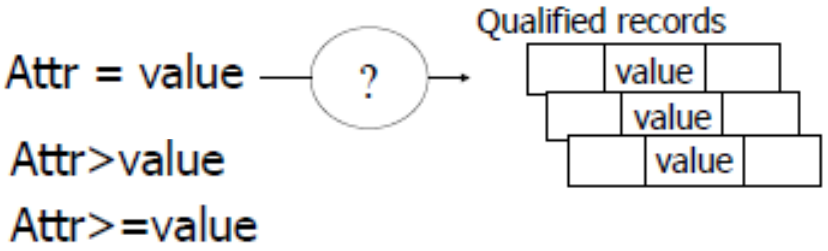
Indexing

Chapter 14, Ulman, 2nd Edition

Indexes (or Indices)

- Data Structures used for quickly locating tuples that meet a specific type of condition
  - – Equality condition: Find Movie tuples where Director=X
  - – Range conditions: Find Employee tuples where Salary>40 AND Salary<50
- Contains (key, value) pairs:
  - – The key k = an attribute value
  - – The value k* = one of:
    - pointer to the record *secondary index*
    - or the record itself *primary index*
- Any subset of the fields of a relation can be the Search key for an index on the relation can be any subset of the fields of a relation and different from '*key*' (minimal set of fields that uniquely identify a record in a relation).
- Many types of indexes. Evaluate them on:
  - – Access time
  - – Insertion/Deletion time
  - – Condition types
  - – Disk Space needed



File Organizations

- Heap files (random order)
- Sorted Files
- Indexes Trees
- Hash tables.
  - Like sorted files, they speed up searches for a subset of records, based on values in certain (“search key”) fields
  - Updates are much faster than in sorted files.

Index Classification

- Clustered/unclustered
  - – Clustered = records close in index are close in data
  - – Unclustered = records close in index may be far in data
- • Primary/secondary
    - Primary = is over attributes that include the primary key
    - Secondary = otherwise
- A file can be clustered on at most one search key
- Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!

