Index Structures

Overview

Index Structures are supplemental data structures created and maintained by the DBMS in order to speed up query processing and data management. Index structures are stored on disk in exactly the same manner as the relational data: each index structure is stored in a single file, the information there is broken into disk blocks.

We will consider the following categories of index structures:

- Indexes on Sequential Files. Sequential files allow for efficient access to data only if there is a efficient way to determine where the data is stored given a key. Index structures in this category address this issue. Two classes of indexes, *dense* and *sparse* are considered.
- Secondary Indexes. Index structures for non-search key attributes in sequential files and/or for heap files.
- B-trees. A more advanced way to organize indexing.
- Hash tables. An implementation of standard hash-table techniques in secondary storage.

Additional classes of indexes exist, such as *inverted indexes* used in Information Retrieval, but our concentration is on index structures used for storage and retrieval of *relational data*.

Indexes on Sequential Files.

Sequential Files store records ordered by values of a selected *search key*.

If relation R is stored sequentially and attribute A is a search key, then in order to give a fast answer to the query `SELECT * FROM R WHERE A= x` we need to know how to find the location in the disk file for R of the records whose search key value is x.

Possibile solutions are:

Scan: If we have only the sequential file to rely on, we can scan the file from the beginning, reading each page in turn, until we find the search key. The worst-case I/O complexity of such algorithm is n, where n is the number of pages in the disk file.

Dichotomy: A slightly better approach is *dichotomy*, which first reads the block in the middle of the file, determines, which side of it, the key value should be, and continues splitting the appropriate region of blocks into half until the necessary block is reached. This allows us to execute the query in $O(\log(n))$ disk accesses.

Indexing: A separate index structure is built, storing the information about the locations of records with specific key values. The index structure is much smaller than the data file, and its traversal can be done faster. Once the desired data is found in the index, only relevant pages are retrieved from the data file. The I/O costs in this case will be $O(m) + O(1)$ where $m << n$ is the size of the index file.

Two different types of index structures can be used in conjunction with sequential files: *dense indexes* and *sparse indexes*.

Simple Indexes

A *simple index* structure is a sequence of records of the form $(Value, Location)$, where $Value$ is the value of the *search key* of a relation stored in a sequential file, and $Location$ is the pointer to the location of the record with this search key[1].

The distinction between the two types of simple indexes, *sparse* and *dense* can be described as follows:

> ***Dense indexes*** *contain information about* ***every*** *key value in the relation, whereas* ***sparse indexes*** *contain information about only a subset of key values.*
>
> ***Dense indexes*** *allow the DBMS to determine* ***both*** *the* ***existance*** *and the* ***location*** *of a record in the relation, given a key value, without accessing the sequential data file.*
>
> ***Sparse indexes*** *allow the DBMS to determine the* ***location*** *of a record with a given key,* ***if such a record exists****. Generally, accessing the sequential file is required to establish that no record with a given key exists.*

[1]Note, that, technically, both $Value$ and $Location$ can be composite. If the search key includes more than one attribute, $Value$ will consist of the list of values of all search key attributes. $Location$ can be typically thought of as a $(PageId, RecordNumber)$ pair, where $PageId$ points to the page in the sequential file, and $RecordNumber$ - to the position of the record on the page.

