Indexes

Dr Eugenia Ternovska

Simon Fraser University

In this lecture

- Indexes: what they are and why they are needed
- Syntax for declaring indexes in SQL
- Selection of Indexes: a trade-off
- Index
  - Dense
  - Sparse
  - Primary
  - Secondary

Arranging data on a disk

- A data element (such as a tuple) is represented by a record that consists of consecutive bytes in some disk block[1]
- The simplest sort of record consists of fixed-length fields, one for each attribute of the represented data element (a tuple)
- Collections of data elements such as relations are stored in data files (a.k.a. files of records – an important abstraction in DBMSs)
- Data files can be stored in one, or spread over several blocks

[1] "Block" is a word used by the database community for a memory page

Indexes

An **index** is any data structure that

- takes the value of one or more fields (attributes) and
- finds the records (tuples) with that value "quickly"

An index organizes data records on a disk based on a **search key** (any subset of the felds of a relation)

- is used to optimize certain kinds of retrieval operations
- supports efficient retrieval of all data records satisfying a given condition on the search key

We can create multiple indexes with different search keys

Motivation for Indexes

Limit operation expense in large relations

```
SELECT * FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

There might be 10,000 Movies tuples, of which only 200 were made in 1990.

Naive way: (1) get all 10,000 tuples and (2) test the condition of the WHERE clause on each

More efficient: get only the 200 tuples from the year 1990 and test each of them to see if the studio is Disney

Even more efficient: obtain directly only the 10 or so tuples that satisfy both conditions in WHERE

Declaring Indexes in SQL

- Declaring Indexes
  CREATE INDEX <index-name> **ON**
  <relation(<attributes>)>;
- Removing Indexes
  DROP INDEX <index-name>;

Selection of Indexes

The choice of which indexes to create is one of the principal factors that influence database performance.

Is it beneficial to create as many indexes as possible? **No!**

Trade-off:

• The existence of an index on an attribute may speed up queries and joins involving that attribute

• But, every index built for attributes of some relation makes insertions, deletions, and updates to that relation more complex and time-consuming

Useful Kinds of Indexes (1)

The most useful index on a relation is an **index on its key**

1. Queries with a value for the key are common $\rightarrow$ index on the key will get used frequently
2. There is at most one tuple with a given key value
   - the index returns either nothing or one location for a tuple
   - at most one page must be retrieved to get that tuple into main memory

Useful Kinds of Indexes (2)

An index on an attribute can be effective, even if the attribute is not on a key:

1. **If the attribute is almost a key:** relatively few tuples have some given value for that attribute
   Even if each of the tuples with a given value is on a different page, we shall not have to retrieve many pages from disk
2. **If the tuples are "clustered" on that attribute:** We cluster a relation on an attribute by grouping the tuples with a common value for that attribute onto as few pages as possible. Then, even if there are many tuples, we shall not have to retrieve nearly as many pages as there are tuples

A multi-attribute index

Example:

