Consider the following query:

SELECT * FROM R WHERE a = 10;

We would like an index structure on the key a that allows us to efficiently find all tuples in R with a = 10.

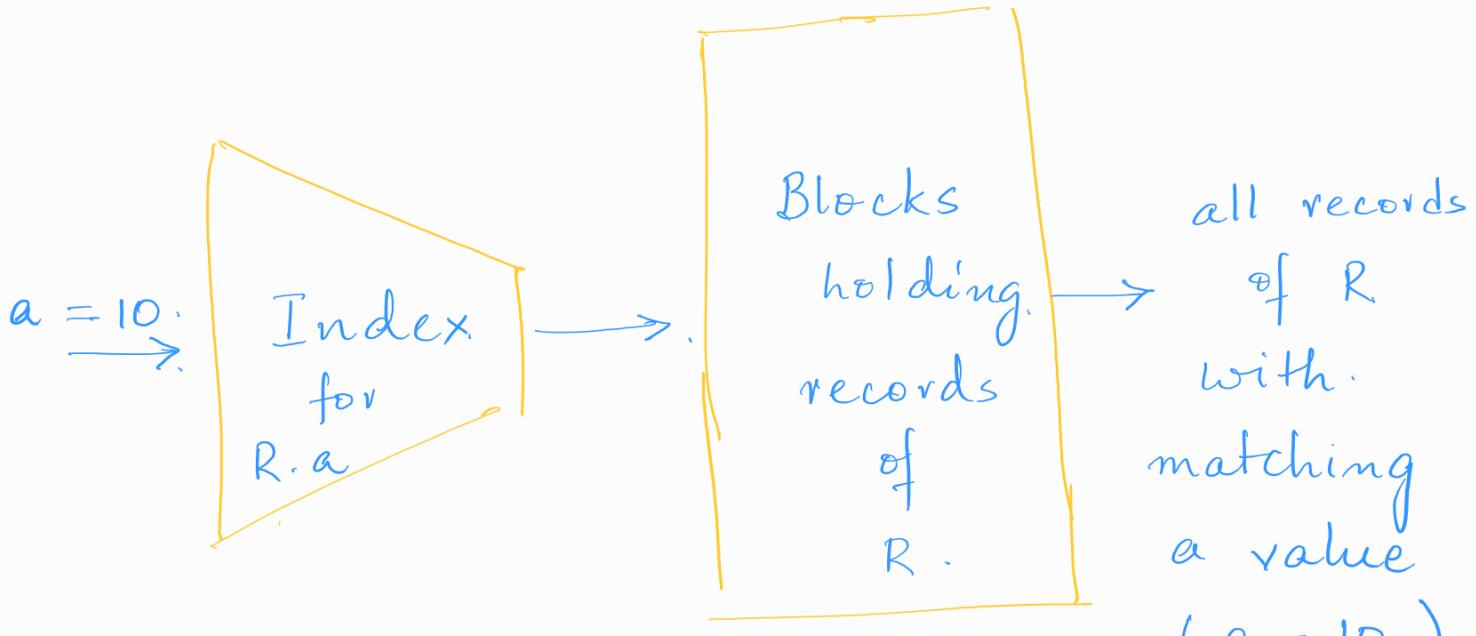


Fig:

Index on an attribute takes a value for some field (here it is 'a') and finds records with matching value (e.g. a=10).

INDEX STRUCTURE BASICS

Some notation: Storage structures consist of files.

1. Data file stores a relation.
2. Index file stores index entries for a relation. A relation

may have several index files.

There are the following "kinds" of indices:

1. Dense Indices, OR, (2.) Sparse indices

3. Primary Index, OR (4) Secondary index

We consider these structures briefly.

Sequential files:

A sequential file keeps the records of a relation sorted by primary key. The tuples are then placed in blocks, in this order.

E.g.

Table R.

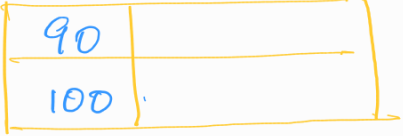
Block 1

Block 2

Keys are numbers.

Assumptions:

1. A block holds only 2 records. (Not typical).



Block 5.