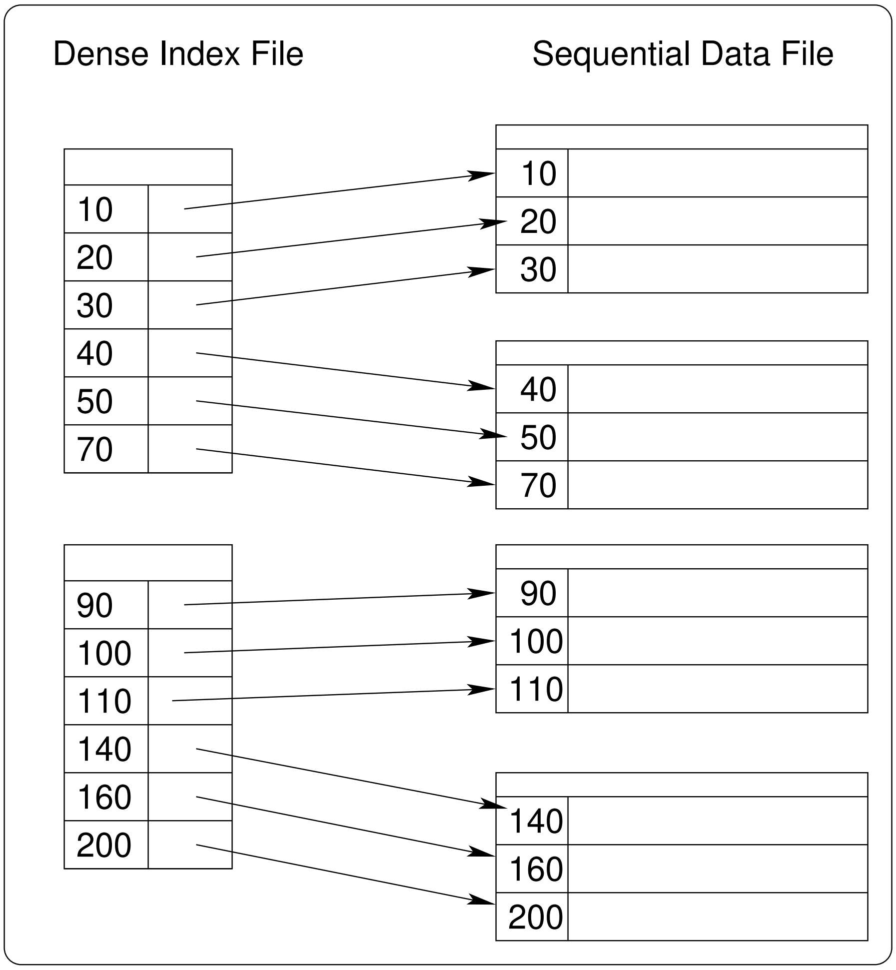
Dense Indexes

Dense indexes store information about *every* key in the data file.

The size of a record in an index file is typically much smaller than the size of a data record. If the key is a single attribute, then we can store a single index file record in $size(keyattribute)+4$ bytes, if we only store the *PageID* of the disk page on which the record is located. (alternatively, if we use $(PageID, RecordNumber)$ pair, the size will be $size(keyattribute) + 8$).

This means that *a single disk block can typically fit many more index records than data file records.*

A typical dense index is shown in the figure below:

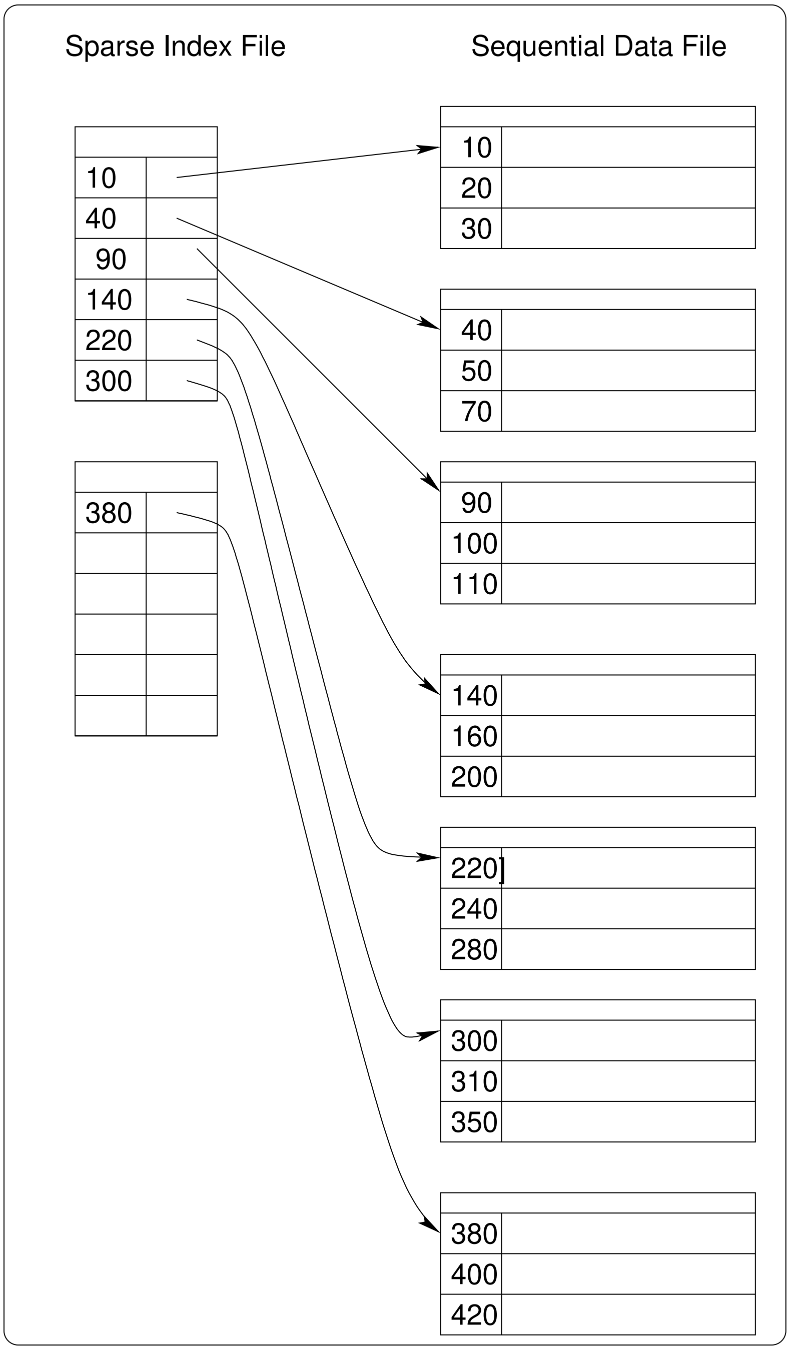


Sparse Indexes

Sparse indexes store information **only** about the first key of each disk block.

Where dense indexes win mostly because of the difference in the record size between an index and a data file, sparse indexes provide extra savings by storing only one (still small) record per disk page. This means that sparse indexes are **very compact**.

A typical sparse index is shown in the figure below:

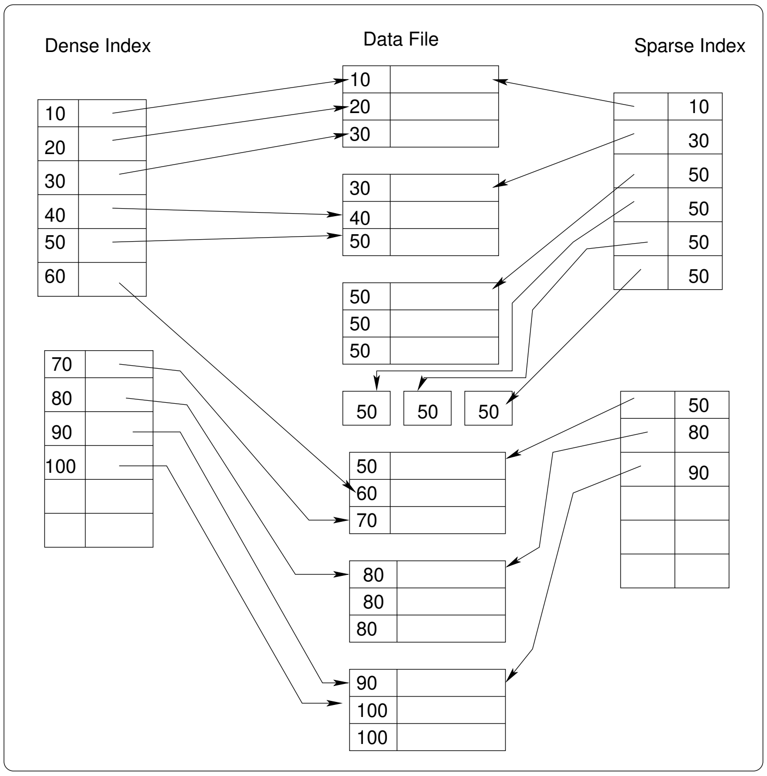


Index Structures for Keys with duplicate values

Both *dense* and *sparse* indexes can be adapted to deal with multiple key occurrences in the relation. The key principles behind these index structures stay:

- *Dense indexes* include one record for *each unique key value*.
- *Sparse indexes* include one record for *each disk block*.

This can be illustrated on the following example.



When retrieving information from indexes for data with duplicate keys the following needs to be observed

For dense indexes: Given a key value x, records with this value will be stored between the location pointed by the desnse index record with key x and the location pointed by the dense index record with the next key value. This interval may span multiple disk pages.

For sparse indexes: Given a key value x, to find records with this key in the data file, do the following:

- If index record with key x exists, retrieve the block it points to, and the previous block. (If more than one record for key x exists, also retrieve all blocks those reccords point to).
- If there is no record with key x, retrieve the block that precedes the first block where keys are greater than x.

Secondary Indexes

Secondary indexes are index structures that index the values of non-*search key* attributes of the relation.

Secondary indexes are used to index heap files and to index sequential files on non-search key attributes.

A *simple secondary index* consists of pairs $(AttributeValue, RecordLocation)$. *Secondary indexes* must be built to admit multiple keys. Because records with the same attribute value can be stored at different locations in the file, *simple secondary index* stores one index record for each tuple in the underlying relation.

An example of a secondary index is shown in Figure 1.

When retrieving from secondary indexa, we need to remember the following:

- Given key value x, the records with this value can be stored anywhere on disk. In the worst-case scenario, each block has exactly one record with this key, and therefore each block will have to be read in turn.

Secondary index structures can be compacted a little bit. Here are some variations.

- *Secondary index with indirection.* Only one record per key is stored in the index. The pointer leads to a page of pointers to locations of all records with given key value. See Figure 2.
- *Postings File.* Only one record per key is stored in the index. The record contains the key value and a list of pointers to locations of all records with this key in the data file.

  Note, that the record sizes in this index file are ***variable***. See Figure 3.

Index Maintenance

Any time an underlying data file is modified, the index structure may need to be modified appropriately.

An ***index file*** *is an example of a sequential data file.*

Therefore, the approach to insertion, deletion and modification into indexes is the same in general.

Notes:

- Index structures can be used during insertion of records into sequential data files to determine the position where the record must be inserted.
- A new index record **must be inserted** into a *dense index* any time a record with a new key is added to the relation.
- A new index record **must be inserted** into a *sparse index* any time a new disk page is created in the data file.