```
CREATE INDEX KeyIndex ON Movies(title, year);
```

Here, (title, year) is an index key, so when we are given both a title and a year, the index will find only one tuple

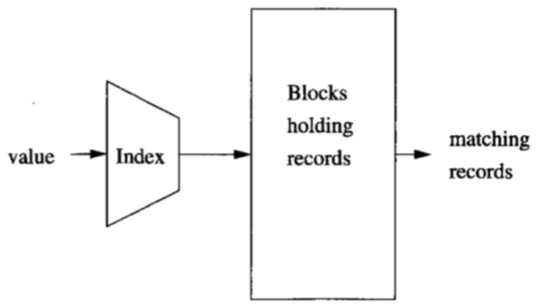
But, we can also use this index to find all the tuples with a given value in the first of the attributes

So, the choice of the order of the attributes is important:

- if we are more likely to specify a title than a year for a movie, then chose (title, year)
- if a year is more likely to be specified, then create an index on (year, title)

Index Structures

- An index lets us find a record without having to look at more than a small fraction of all possible records
- The field(s) on whose values the index is based is called the search key
- Most common form of index in database systems: B-tree



From Database Systems: The Complete Book, 2nd Edition

Index Structures

- A data file may be used to store a relation
- The data file may have one or more index files
- Index file associates values of the search key with pointers to data-file records that have that value for the attribute(s) of the search key
- Index
  - Dense
  - Sparse
  - Primary
  - Secondary

Several types of Indexes

Dense Indexes: there is an entry in the index file for every record of the data file

Sparse Indexes: only some of the data records are represented in the index, often one index entry per block of the data file

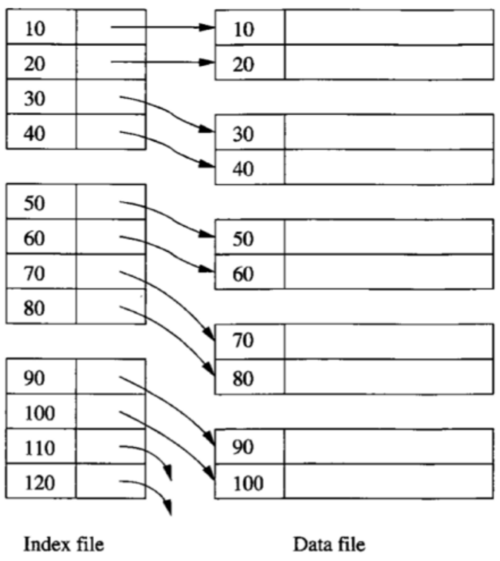
Primary index: determines the location of the records of the data file

Secondary index does not does not determine it

It is common to create a primary index on the primary key of a relation and to create secondary indexes on some of the other attributes

Index Structures: Sequential File

- Created by sorting the tuples of a relation by their primary key
- The tuples are then distributed among blocks, in this order



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

From Database Systems: The Complete Book, 2nd Edition

Index Structures: Dense Indexes

A dense index is a sequence of blocks holding only the keys of the records and pointers to the records themselves

- This index is advantageous when it can fit into the main memory, but not the data file
- By using the index, we can find any record given its search key, with only one disk I/O per lookup

- Supports queries asking for a certain key value K
- We search the index blocks for K, and when we find it, we follow the associated pointer to the record with key K
- Efficient
  - Small number of index blocks (compared to data blocks)
  - Keys are sorted: Binary search to find K (n blocks of the index: log2n)
  - Small index: Kept in memory permanently (no expensive I/O)

Index Structures: Sparse Indexes (1)

- A sparse index typically has only one key-pointer pair per block of the data file

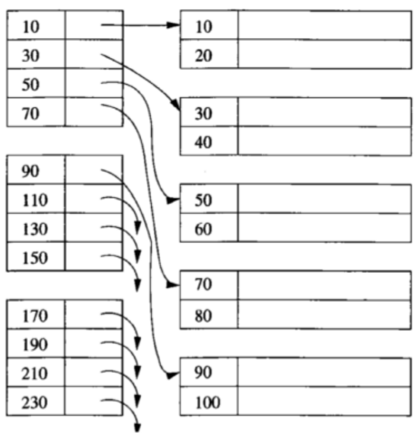
- It thus uses less space than a dense index, at the expense of more time to find a record given its key

- You can only use a sparse index if the **datafile is sorted by the search key**, while a dense index can be used for any search key

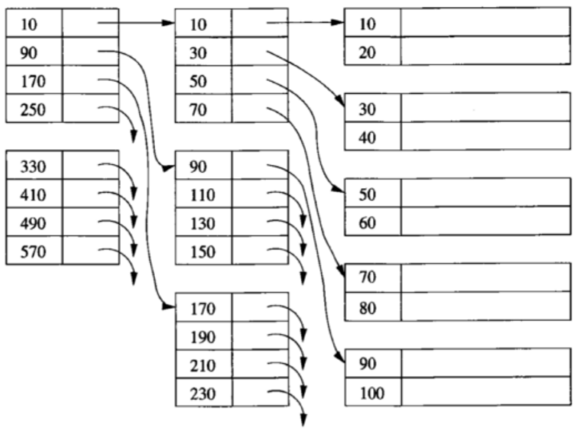
Index Structures: Sparse Indexes (2)

- We can define index on index, multi-level

But, this idea has its limits, and we prefer the B-tree structure over building many levels of index



A sparse index on a sequential file



Adding a second level of sparse index

From Database Systems: The Complete Book, 2nd Edition

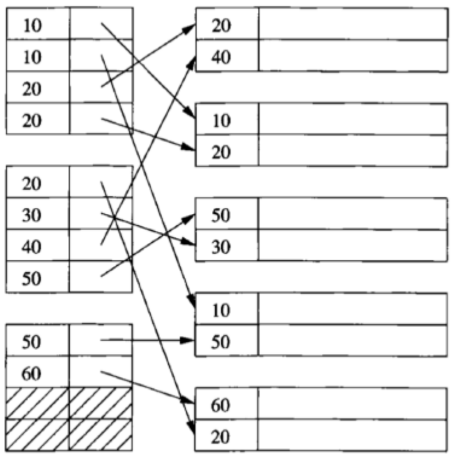
Index Structures: Secondary Indexes

- Does not determine the placement of records in the data file
- Tells us the current locations of records (which may have been decided by a primary index on some other field)

Always dense

Does not influence location of records

May result in many more disk I/O



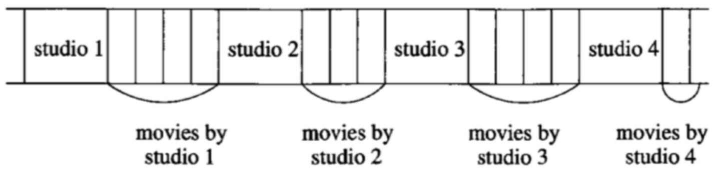
From

A secondary index

Database Systems: The Complete Book, 2nd Edition

Secondary Indexes: Applications

- There are some data structures where secondary indexes are needed for even the primary key
  - Heap Structure
  - Clustered File



A clustered file with each studio clustered with the movies made by that studio

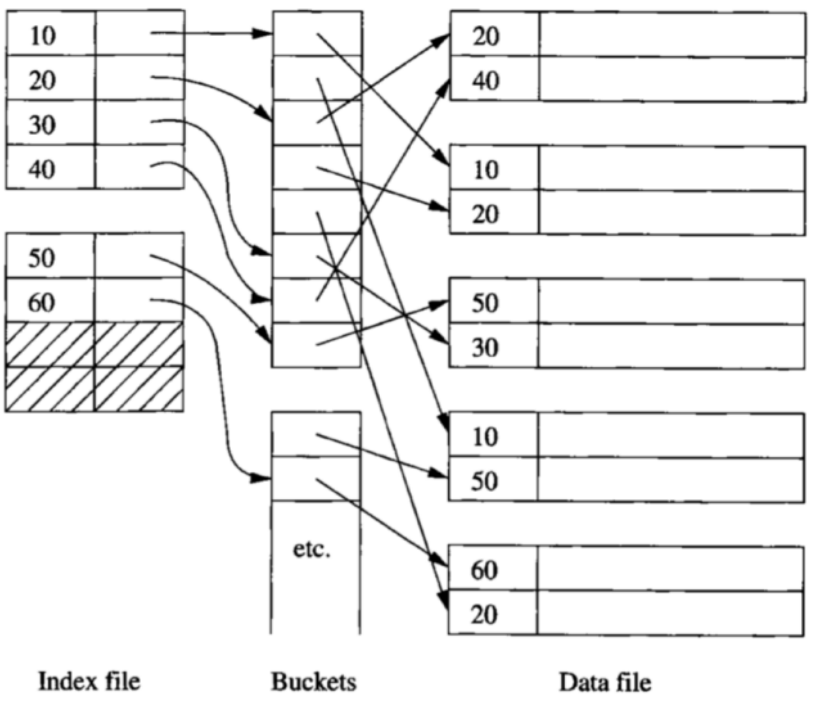
Studio(name, address, presCNum)
Movie (title, year, length, genre, studioName, producerCNum)

