CSC 370

Index structures

Overview

- Indexes vs. data file access
- Index structures: basic terms
- B-trees
- Hash tables
  - extensible hash tables
  - linear hash tables
- Multidimensional indexes
  - Motivation
  - Brief survey
- Bitmap indexes

Recall: relations/tables are really files

- Databases are normally too large to store in main memory
  - There are some exceptions yet we will ignore these
- Therefore some non-volatile storage is needed
  - Hard disks
  - SSD
  - Tertiary storage such as optical disks, tapes
- Efficient database operations therefore means reducing time required to access non-volatile storage
  - Memory hierarchy can help us somewhat here
  - Yet a heavier hand is needed (i.e., cannot always depend upon default hardware / OS behavior)

Reducing overhead of file I/O

- There are two basic approaches

1. Amortize the cost of a disk-block reads and writes over many operations
   - Block in many memory is used as much as possible.
   - Assumes we can predict ahead which blocks yield such benefits
2. Reduce the number of disk-block reads and writes as much as possible:
   - Use compact representations of data
   - **Indexes as an I/O optimizer**

Indexes are also files...

- Indexes are simply a kind of lookup mechanism for an SQL table
  - Lookup uses a **search key...**
  - and gets back a **pointer to / number for a disk block** holding tuple(s) satisfying that search key
- The search key for the index is specified at index creation time
  - It may correspond to the primary key(s) of the table
  - Or it may use other attributes
- When an index is working properly...
  - ... the total disk-block transfers for index file + data file are less than if the data file was use alone

Indexes are only one part of the puzzle

- Goal is to reduce overall disk I/O
  - Reduce number of disk blocks transferred between main memory and secondary storage
  - Doing this will increase DBMS performance
- Naïve use of indexes can actually increase the amount of I/O
  - Normally a DBMS will be able to calculate whether to go to an index or not
  - But sometimes we cannot depend upon this
- Must account for the mix of:
  - Typical queries
  - Characteristics of data returned from queries
  - Presence of indexes

Scenario

- Assume the following relation

```
StarsIn(movieTitle, movieYear, starName)
```

- Further assume three typical database operations:
  - Two different kinds of queries
  - One insertion operations
- We will eventually make assumptions the characteristics of data involved in the three operations

Query 1

- What are the titles and years of movies for which a given star appeared?
  - The star name is indicated by <s>
  - This would be replaced by a real string in the actual query.

```
select movieTitle, movieYear
from StarsIn
where starName = <s>;
```

Query 2

- Who are the stars that have appeared in a given movie?
  - Need both the movie title and movie year to uniquely identify an actual movie
  - Movie title: <t>
  - Movie year: <y>

```
select starName
from StarsIn
where movieTitle = <t> and movieYear = <y>;
```

Insertion operation

- Insert a new tuple into StarsIn
  - The star: <s>
  - Movie title: <t>
  - Movie year: <y>

```
insert into StarsIn(starName, movieTitle, movieYear)
    values (<s>, <t>, <y>)
```

Characteristics of data

- We make these assumptions to help with an analysis of disk I/O

1. StarsIn occupies 10 pages
   - If we need to examine the entire relation, disk I/O cost is 10 block reads
   - Ignore for now disk-block caching effects.
2. On average:
   - A star has appeared in three (3) movies.
   - A movie has three (3) stars.

Characteristics of data

3. Data is spread out through the ten blocks of StarsIn
   - Tuples for a given star are likely to be spread over the 10 pages
   - Same for tuples of a given movie.
   - If we have an index for starName, we still need to read one block of StarsIn for each tuple returned in Query 1.
   - If we have an index for the combination of movieTitle and movieYear, we will still need to read in one block of StarsIn for each tuple returned in Query 2.
   - However, if we have no indexes at all, then all ten blocks will need to be read in.

Characteristics of data

4. The index is stored on disk
   - One disk access is needed to read a page/block of the index when using the index
   - If the table is modified, then another disk access is needed to write a modified block back for the index.
5. Insertions result in one read and one write to StarsIn
   - That is, two disk-block transfers will occur
   - Assume, however, that we can determine which block will hold a tuple without needing to scan the whole relation (i.e., without scanning the whole file)

Further possibilities

- With respect to indexes, four combinations:
  A. There are no indexes
  B. There is a single index (on starName)
  C. There is a single index (on movieTitle and movieYear)
  D. There are two indexes (the two above)
- With respect to operation frequency:
  - Query 1 occurs with a probability of p_1
  - Query 2 occurs with a probability of p_2
  - Insertion occurs with a probability of $(1 - p_1 - p_2)$

I/O costs (i.e., number of block transfers)

Action	A. No Index	B. Star Index	C. Movie Index	D. Both indexes
Query 1	10	4	10	4
Query 2	10	10	4	4
Insertion	2	4	4	6
Average	$\boldsymbol{2 + 8p_1 + 8p_2}$	$\boldsymbol{4 + 6p_2}$	$\boldsymbol{4 + 6p_1}$	$\boldsymbol{6 - 2p_1 - 2p_2}$

What if $p_1 = 0.5$ and $p_2 = 0.1$?

