear twice (ie 2 records have the same key) +

⅓ keys appear thrice (ie 3 records have the same key)

Therefore no. of IOs while following Key K

= ⅓ (read only one block) + ⅓ (⅔ times only 1 + ⅓ times 2) +

⅓ (⅓ times only 1 + ⅔ times 2)

= ⅓ + ⅓ * 4/3 + ⅓*5/3

= 4/3

Exercise 13.1.4 : Answer

Since the location of the index block containing key K is known, although it is on disk we invest 1 IO to read this block.

Therefore average IOs
= 1 + 4/3 = 7/3

Given : ⅓ keys appear only once +
⅓ keys appear twice (ie 2 records have the same key) +
⅓ keys appear thrice (ie 3 records have the same key)

Therefore no. of IOs while following Key K
= ⅓ (read only one block) + ⅓ (⅔ times only 1 + ⅓ times 2) +
⅓ (⅓ times only 1 + ⅔ times 2)
= ⅓ + ⅓ * 4/3 + ⅓*5/3
= 4/3

⅓ chance we will find our 3 records in the same block, ⅔ times we will have to check the adjacent block

These blocks can hold 3 records, so there is a ⅔ chance we will find both our records in the same block.

Exercise Sheet 6, Q1

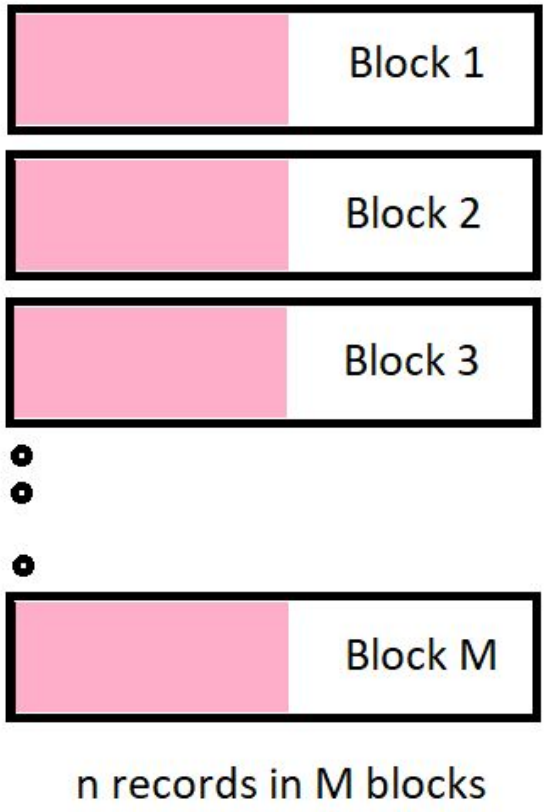
1) Suppose that we handle insertions into a data file of n records by creating overflow blocks as needed. Also, suppose that the data blocks are currently all half full. If we insert new records at random, how many records do we have to insert before the average number of data blocks (including overflow blocks if necessary) that we need to examine to find a record with a given key reaches 2?

Assume that on a lookup, we search the block pointed to by the index first, and only search overflow blocks, in order, until we find the record, which is definitely in one of the blocks of the chain.

Sheet 6, Q1 : Explanation-1/5

Initially, we have what's on the left. We have a certain number of primary blocks, with n records, and all the blocks are half full.

Number of blocks (including overflow blocks if necessary) that we need to examine to find a record with a given key : The index points to the blocks (not the records!) which means it a sparse index. In the current situation if we need to fetch a record, we will follow the pointer to the block and fetch the block (bring it to memory) and retrieve record by **examining only one block**. (No overflow blocks as of now)