2. Simplicity: Keys are multiples of 10.

Dense Indices

1. Table is sorted by a particular attribute(s).
2. A dense index (created on this attribute(s)) is another file, that holds each key value of a record and a pointer to that record.

Called "dense" because each record has exactly one corresponding index entry.

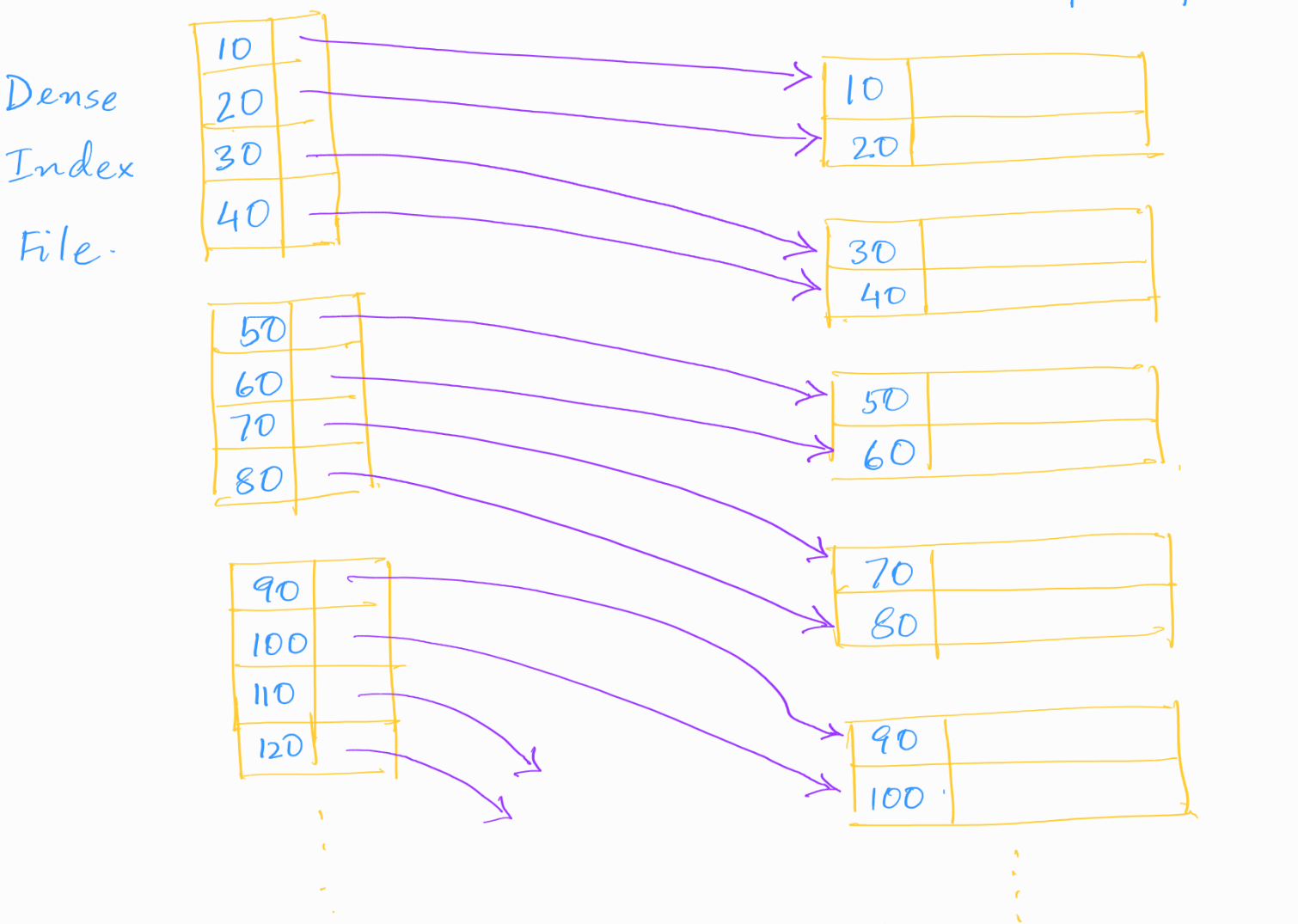


Fig: A dense index file (left) on a sequential data file (right).