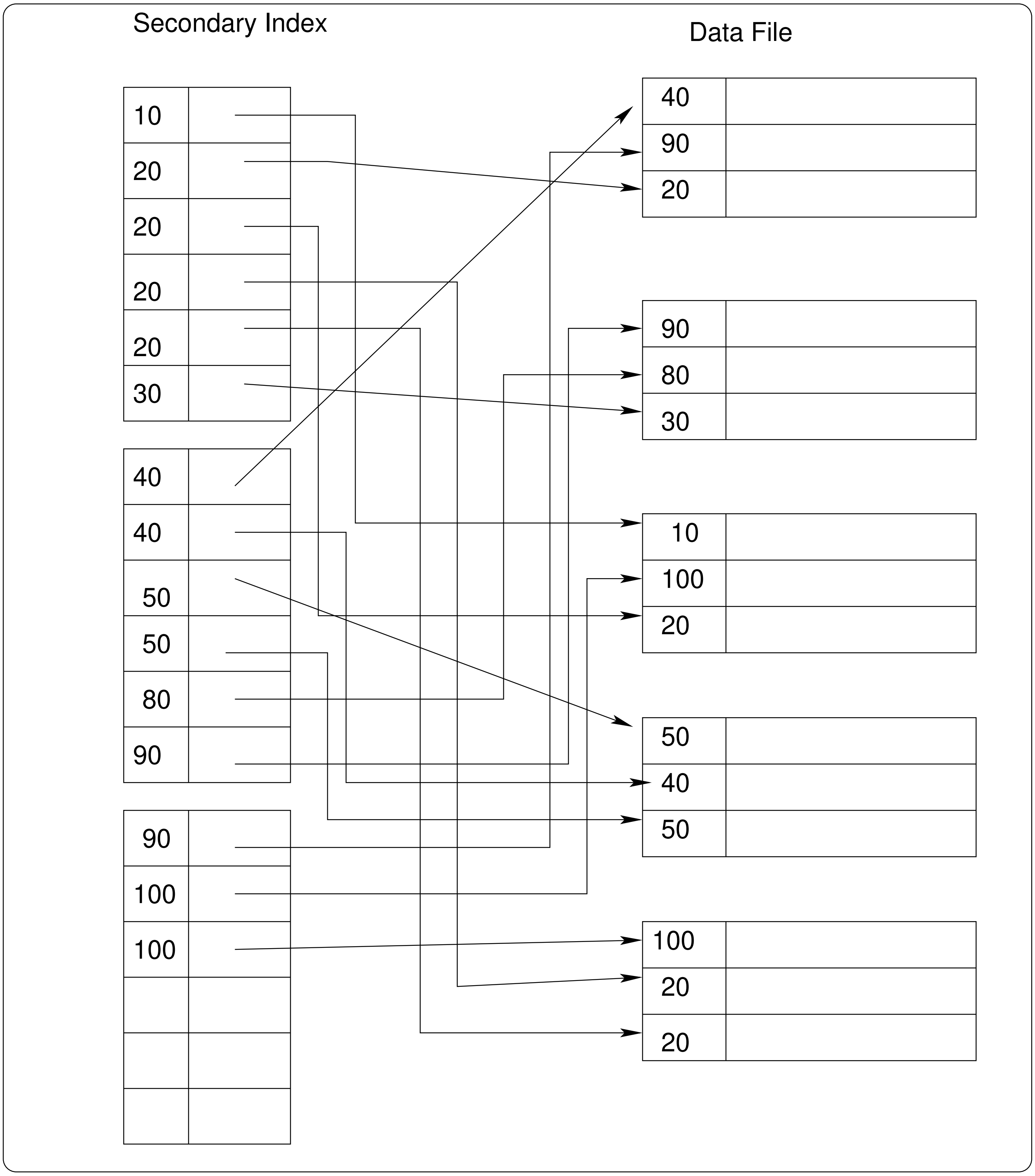


Figure 1: A Simple Secondary Index

- A new index record **must be inserted** into a *secondary index* any time a new record is inserted into the data file.
- Any time record insertion into a data file causes *sliding* or creation of an *overflow page*, i.e., any time, records are *moved* in the data file, **index structures must be updated** That is, pointers must be *updated* in some existing index records.

Action	Dense Index	Sparse Index	Secondary Index
Create empty overflow block	none	none	none
Delete empty overflow block	none	none	none
Create empty sequential block	none	insert	none
Delete empty sequential block	none	delete	none
Insert record	insert(?)	update(?)	insert
Delete record	delete(?)	update(?)	delete
Slide record	update	update (?)	update

"(?)" means "possibly."

