## Chapter 3.1 The basics of Index Structure

***Definition:***

*Storage Structure:*

The Storage Structure is constructed by files, here the concept of file similar with the concept in Operation System.

*Example:*

One data file can be used to store one Relation. One data file can own one or multi - index files, each index file build the connection between the Query Key and Data Record. The pointer of the Query Key points to the record with the same property.

*Dense Index:*

The index could be Dense, which is to say that each record in data file will set it’s corresponding index in the index file.

*Sparse Index:*

The index could be Sparse, which is to say that some of the records in the index file will be present in the index file, normally each data block will be set one index in the index file.

*Main Index:*

The Main Index could be used to identify the location of the records in the data file.

*Auxiliary Index:*

The Auxiliary Index could not be used to identify the location of the records in the data file.

*(Normally, we will create the main index on the main key among Relation, and construct the Auxiliary Index on other attributes.)*

*Inverted Index:*

Given one or several key words, then through using Inverted Index, then we can get information from texts more efficiently.

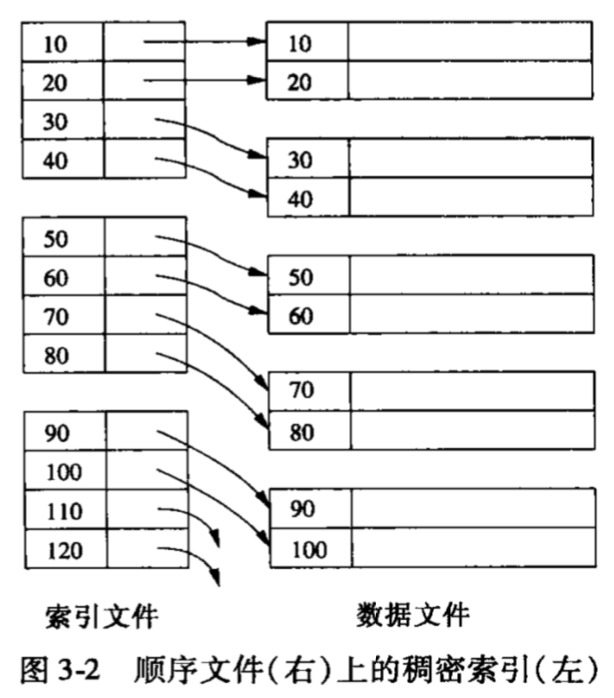
### Chapter 3.1.1 Sequential File

***Definition:***

Sequential File is the file that sorted all tuples of Relation according to the Main Key. The tuples are distributed among multi - blocks.

***Example:***

* In the image below, the sequential file has been displayed. Here assume that the Main Key is integer and there only exist two records in each storage block.
* The first block of the file just stored two records whose key value are 10 and 20. Also we use the continuous multiple as the Main Key.