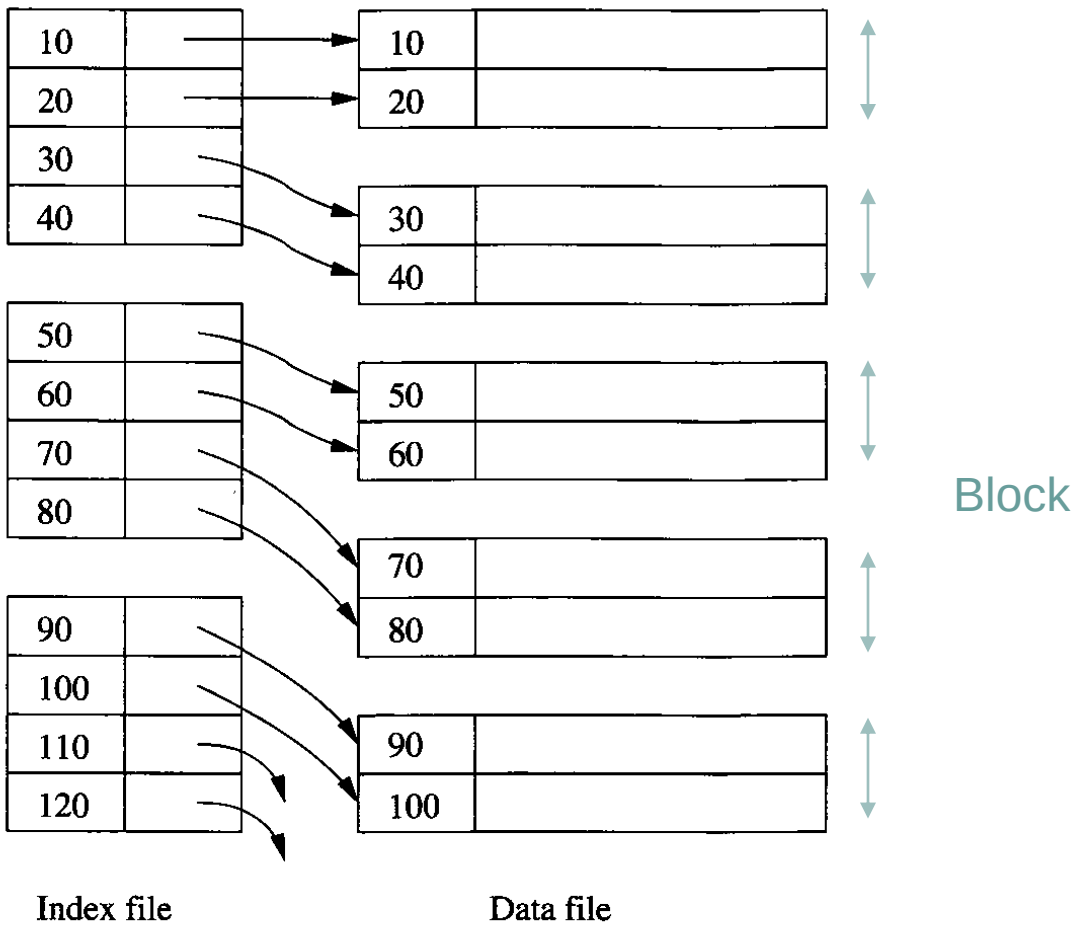


Figure 14.2: A dense index (left) on a sequential data file (right)

Sequential Files

- A *sequential file* is created by sorting the tuples of a relation by their primary key. The tuples are then distributed among blocks, in this order.
- Index file needs much fewer blocks than the data file, hence easier to fit in memory
- It is common to leave some space in each block, else insertion of new samples need to handled by overflow

Dense Index

- A **dense index** is a sequence of blocks holding only the keys of the records and pointers to the records themselves
- Index blocks of the dense index maintain these keys in the same sorted order as in the file itself.
- When is this indexing useful?
  - The index is especially advantageous when it, but not the data file, can fit in main memory.
- Whats the search time?
  - Following binary search only log(n) of n blocks need to be accessed to find a given key K
  - Then, by using the index, we can find any record given its search key, with only one disk I/O per lookup.
  - Given a key-value K, we search for index blocks for K
- create index DN_indexB on department(building);
- cluster department using DN_indexB;

Class Activity

- What dense index could possibly help?
  - Number of index blocks are small compared to the number of data blocks
- When not to use index?
  - Small table
  - Attribute(s), in which index is defined is frequently updated
  - Large number of NULL
  - Not frequently used by the users in search
  - …