What if $p_1 = 0.1$ and $p_2 = 0.5$?

Summary so far...

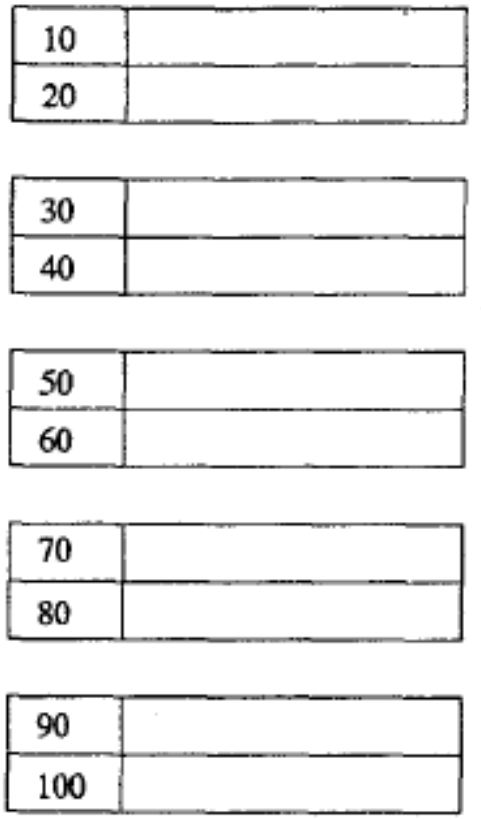
- Tables are stored in files in a sequence of disk blocks.
- Indexes can help a DBMS reduce the number of disk-block transfers needed to perform an SQL statement.
- However:
  - This depends very much on the kinds of SQL statements and the probability of their occurrence.
  - Naïve use of indexes **could actually increase** number of disk-block transfers needed.
- From here on, we will assume that a DBMS will choose intelligently whether or not to use an index.
  - Note that we only provide the index
  - We never indicate its use in an SQL statement that we write!

Index-Structure Basics

- Storage structures are made up of files residing on some OS
  - **Data file**: used to store a relation
  - **Index file**: use to store the index information for some data file
- **Sequential file**
- **Dense index**
- **Sparse index**
- **Secondary index**
- **Inverted index**

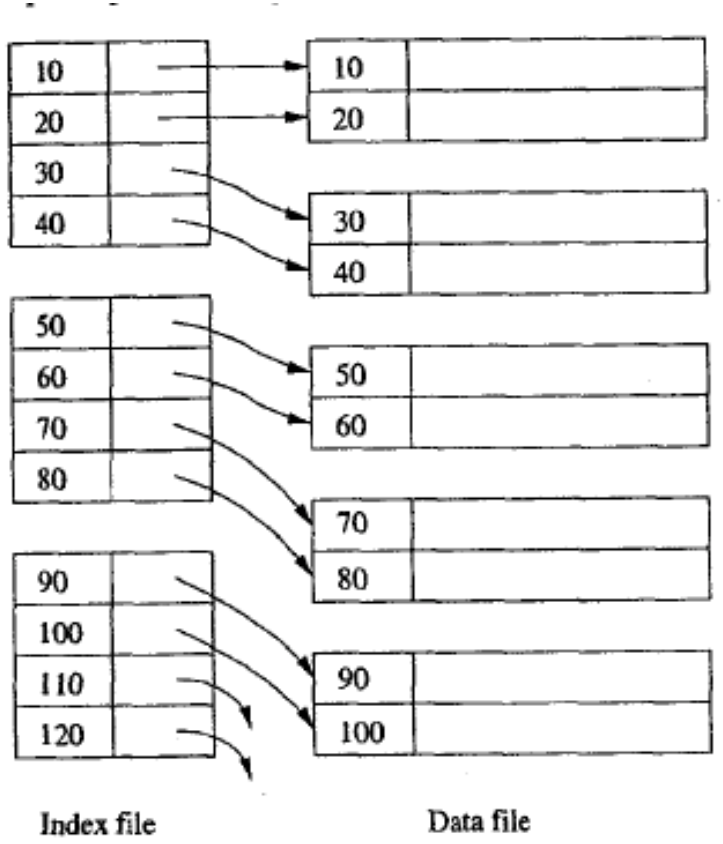
Sequential file

- This is a data file
  - Used to store the tuples of a relation
- Usually the tuples are stored in some sorted order
  - The order is by the table's primary key
- Running example can be seen on he right
  - Keys are integers (multiples of 10)
  - Ignore for now the rest of the tuple contents
  - Assuming very small blocks