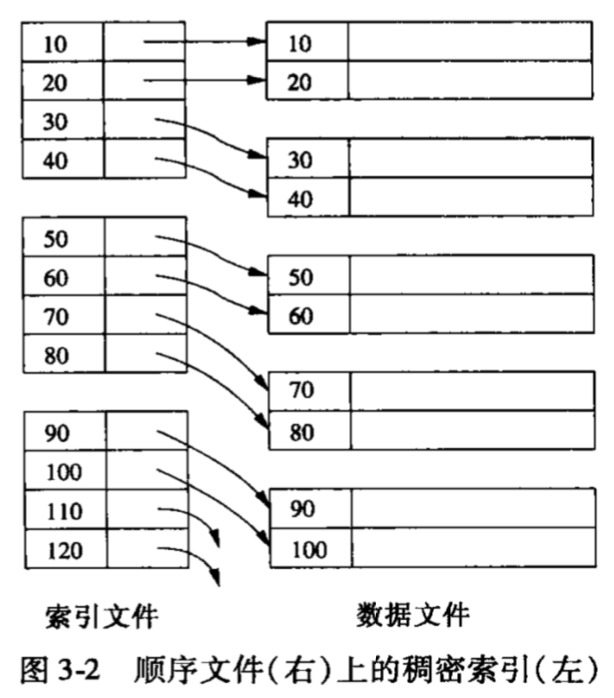
### Chapter 3.1.2 Dense Index

***Definition:***

* If the records are sorted, then here we can build Dense Index on records. Dense Index is such kind of storage block. Only save the record key and the pointers that point to records. The pointer is address. *(The Key Value Sequence of Dense Index is the same with the Sorted Sequence in file.)*
* Since the total size of Main Key and pointer are much less than the record, therefore we can assume that the blocks used for store index file is much smaller than store data file.
* When main memory can only be used to keep the index file, then it can increase the speed to query. Through using the index, then we can query only once to find the record with the given value.

***Example:***

Below is the Dense Index build on the Sequential File.



1. The first index block, four pointers point to the first four records.
2. The second index block, four pointers point to the next four records.

***Query based on Dense Index:***

Given the key value k of the record, then first search in the index block. After we find the K, then according to the corresponding pointer find the record from the data file.

Before we find the specific record, we need to search each storage block or almost half of the storage block.

***Advantage:***

* The number of index block is much less than data block.
* Since the Key are sorted, we can use Binary Search to find K. If there exist n index block, we only need to find log2(n).
* The index file may be small enough so that it can be saved in the main memory permanently. Then when we query, it only relates to the main memory access but no need other I/O operation.

### Chapter 3.1.3 Sparse Index

***Definition:***

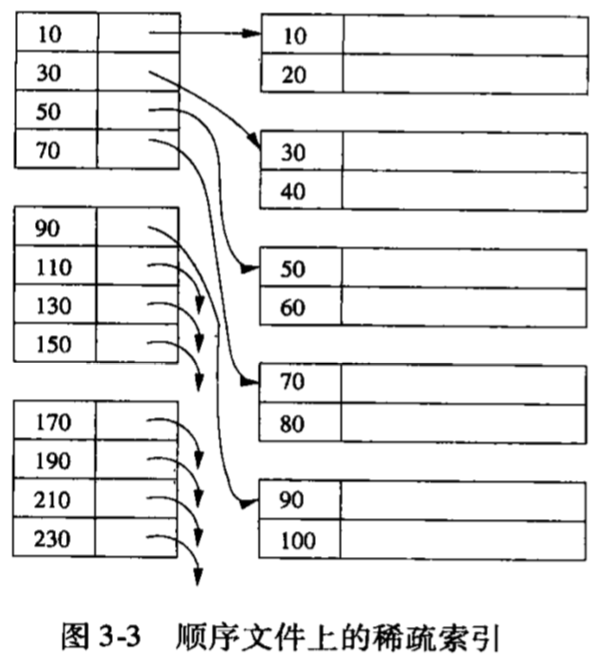
Sparse Index only sets one Key - Pointer pair for each storage block for the data file, it saves more Storage Space compared with Dense Index, but find the specific value of one record requires more time.

***Condition:***

Only when data file is sorted according to Query Key, then Sparse Index on the Query Key will be used. Dense Index can be used on any Query Key.

***Example:***

*Sparse Index set one Key - Pointer pair for each storage block. The Key - Value is the first record of each Data Block.*



Take the picture above as reference, the data file has been sorted, and the key value is the multiple of continuous 10, until a bigger data. We assume that each storage block can store 4 key - pointer pairs. In the first index file, there exist 4 key - pointer pairs, 10, 30, 50, 70. In the second index file, there exist 4 key - pointer pairs, they are 90, 110, 130, 150.