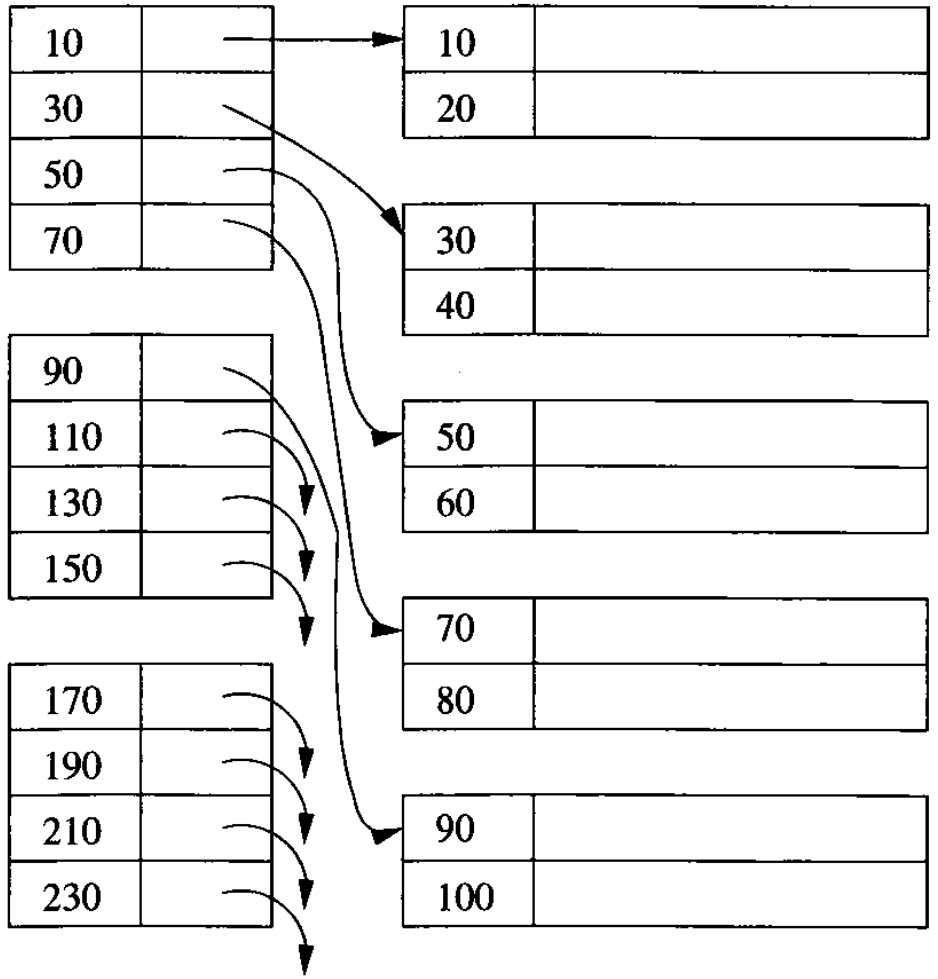


Figure 14.3: A sparse index on a sequential file

Sparse Index

- Typically, only one key per data block
- Searching: Find the index record with largest value that is less or equal to the value we are looking
- What is its advantage over dense index?
- It thus uses less space than a dense index, at the expense of somewhat more time to find a record given its key.
- When should you use it?
- **only use a sparse index if the data file is sorted by the search key, while a dense index can be used for any search key.**