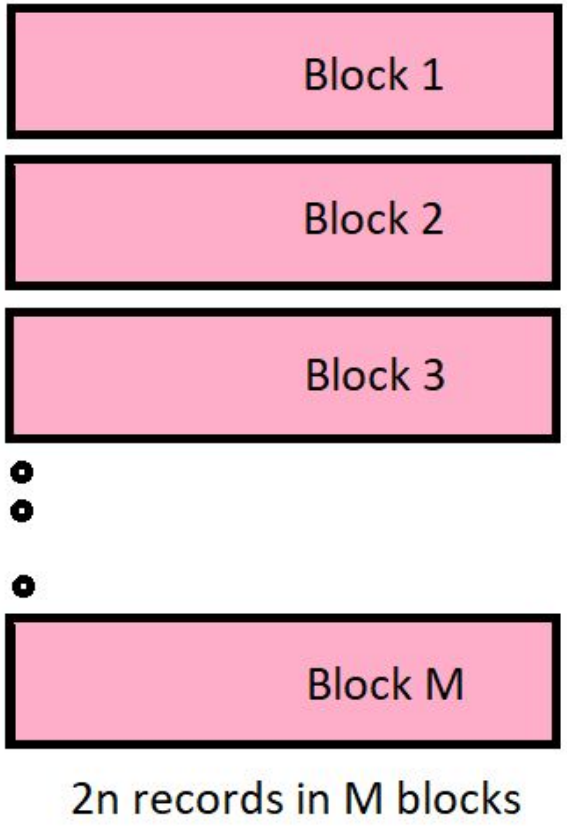
Sheet 6, Q1 : Explanation-2/5

Let's insert another n records, since we are inserting *at random* we can safely assume all the blocks to be full now and no overflow yet.

Number of blocks (including overflow blocks if necessary) that we need to examine to find a record with a given key :

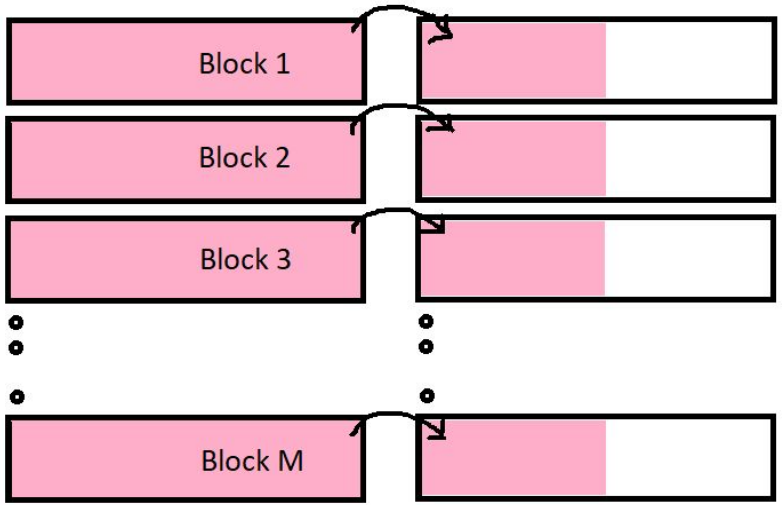
If we need to fetch a record, we will follow the pointer to the block and fetch the block (bring it to memory) and retrieve record by **examining only one block still**.

(Still no overflow blocks)



Sheet 6, Q1 : Explanation-3a/5

Let's insert another n records, since we are inserting *at random* we can assume all the blocks to have half full overflow blocks now. **Overflow blocks do not have entry in sparse indexes (they are a logical extension of the primary but still they are blocks and would cost IO if we need to fetch it.)** If we need to a fetch a record which is in an overflow block, we will have to fetch the primary block first, examine it, and then follow the pointer to the overflow block fetch the overflow block and retrieve that record.



2n records in M primary blocks and n in M overflow blocks

Number of blocks (including overflow blocks if necessary) that we need to examine to find a record with a given key :

Two scenarios :

1. Record is in primary block : 1 block needs to be examined and we will find the record
2. Record is in overflow block : First we examine primary block, do not find the record there, we follow the pointer to overflow block, examine overflow block, find the record. Therefore 2.