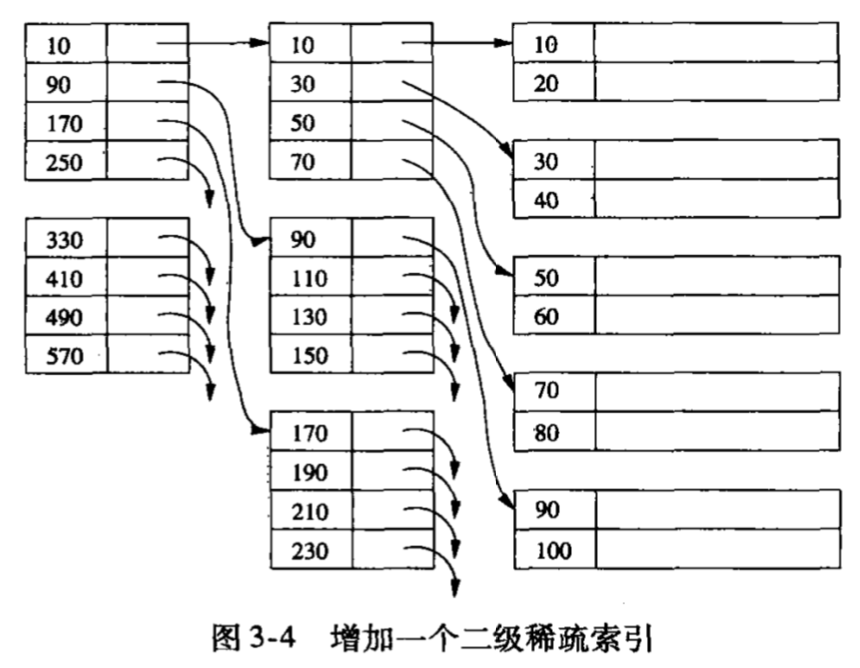
Given the Sparse Index, then we need to find the record with Query Value K, then we need to find the biggest Key that smaller than or equals to K in the Index file, this time we can use the Binary Search Method to locate the index, then we can find the corresponding data block according to its pointer.

After we locate the Storage Block, then we can search this Data Block to find the corresponding record with the Key Value K.

### Chapter 3.1.4 Multi - Level Index

***Principle:***

Index files probably occupy multi blocks, even we can locate the index block, and using Binary - Search Algorithm to find the required record. Through building index on index file, then we can make the first index level more efficient.



The example enlarges the second index level on the first index level. We need to pay attention that the second index file or the higher level must be sparse, since the second level index must have the same number key pointer.

### Chapter 3.1.5 Auxiliary Index

***Definition:***

*Auxiliary Index can be used for any index, such data structure can be used to help find the record with one or more fields.*

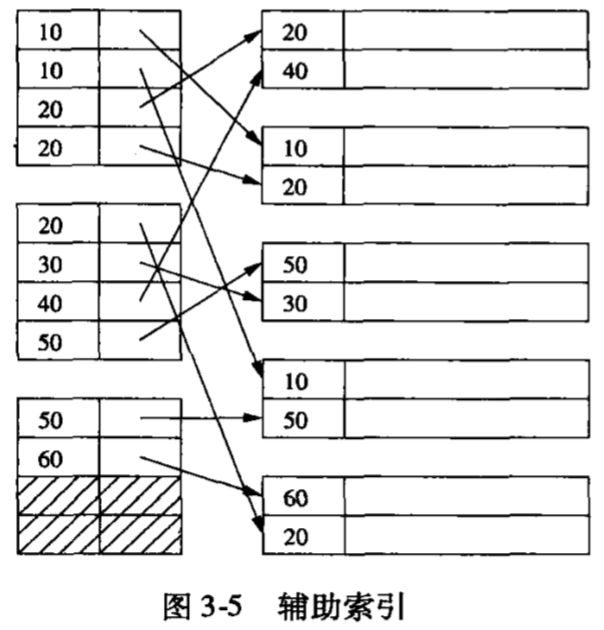
The Biggest Difference between *Auxiliary Index* and *Main Index* is that the Auxiliary Index *can not be used to decide the stored location of some fields.* But it can only helps us find the current location of record, while the location maybe decided by the Main Index that built on some other fields.

***Conclusion:***

*Auxiliary Index is always the Dense Index*, it is meaningless to talk about one Sparse Auxiliary Index. Since the Auxiliary Index can not influence the Storage Location of the record, and then we can not predict the location of the record that has not been recorded into the Auxiliary Index.