In the example, an index block can store 4 (key, pointer) pairs. (atypical number).

Generally, (key, pointer) pair may take much less space than a record. So index blocks are dense and we expect to use many fewer index blocks than the data file itself.

A dense index supports queries. key = value (e.g. K = 60?). Given K, we search for index blocks for K, and when we find it, we follow the associated pointer to the record with key K.

Efficient Access is made using.

- Binary search to find K. If there are n index blocks, we only look at $\log_2 n$ of them.

Also n is usually << No. of data file blocks.

Note: 1. We have made an implicit assumption about some file header (INODE type) information telling us about the block addresses of the consecutive blocks

in the data/index file

2. "Dense" index means that there is an entry in the index file for every record in the file. File is supported by the key attribute of the index. But it is not necessary that the key attribute of the index is actually a primary or candidate key of the data file. Index key attribute for search may have duplicate values.

Sparse Index.

- Has one (key, pointer) pair per block of the data file.
- Assumes data file is sorted by the index key attribute.
- Uses less space than a dense index.
- Cannot be used for data file not sorted by the indexing key attr.

Example:

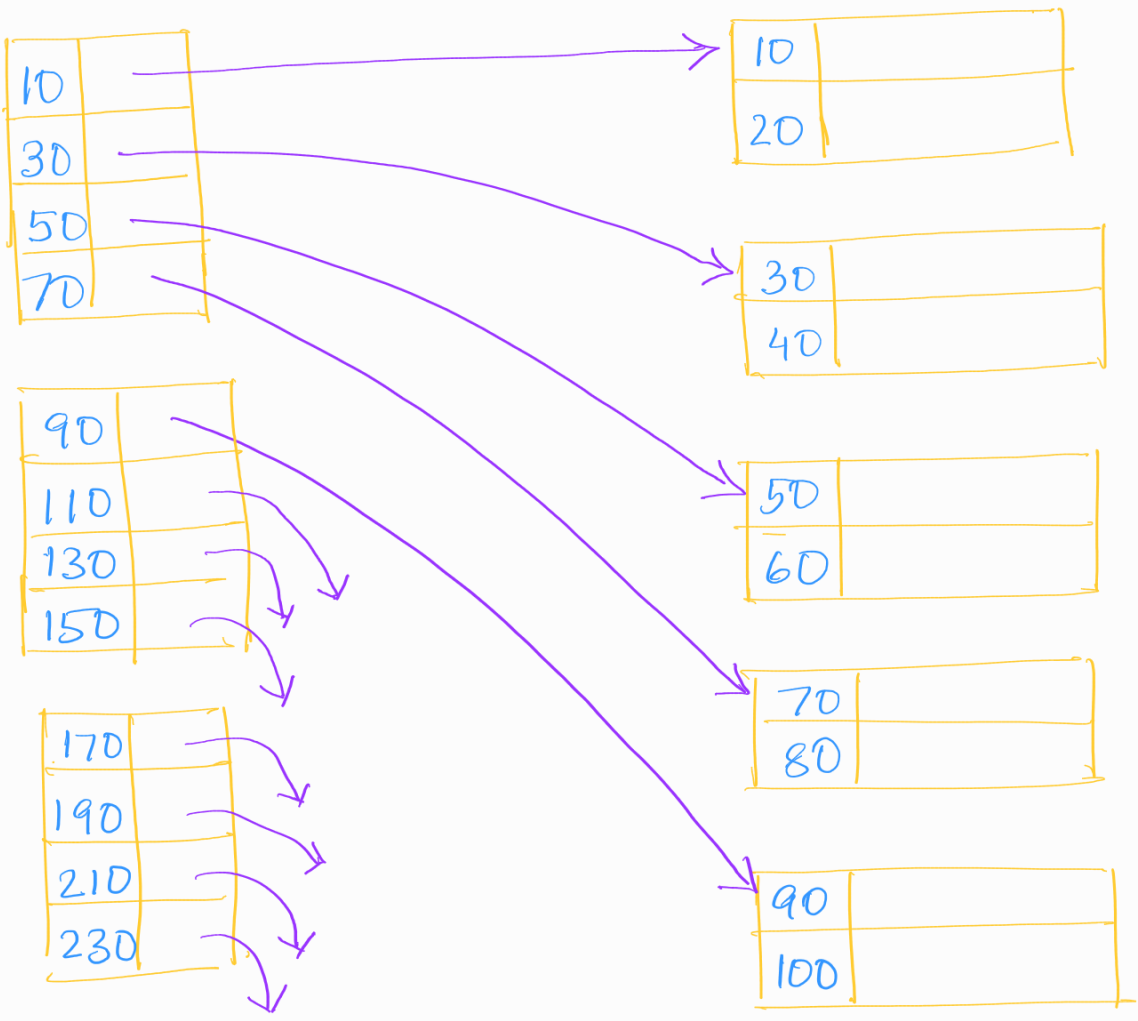


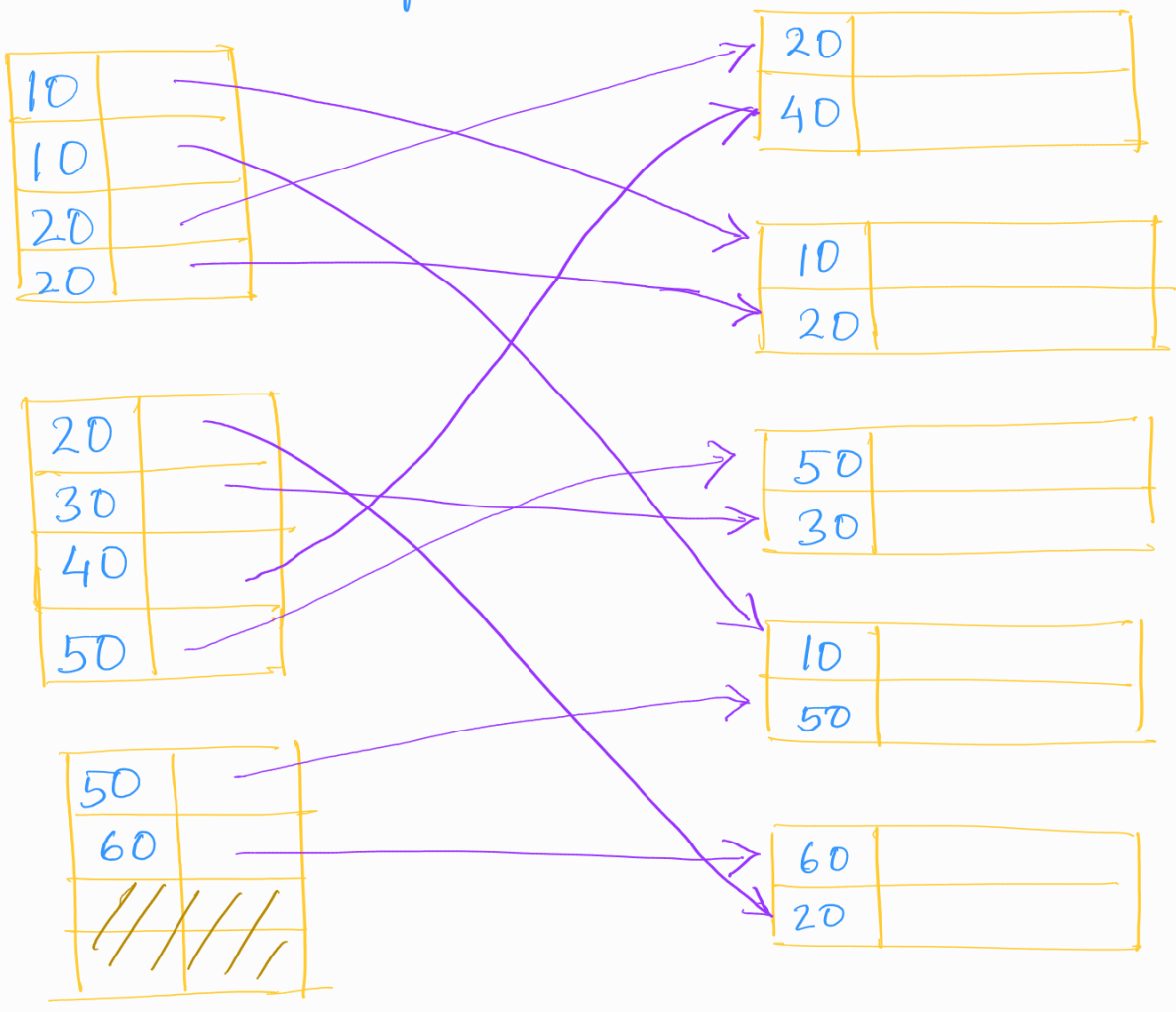
Fig: Sparse index (left) on a sequential file (right).