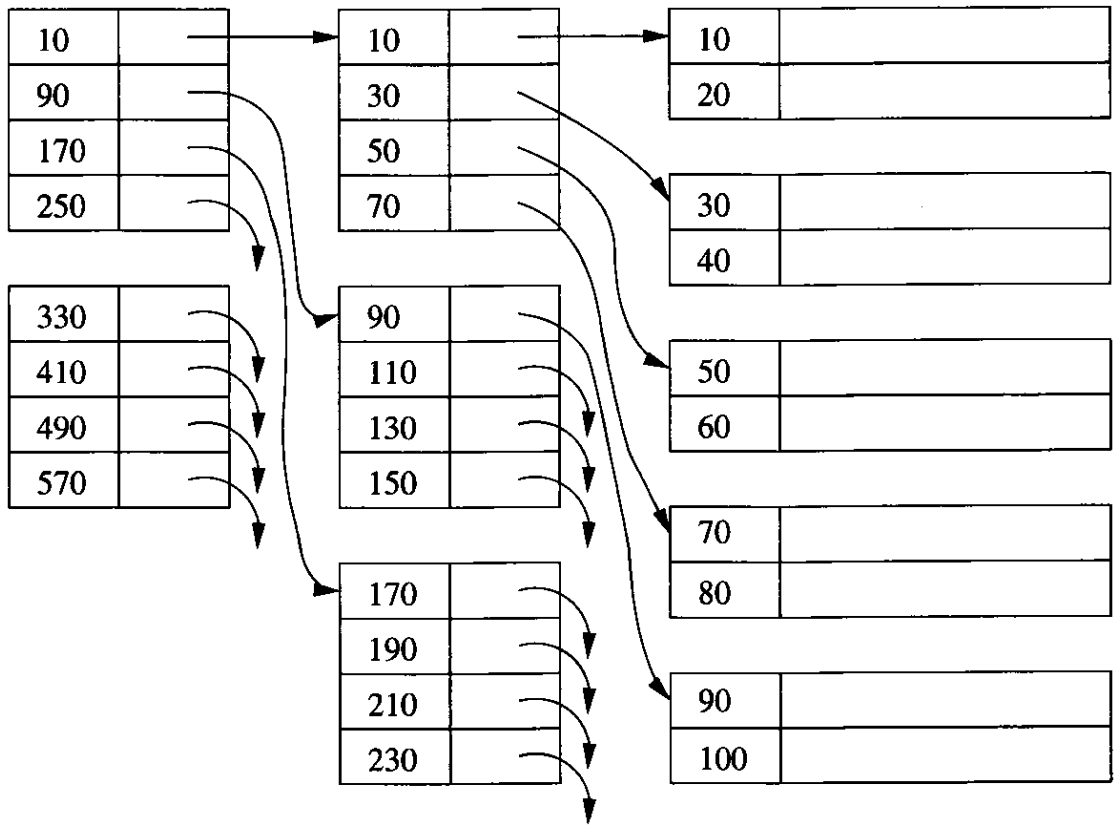


Figure 14.4: Adding a second level of sparse index

Multiple Levels of Index

- An index file can cover many blocks.
  - Even if we use binary search to find the desired index entry, we still may need to do many disk I/O 's to get to the record we want.
- An index on the index, we can make the use of the first level of index more efficient.
- But do you think it would always work?
- This idea has its limits, and we prefer the B-tree structure, to be discussed a bit later in this discussion
- In the example, we see the second level is sparse index, can we have multiple layers of dense index instead to enhance search efficiency?

Secondary Index

- It serves the purpose of any index
- It is a data structure that facilitates finding records given a value for one or more fields.
- Secondary index does not determine the placement of records in the data file. Rather, the secondary index tells us the current locations of records; that location may have been decided by a primary index on some other field.
- Applications:
  - Clustered file

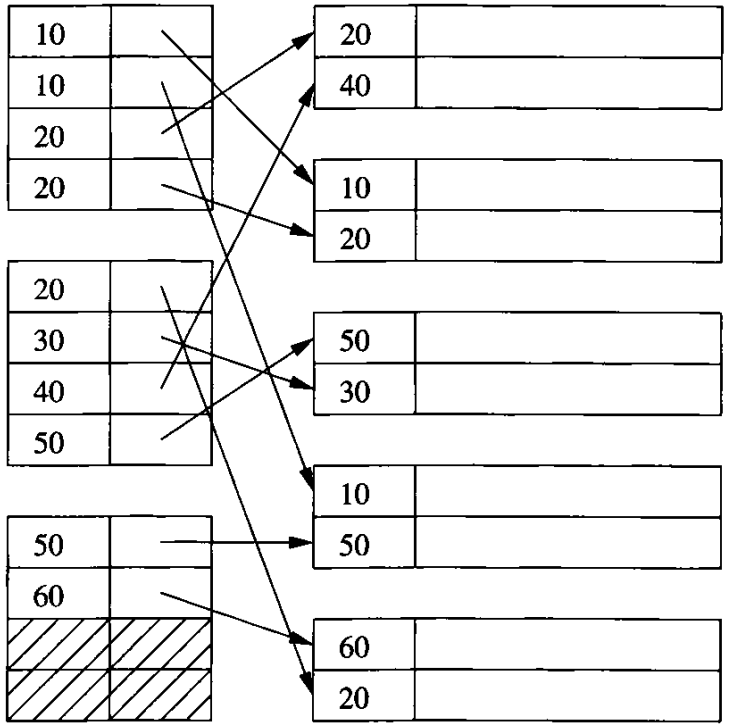


Figure 14.5: A secondary index

Applications of Secondary Index

- Suppose there are relations *R* and *S,* with a many-one relationship from the tuples of *R* to tuples of *S.* It may make sense to store each tuple of *R* with the tuple of *S* to which it is related, rather than according to the primary key of *R.*
- Example:

  Movie (title, year, length, genre, studioName, producerC#)
  Studio(name, address, presC#)

  select *title, year*
  from *Movie, Studio*
  where *presC# == zzz* **AND** *Movie.studioName=Studio.name*;

  What is the primary key here? Would search using primary key be useful here?
  Search with the primary key title and year are not preferable here

  Cluster movie using studioName;

Example

- We can create a clustered file structure for both relations that has the below configuration
- Then if we create an index for Studio with search key presC#, then whatever the value of *zzz* is, we can quickly find the tuple for the proper studio.

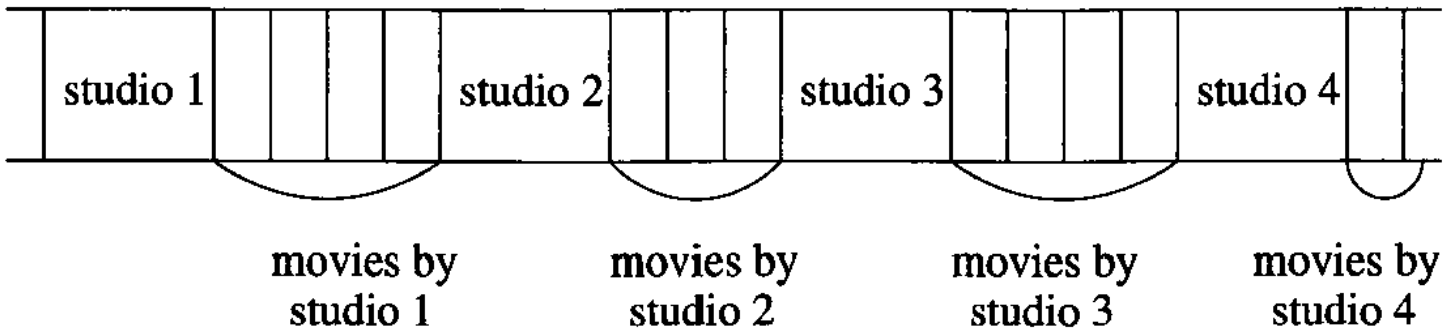


Figure 14.6: A clustered file with each studio clustered with the movies made by that studio

Indirection in Secondary Index

What happens when some keys are repeated?

Try to search for key value 10 and 50

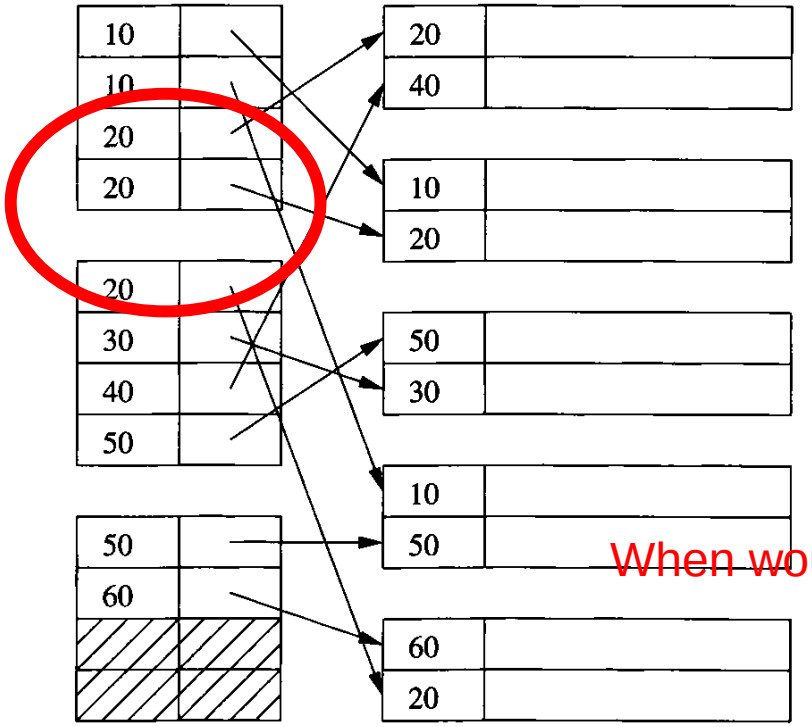


Figure 14.5: A secondary index

When would this approach be helpful?