***Example:***

In the picture below depicts one classic Auxiliary Index, an it is just the same as the principle before, each data file stores two records. The record just shows the Search Key.



*The Key in the index file are normally sorted, then it will cause that the pointers will not point to the one or less several continuous storage but points to several different data blocks.*

For example, if we want to search for all records with the index key equals to 20, then we need to first locate two Dense Blocks and then go to three different Data Blocks. So when we want to find the same number records, then the use of Auxiliary Index will takes much more Disk I/O than Main Index.

### Chapter 3.1.6 The Usage of Auxiliary Index

***Usage:***

1. Additional Index in the Sequential File, just as introduced in Chapter 3.1.5
2. Main Index of some Data Structure, such as ‘heap’, in Heap, data records have no specific order. And the second common used data structured is Clustered File.

Assume that Relation R and S, the tuple in R and tuple in S have the N - 1 Relation. The former Structure is to store each tuple of Relation R with tuple in Relation S, while the other one is to store Relation R using Key.

***Example:***

Consider two standard Relation movie and studio:

Movie(title, year, length, genre, studioName, producerC#)

Studio(name, address, presC#)

The normal format of Query is:

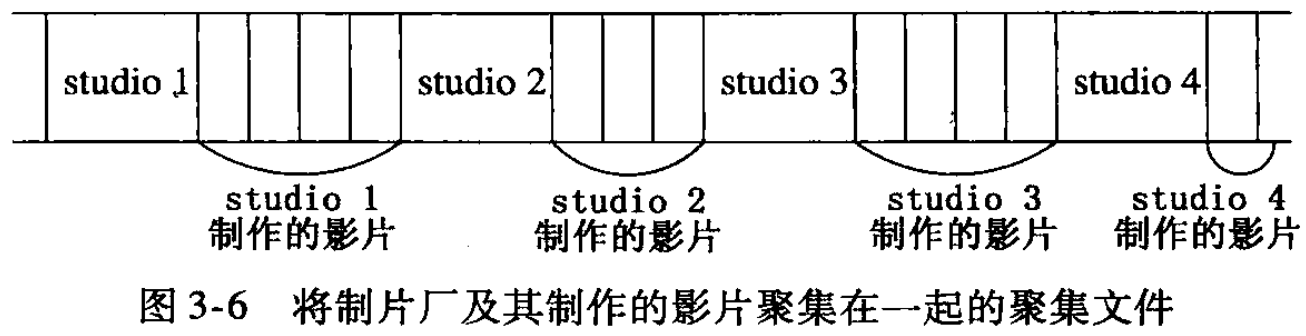
*SELECT title, year*

*FROM Movie, Studio*

*WHERE presC# = zzz AND Movie.studio.Name = Studio.name;*

Here, zzz represents the certification number of random Producer Manager, which is to say, for one known manager of a manufacture, we need to find all movies from this manufacture.

If we want to make sure that if this kind of Query is the classical Query, then we do not need to sort based on Key title and year, but to construct one Clustered File Structure for Relation Studio and Movie, just as the picture below. We just store all movie tuples after each tuple of Studio.