Search key value K: Search the index file for the largest key $\leq$ K. (Use binary search). Follow associated pointer to the data block. Now search this block for the record with key K.

2nd-level index: We can build a sparse index as an index of a first-level sparse or dense index

on some key attribute. A bit more systematic solution is B^+-trees.

SECONDARY INDEX: Serves the purpose of any index, but, it does not determine placement of records in the data file.



1. Data file has duplicate key values and is not sorted by it.
2. Secondary index is dense: every key value has an entry. E.g. key 20 appears thrice; so index file has 3 entries corresponding to it. Keys in index file are sorted.

3. Pointers in one index block can go (zig zag) to many different blocks.

4. To retrieve all records with search key 20, we have to look at 2 index blocks and follow 3 pointers to data blocks

Some uses of Secondary Indices.

1. Suppose the data file is a "heap structure", i.e., records are kept in no particular order. Secondary indices can be built on primary key, and on other attributes.

2. Computing using equality / non-equality of pointers.

E.g. Consider the StarsIn table

StarsIn (starName, title, role)

Suppose we have a secondary index on both starName and role.

Consider the query

```
select   title
from     StarsIn
where    starName = 'SHAHRUKH KHAN'
   and   role = 'VILLAIN'
```

Suppose that by accessing StarsIn by secondary index starName, we get a list of pointers to movies for starName = 'SHAHRUKH KHAN'. Likewise, by accessing the secondary index by role and setting role = 'VILLAIN', we get a list of pointers to movies with this role.

We now simply take the intersection of these two pointer lists. From the intersection pointer values, we follow to the "movies" record and retrieve the title. Since checking equality is done over pointers rather than accessing the Movies relation, costs are reduced.

DOCUMENT RETRIEVAL and INVERTED INDEXES.

This discussion is inspired from the information-retrieval community, for designing indexes for efficient retrieval of documents based on (say) English keywords. The document repository is (say) a database of online WWW html documents.

Modeling a document:

1. A document is thought of as a tuple in a relation Doc.
2. The relation has many attributes, e.g. all keywords in the English language.

E.g. consider two keywords

cat and dog among the millions of key words (attributes).

3. For a particular web document each entry is boolean, 0 or 1.

The entry corresponding to a document for attribute cat is 1 if the keyword cat occurs in the document and is 0 otherwise.

like-wise, the entry for this document under attribute dog is 1 iff the keyword dog appears in the document and is 0 otherwise.

3. Keep a secondary index on each of the attributes of Doc; but, we only index those documents for which the keyword is present, i.e., the entry is 1.

4. Instead of creating separate indices for each attribute, we combine them into one, called inverted index.

A slight variation called indirect buckets for space efficiency is used.