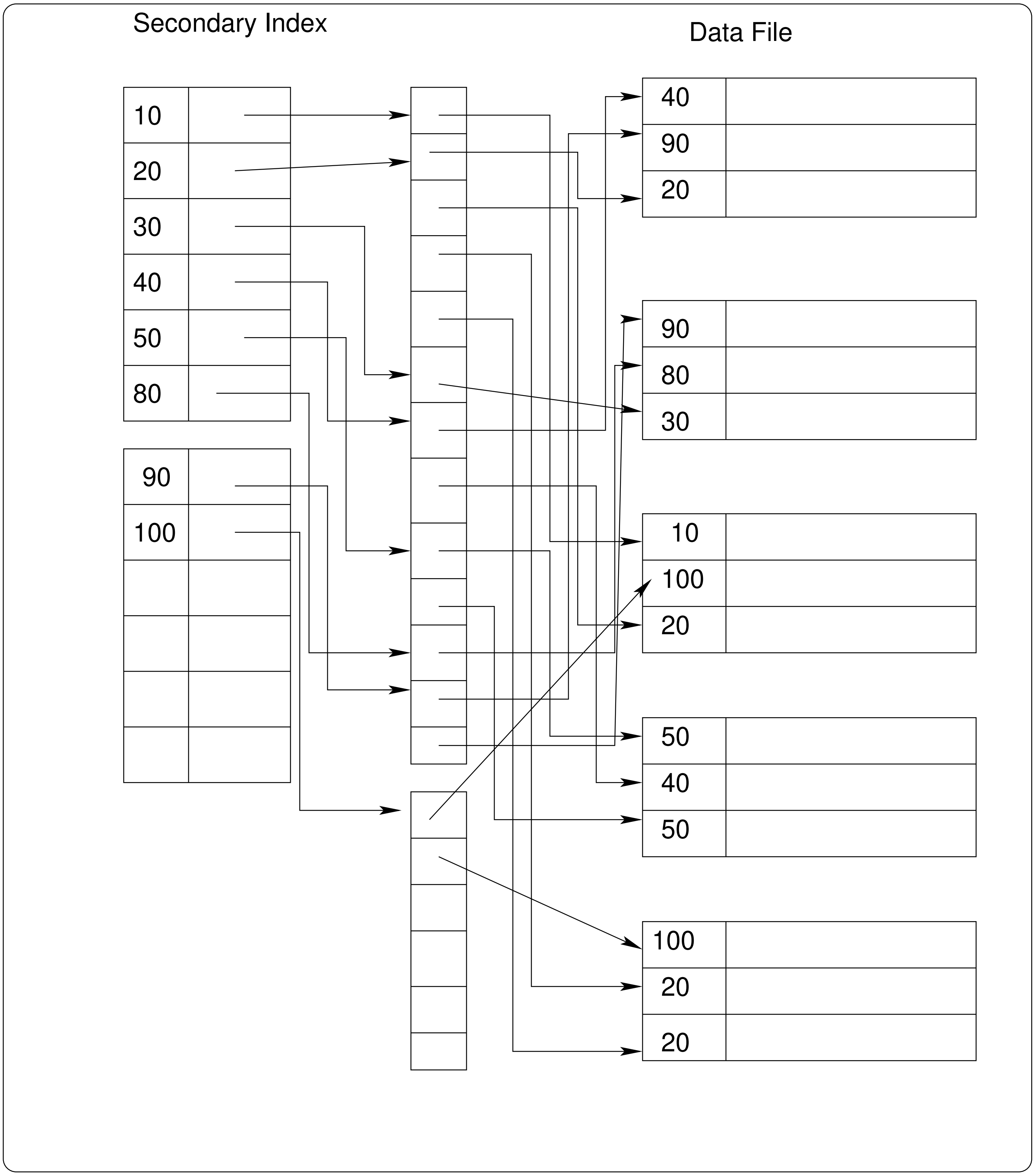


Figure 2: Secondary index with indirection.