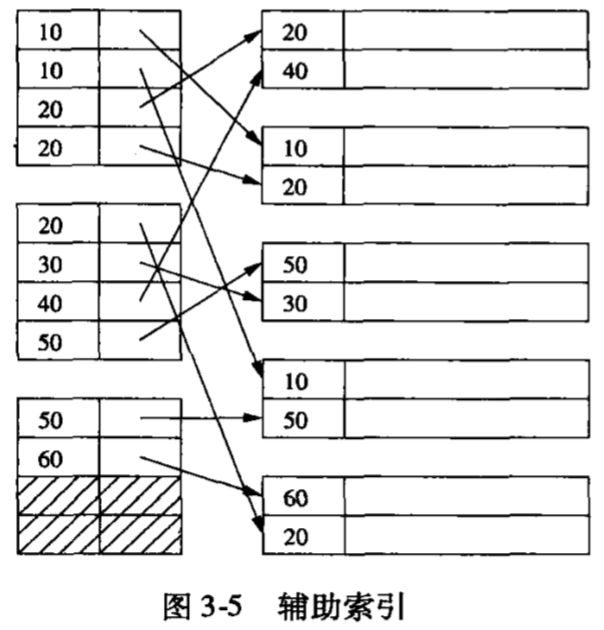
***Supplement:***

Here, if we construct the index on presC# for Relation Studio, then no matter what the zzz is, then we can locate all tuples in the manufacture as long as they satisfies the condition. Also, needs to mention that if the tuples in the Movie have the same name with the manufacture, then we can decrease the I/O times and then we can find all movies for this manufacture, since the required tuples have already followed the specific Studio. Although like this, then we can only build Auxiliary Index for random attributes on Relation Movie.

### Chapter 3.1.7 The Auxiliary Index’ Indirect

***Motivation:***

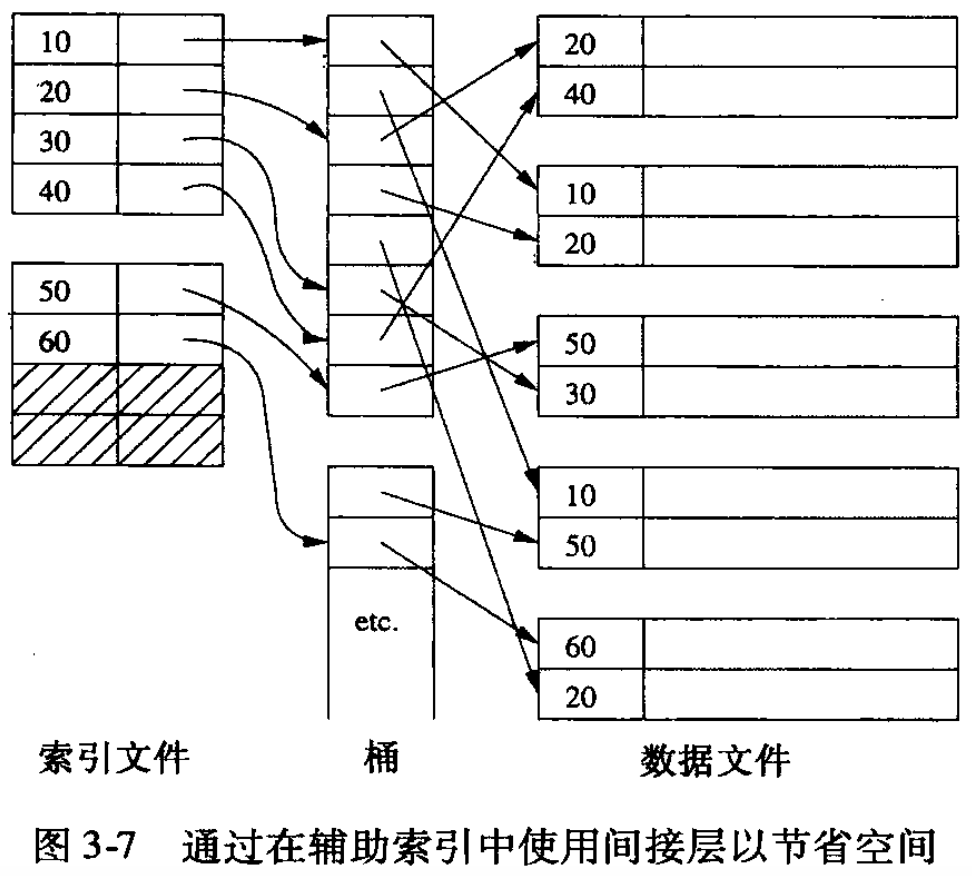
Tell from image 3-5, we find that there exist lots of space waste and sometimes the waste is extremely large. Assume that some Index value appear n times in the data file, then this index also needs to be written in the Index file for n times, if we can save Index for just one time, this would be better.