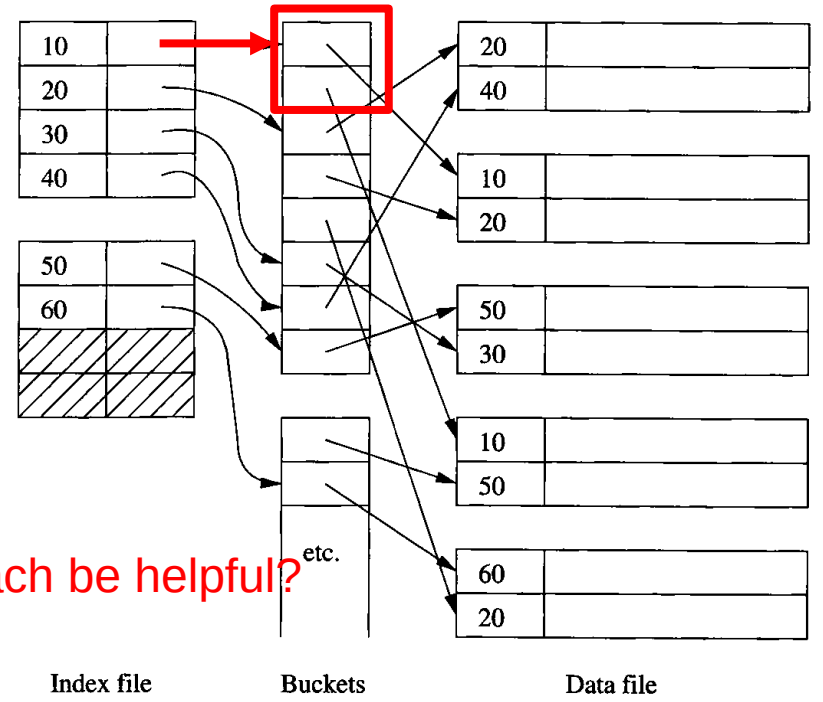


Figure 14.7: Saving space by using indirection in a secondary index

Useful as long as the search key value is larger than the pointers replacing their repeated entries and you actually have repetitions in the table on this search key value

Indirection in Secondary Indexes

- Several conditions to a query, and each condition has a secondary index to help it
- Find the bucket pointers that satisfy all the conditions by intersecting sets of pointers in memory, and retrieving only the records pointed to by the surviving pointers.
- Save the I/O cost of retrieving records that satisfy some, but not all, of the conditions.

select *title*
from *Movie*
where *year == 2005* **AND**
Movie.studioName=''Disney;