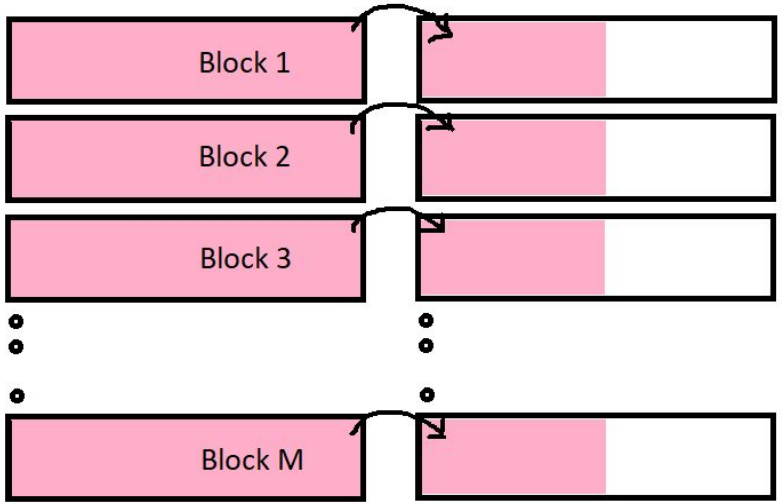
Sheet 6, Q1 : Explanation-3b/5

Since ⅔ records are in the primary block and ⅓ records in the overflow blocks, we will find the record within primary block ⅔ s of the time, and will examine a second block only ⅓ of the time, therefore average number of blocks we need to examine in this case would be:

= ⅔ (1) + ⅓ (2)

= 4/3

Still a bit far of from 2! Let's add more...

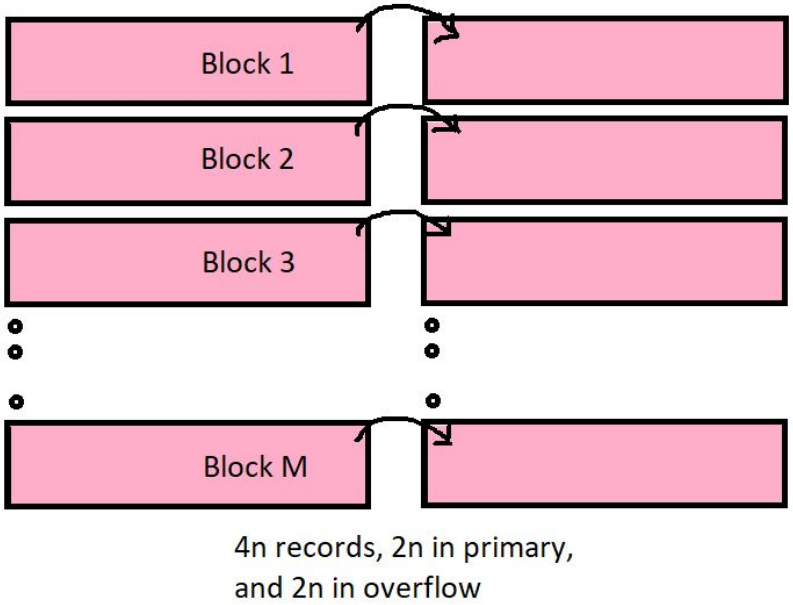


2n records in M primary blocks
and n in M overflow blocks

Sheet 6, Q1 : Explanation-4/5

Two scenarios :

1. Record is in primary block : 1 block needs to be examined and we will find the record
2. Record is in overflow block : First we examine primary block, do not find the record there, we follow the pointer to overflow block, examine overflow block, find the record. Therefore 2.