***Solution:***

We can use *Intermediate Level ‘Bucket’* to avoid key repetition, and it is between *Auxiliary Index File and Data File*. Just as the image, each Query Key K has one Key - Pointer Pair, and the pointer points to the Bucket File, and this file saves K bucket. *Start from the current position to the next position that this index points to, and these pointers point to all records which index value equals to K.*

***Example:***



We try to find from ‘Bucket’ File through the Index Key ‘50’. This pointer brings us to the last pointer of the first Block in the Bucket. We keep forward to query, and find the first pointer of the next block. Because the Index Value 60 points to the second pointer of the second Block in the Bucket, we stop searching.

***Advantage:***

Intermediate Level can be used to help solve answer some querys without visiting the data file records. *Especially when query multi-conditions, and each condition has one usable Auxiliary Index, then we can calculate the Intersection Collection in main memory to find out all pointers that satisfy the condition, and find out the record from Intersection Collection.* We can save I/O cost that just query partial conditions but not all conditions.

***Example:***

*Movie(title, year, length, genre, studioName, producerC#)*

Assume that we build intermediate auxiliary index on Relation studioName and year, and we need to execute Query below:

*SELECT title*

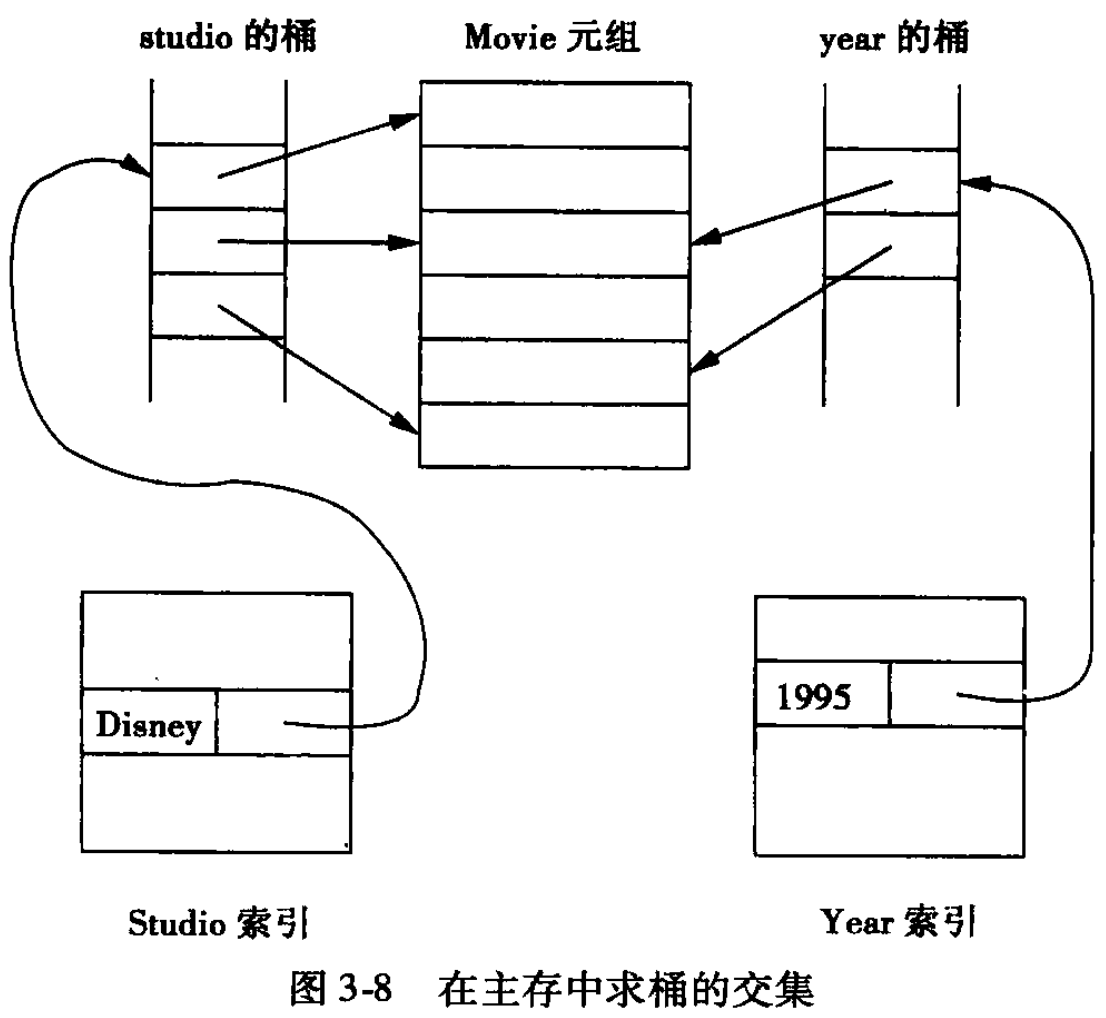
*FROM Movie*

*WHERE studioName = ‘Disney’ AND year = 2005*

Find all movies manufactured in year 2005 and studioName is Disney.