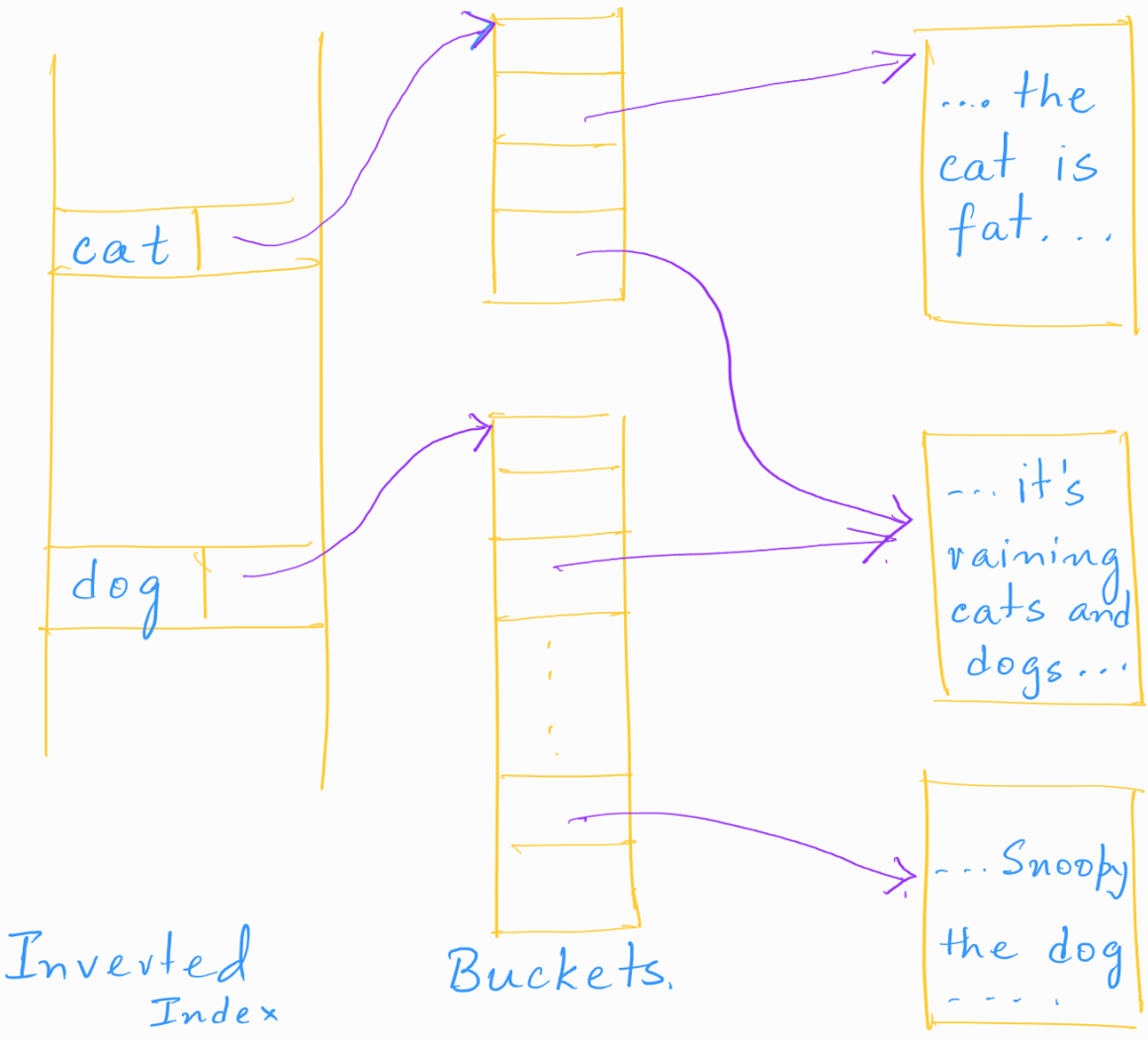


Fig: An inverted index on documents.