```
SELECT title , year
FROM Movie, Studio
WHERE presCNum = zzz AND Movie.studioName = Studio.name;
```

Secondary Indexes: Indirection

- In secondary indexes, a significant amount of space is wasted
- Search-key value n times in the data file → the value is n times in the index file
- Avoid repeating values
  Use **buckets**: a **level of indirection** between the secondary index file and the data file
- Use the pointers in the buckets to help answer queries without ever looking at most of the records in the data file

Secondary Indexes: Indirection

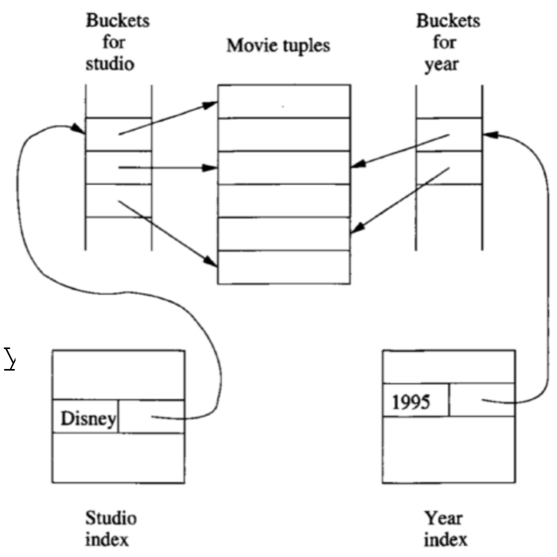


Saving space by using indirection in a secondary index

From Database Systems: The Complete Book, 2nd Edition

Secondary Indexes: Indirection

```
SELECT title
FROM Movie
WHERE studioName = 'Disney
AND year = 2005;
```



Intersecting buckets in main memory

From Database Systems: The Complete Book, 2nd Edition

Summary

In this lecture, we discussed:

- Indexes: what they are and why they are needed
- The syntax for declaring indexes in SQL
- How to select indexes
- Different kinds of Indexes: Dense, Sparse, Primary, Secondary

Acknowledgements

[1] Database Systems: The Complete Book, 2nd Edition Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer WidomPrentice Hall, 2009

[2] Database System Concepts, Seventh EditionAvi Silberschatz, Henry F. Korth, S. SudarshanMcGraw-Hill, March 2019www.db-book.com

Additional references and resources used in preparation of this course are listed on the course webpage or mentioned in slides.