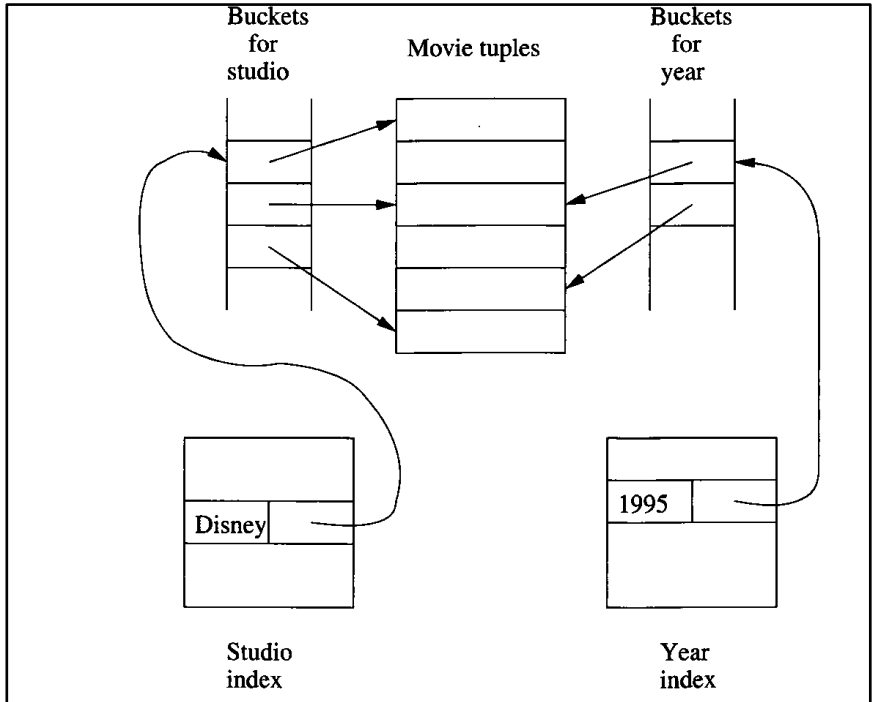
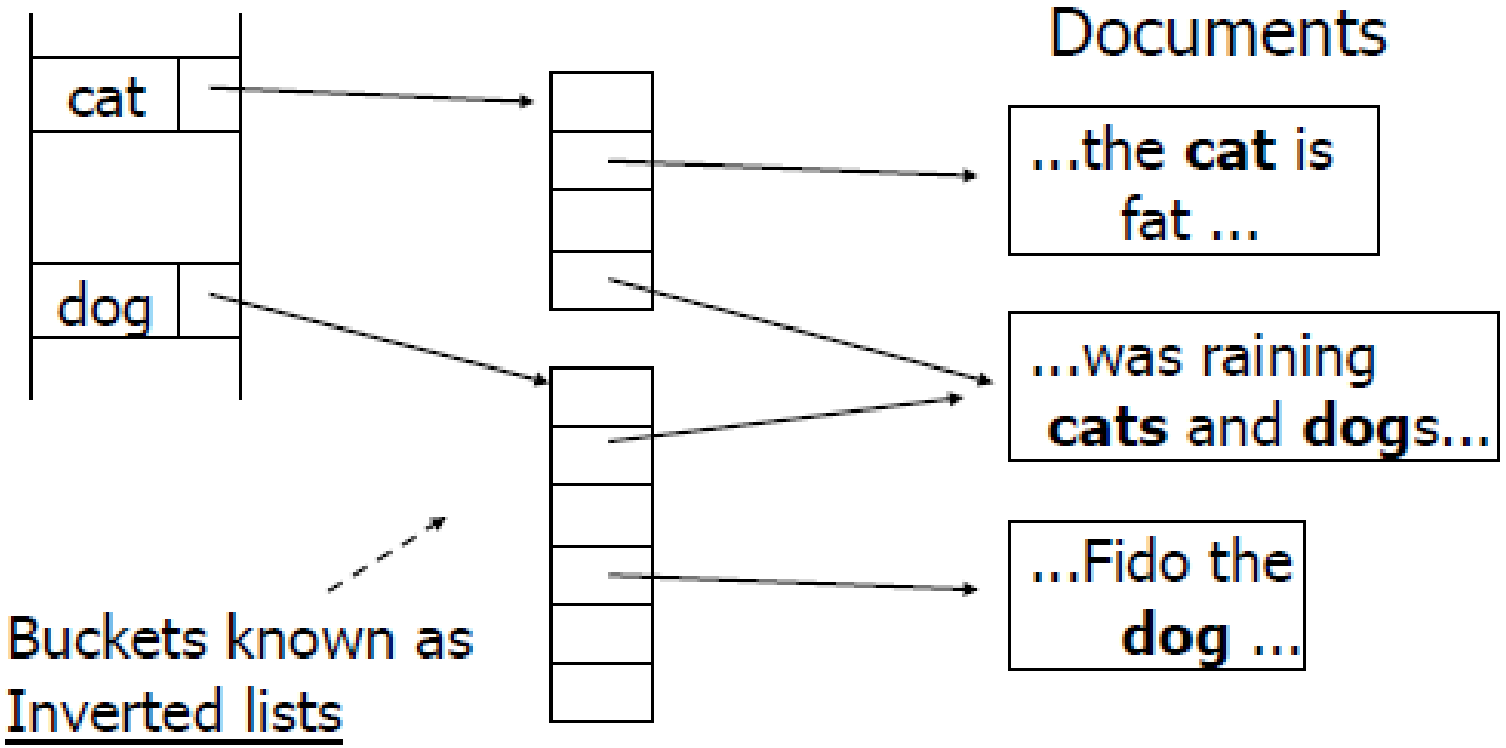