* By index studioName, find all Movie whose studioName equals to Disney.
* By year index, find all Movies which manufactured in 2005.

*Finally intersect two data collection, and return all movies manufactured in year 2005 and whose studioName equals to Disney.*

**

### Chapter 3.1.8 File Search and Inverted Index

***Background:***

For many years, information search in the File Store and File Query according to Key has been popular for many years.

As the appear of WWW and save files online possible, based on Key Words to index file has been one of the most difficulty problem.

***Principle:***

Although there has a lot of Document Query, but the most simple and normal one can be described as:

* *One Document can be described as Relation Doc tuple.* This Relation has a lot of attributes, each attribute corresponds to one word that may appear in the File. *Each attribute is bool which shows whether this word appears in the File or not.* Therefore, this Relation can be described as:

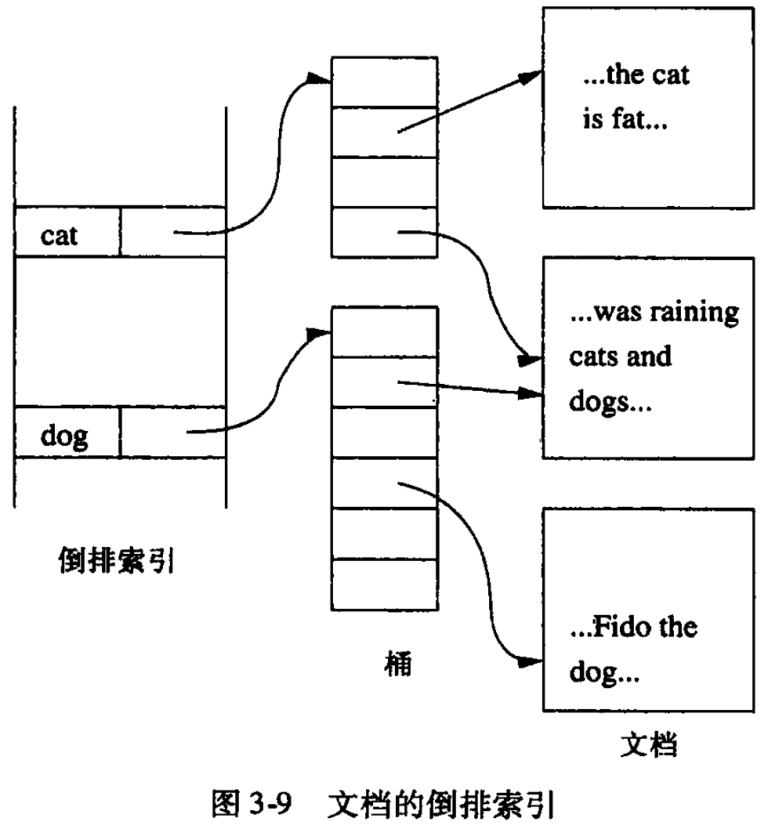
*Doc( hasCat, hasDog, ... )*

*Among relation, if hasCat equals to true when the word ‘cat’ appear at*  *least once in the word.*

* *Each attribute in Relation Doc build Auxiliary Index. But, we do not need to pay attention to the tuples whose value equals to FALSE*, conversely, the index can only bring us to the file where the attribute appears, which is to say, *the index only find the attribute whose value equals to TRUE*.
* We do not build one single index for each attribute, but just combine all indexes into one, which is called Inverted Index. This index uses Intermediate Bucket to improve the space utilize, which is discussed before.