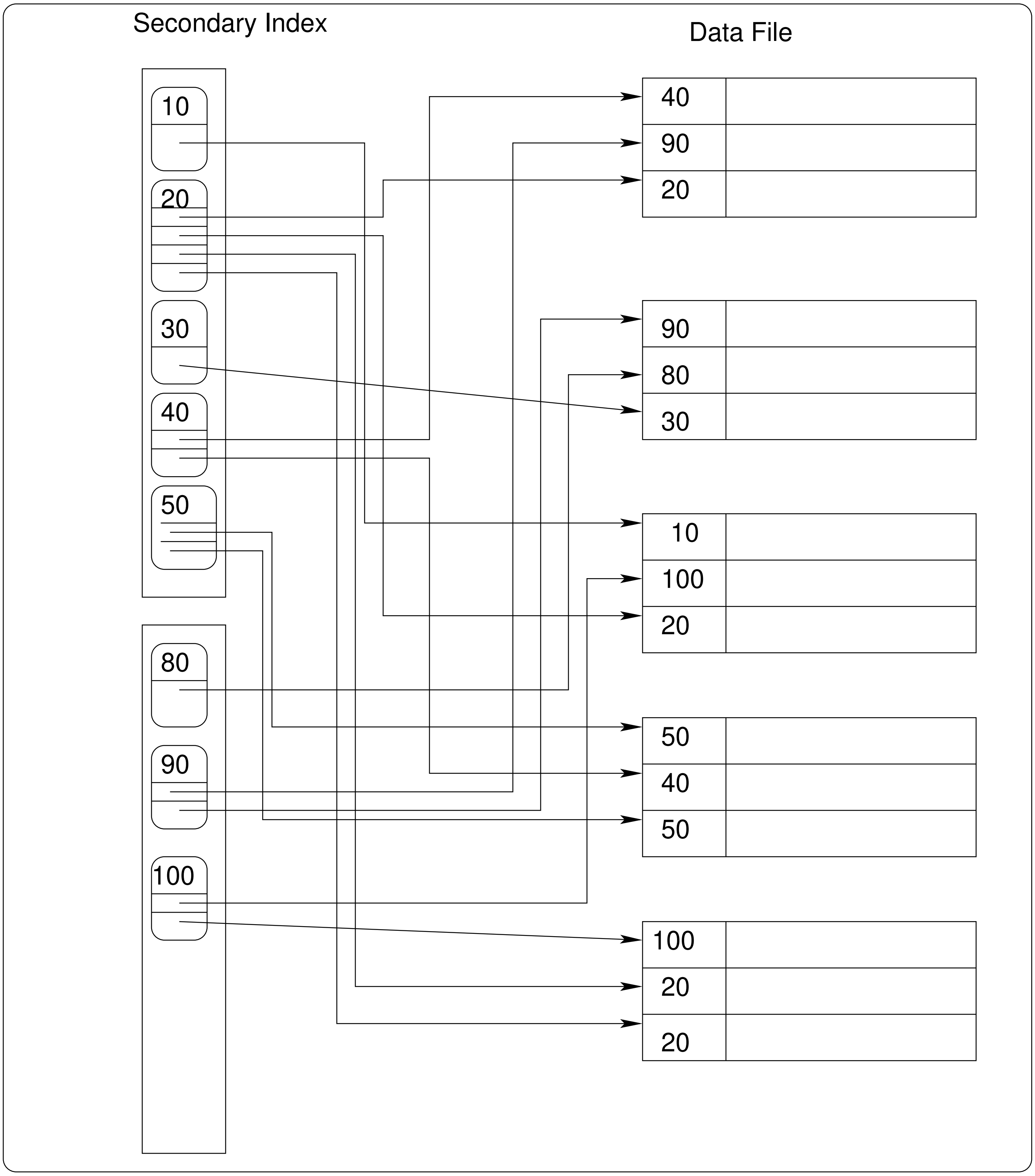


Figure 3: Secondary index with postings.