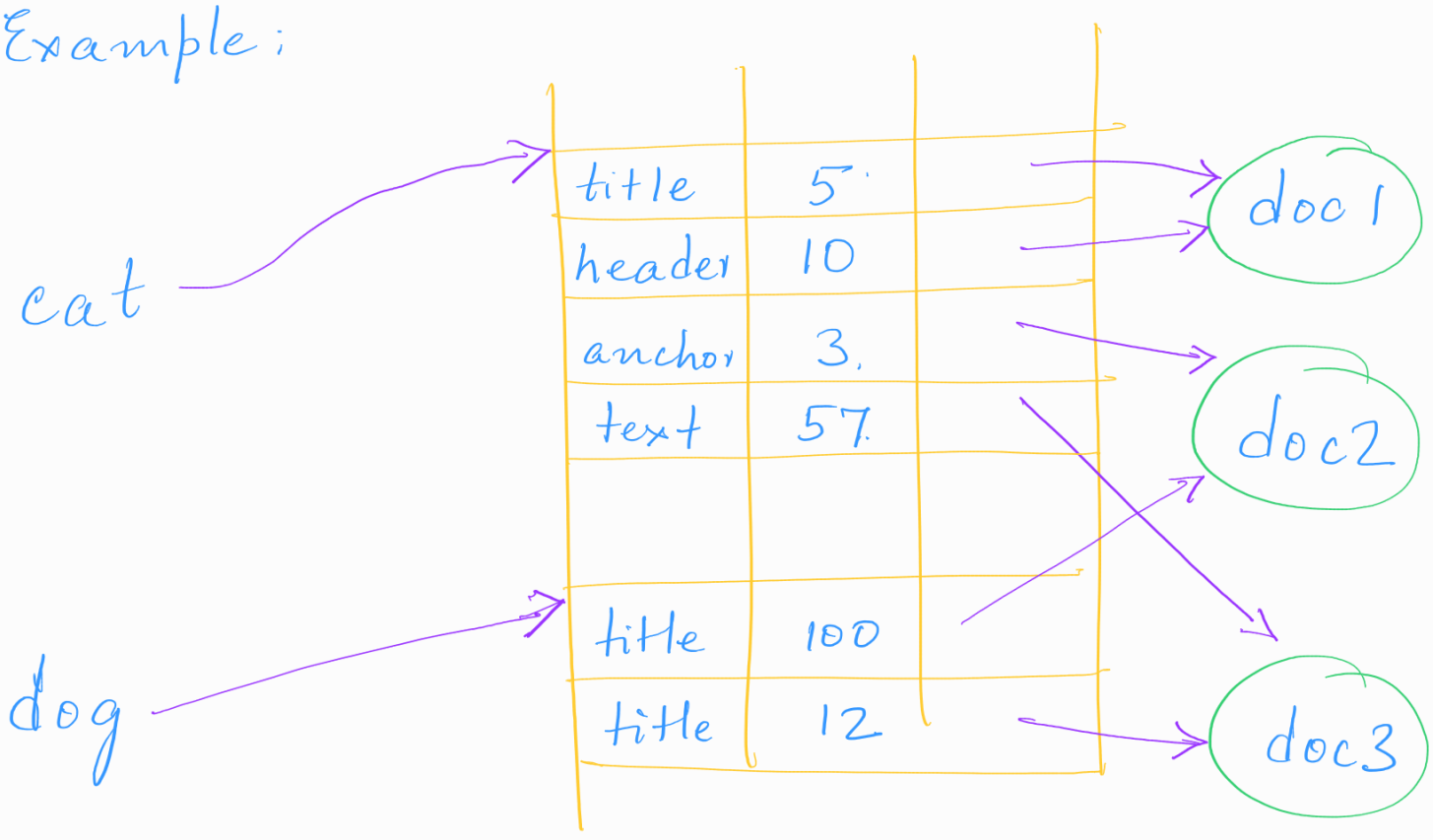
- Inverted index is a set of (keyword, pointer) pair, the pointer points to a sequence of blocks (buckets) containing pointers to the documents containing this keyword. Inverted index is kept in a seq. of blocks (like any other index).

- Pointers refer to positions in a "bucket" file.

Pointers in the bucket file can be:

1. Pointers to the document itself.
2. More refined, pointers to each occurrence of the word. Here pointer ≡ (addr. of first block of doc, integer indicating number of word in the doc).
3. The "refined" info is kept in a bucket entry. Eg.

Example:



- First col in buckets table indicates "type" of occurrence.
- The (second, third) pairs gives a pointer

to the occurrence.

This data structure can be used to answer many queries without further examining the documents. For e.g. suppose we want to find documents about dogs that compare them with cats!

A possible rationale (hint):
Find documents that

a) mention dogs in title,

b). Mention cats in an anchor

Answer the query by intersecting pointers.

1. Follow pointer associated with cat to find its occurrences.
2. Select from bucket file the pointers to documents corresponding to occurrences of "cat" where the type is "anchor".
3. Then select bucket entries for "dog" and select from them the document pointers with type "title".
4. Finally, intersect the two sets of pointers. This gives the documents.