***Example:***

Each file can be saved on one or multiple Disk Block. Inverted Index consists of a series of word - pointer pair, here word is actually the Query Key of the Index. Inverted Index is stored in the continuous Blocks.



Pointers point to the location of Bucket. The word ‘cat’ in the file, there has one pointer points to the Bucket File. The Bucket File includes all pointer heads that point to all files that contain ‘cat’. Similarly, the word ‘dog’ points to the pointer list, this pointer points to all files that include ‘dog’.

*Pointers in the Bucket File can be:*

1. *Pointers that point to File.*
2. *Pointer that points to the word that appear once. Under this situation, the pointer can point to the pair which consist by the first block and the times which represents the word appears in the File.*

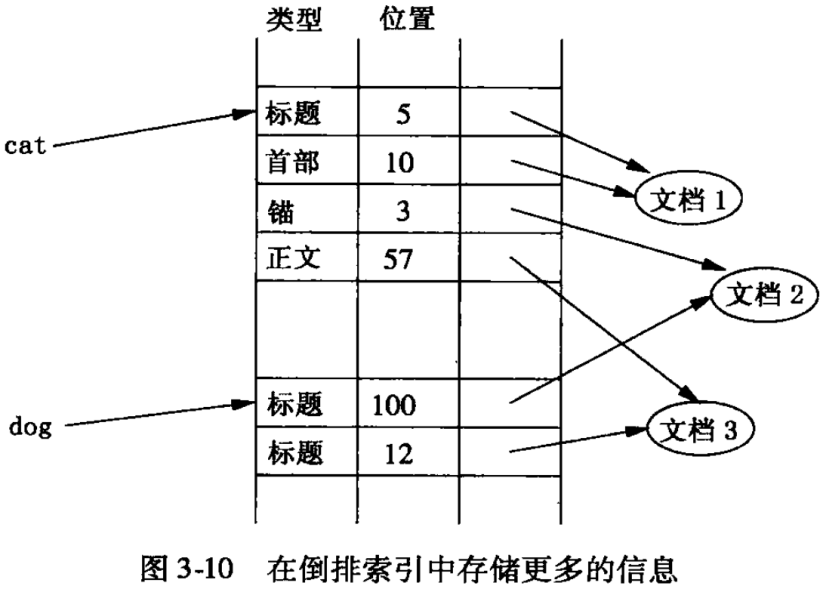
When we use the pointers point to multi - times that the word appears in the file, we may want to enlarge the thinking, make the bucket contain more information about the word. The Bucket itself would be the important structure consists of record collection. This apply on early period when differentiate whether the word appears in Title, Abstract or Main Body.

***Example:***

The bucket file shows that the word appears in HTML file.

* If appears the mark, then identify in the first line.
* The second and third line together consist the pointer and points to where the word appears.
* The third line identify the file, and the second line gives the location where the word appears.

*In the image, we can locate all ‘dog’ and ‘cat’. We can tell that ‘dog’ appears in the title, while ‘cat’ is also mentioned in the anchor - This anchor may be connect with the link that describes the document.*



For this question, we can answer the question through *Pointer Intersection*. Which is to say, the pointer ‘cat’ find all appearances of the word. We find all file pointers in the bucket file with the key word ‘cat’ and the type as anchor. Then we find the bucket with ‘dog’ key word, and select the file pointer while the type is ‘title’. *So if we intersect two pointer collection, then we can get the file which has ‘dog’ in title and ‘cat’ in anchor.*