Figure 14.8: Intersecting buckets in main memory

Document Retrieval and Inverted Indexes



Example

- The example shows a bucket file that has been used to indicate occurrences of words in HTML docs.
- The first column indicates the type of occurrence, i.e., its marking, if any.
- The second and third columns are together the pointer to the occurrence.
  - The third column indicates the document, and the second column gives the number of the word in the document.

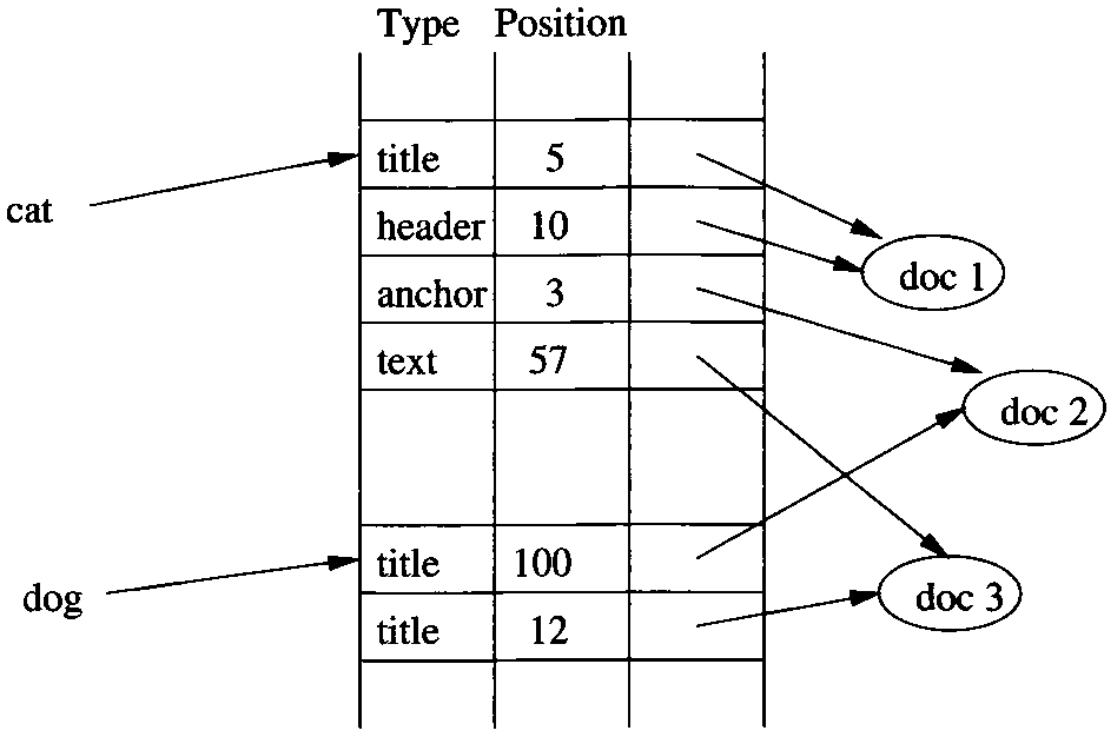


Figure 14.10: Storing more information in the inverted index