4n records, 2n in primary, and 2n in overflow

Number of blocks (including overflow blocks if necessary) that we need to examine to find a record with a given key :

Since ½ the records are in the primary block and the remaining ½ records in the overflow blocks, we will find the record within primary block half the time, and will examine a second (overflow) block the other half, therefore average number of blocks we need to examine in this case would be:

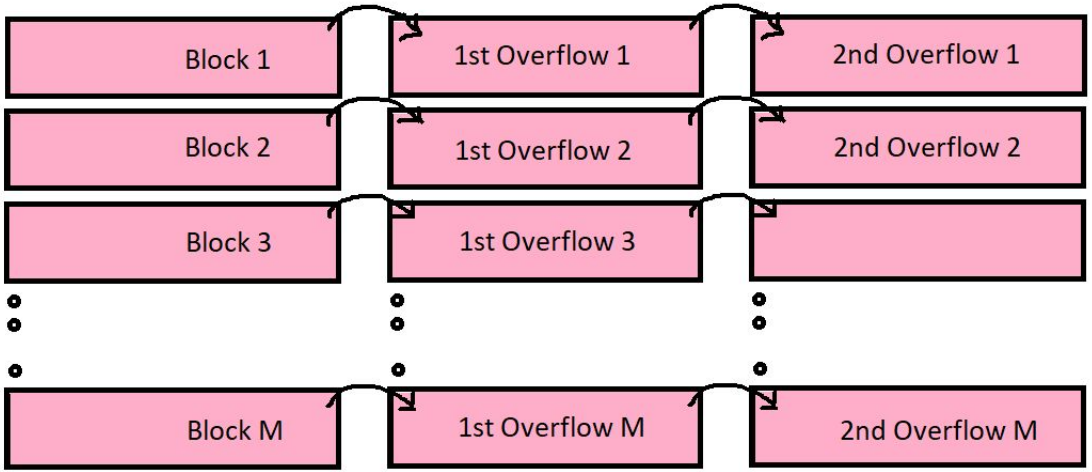
= ½ (1) + ½ (2)

= 3/2 = 1.5 **(Nearly there! Let's add more 2n more next)**

Sheet 6, Q1 : Explanation-5/5

Three scenarios :

1. Record is in primary block : 1 block needs to be examined and we will find the record
2. Record is in 1st overflow block : First we examine primary block, do not find the record there, we follow the pointer to overflow block, examine overflow block, find the record. Therefore 2.
3. Record is in 2nd overflow block. First we examine the primary, then the 1st overflow block and finally find our record in the second overflow block. Therefore 3.



6n records, 2n in primary, 2n if first overflow, 2n in second overflow

Number of blocks (including overflow blocks if necessary) that we need to examine to find a record with a given key :

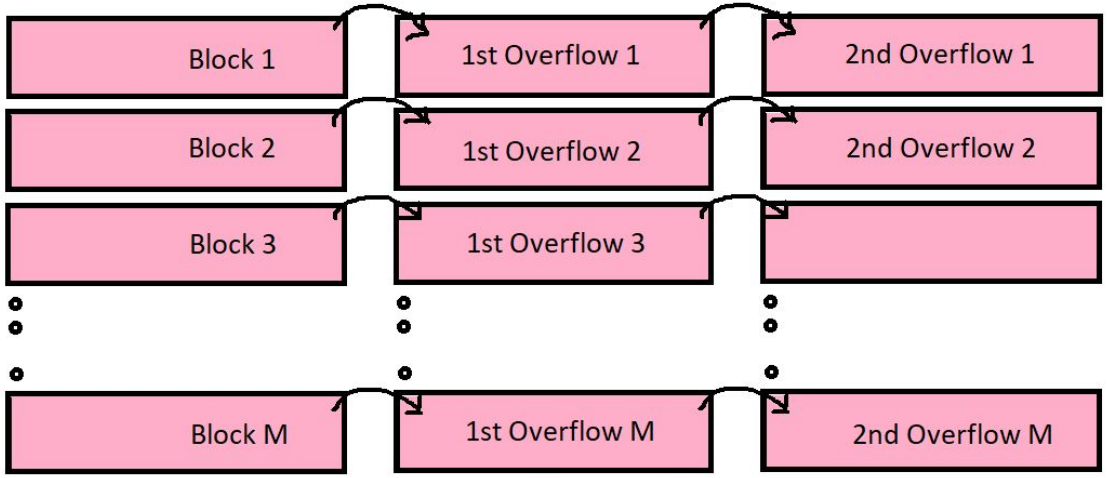
Since ⅓ of the records are in the primary block, ⅓ are in the first overflow blocks, and ⅓ in the second overflow blocks, we will find the record within primary block 1/3rd of the time, and will examine a second (overflow) block 1/3rd of the time and the second overflow block the remaining 1/3rd time, therefore average number of blocks we need to examine in this case would be:

= ⅓ (1) + ⅓ (2) + ⅓ (3)

= 2

Sheet 6, Q1 : Notes

Here we have followed a very step by step approach just to make things clear. You don't need write the answer in so many steps! It's just important to understand how blocks and therefore records are fetched and how sparse indexes work, and the effect of overflow blocks.



6n records, 2n in primary, 2n if first overflow, 2n in second overflow

As we saw, overflow blocks do not affect the sparse index at all, it still continues pointing to the same block. What is affected is how many blocks we need to examine to find our record.

Examining a block = reading a block (fetching it to main memory) = one I/O, therefore this question can be dealt with in terms of I/Os as well.

Exercise 13.2.5 New!

Exercise 13.2.5 : On the assumptions of Exercise Sheet 6 Q2 what is the average number of disk I/Os to find and retrieve the ten records with a given search-key value both with and without the bucket structure? Assume nothing is in memory to begin, but it is possible to locate index or bucket blocks without incurring additional I/O's beyond what is needed to retrieve these blocks into memory.

(Note from Q6: Blocks can hold either 3 records, 10 key-pointer pairs, or 50 pointers. We use the indirect bucket scheme. We have 3000 records and each search-key value appears in 10 records.)

Please try yourself first!

Exercise 13.2.5 Sol

Exercise 13.2.5 : On the assumptions of Exercise Sheet 6 Q2 what is the average number of disk I/Os to find and retrieve the ten records with a given search-key value both with and without the bucket structure? Assume nothing is in memory to begin, but it is possible to locate index or bucket blocks without incurring additional I/O's beyond what is needed to retrieve these blocks into memory.

(Note from Q6: Blocks can hold either 3 records, 10 key-pointer pairs, or 50 pointers. We use the indirect bucket scheme. We have 3000 records and each search-key value appears in 10 records.)

1. With Buckets : 1 (for index) + 1(for bucket) + 10 (for records)
2. Without Buckets : 1 (for index) + 10 (records)

   Since index is sorted and one block can hold 10 key-pointers and exactly ten records share the same key we will need only 1 I/O.

Modifications : Affect on Index 1

With time files will change. Records will be inserted, deleted, and sometimes updated. As a result, an organization like a sequential file will evolve so that what once fit in one block no longer does.

An index file is an example of a sequential file; the key-pointer pairs can be treated as records sorted by the value of the search key. Thus, the same strategies used to maintain data files in the face of modifications can be applied to its index file.

Modifications : Affect on Index 2

1. Overflow Blocks : Created if extra space is needed, or deleted if enough records are deleted that the space is no longer needed. **Overflow blocks do not have entries in a sparse index.** Rather, they should be considered as extensions of their primary block. For a spare index : Primary Block + its overflow blocks = 1 big primary block.
2. Instead of overflow blocks, we may be able to insert new blocks in the sequential order. **If we do, then the new block needs an entry in a sparse index**.
3. When there is no room to insert a tuple into a block, we can sometimes slide tuples to adjacent blocks. Conversely, if adjacent blocks grow too empty, they can be combined. Removing a block means we need to delete an entry from sparse index.

Modifications : Affect on Index 3

Action	Dense	Sparse
Create overflow empty block	none	none
Delete empty overflow block	none	none
Create empty seq. block	none	insert
Delete empty seq. block	none	delete
Insert record	insert	update?
Delete record	delete	update?
Slide record	update	update?

Overflow blocks have no effect on dense or sparse.

Creating/deleting primary blocks affects sparse index.

Creating/deleting/moving records affects dense index.

Only if the first record of the block is affected, in that case we have to update the sparse index.