Dense Indexes

- A sequence of block holding only keys of tuples/ records and points to data-file blocks themselves
  - Assumes here the tuples are sorted in the sequential file
- Keys and pointers usually take less room than tuples themselves
  - Therefore we expect many fewer blocks to store an index than the data file

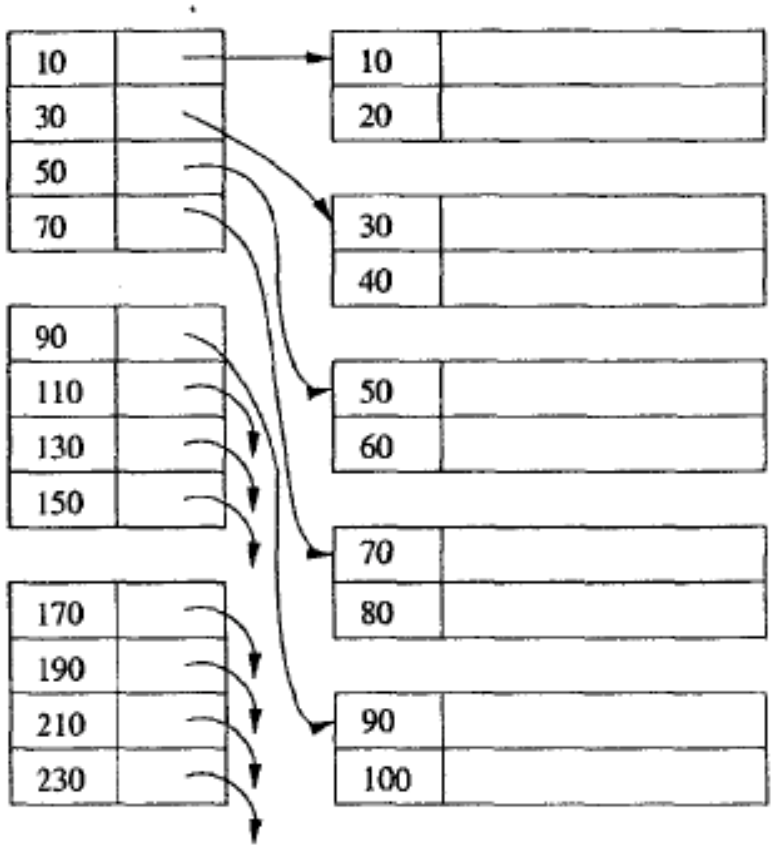


Dense indexes

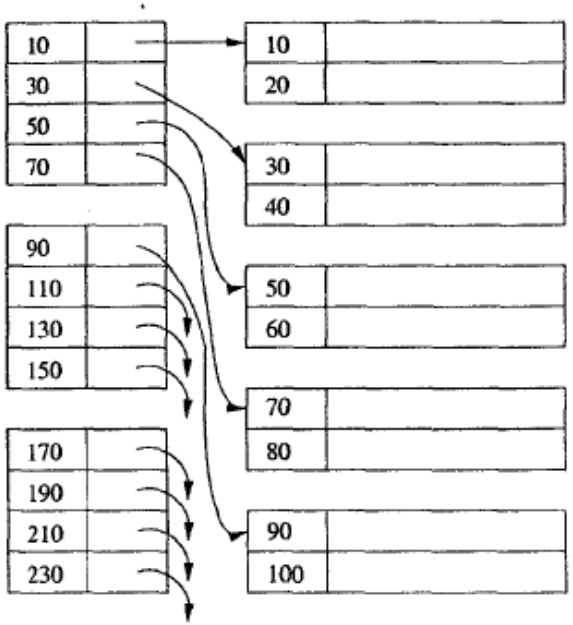
- Used as follows:
  - Given a key value K...
  - ... we search the index blocks for K.
  - When (if?) we find it, we follow the pointer to the disk block containing the tuple with key K.
- Some observations:
  - Number of index blocks is usually smaller than data file blocks
  - Since keys are sorted, a binary search can be used to find K (i.e., if index has n blocks, search would look through at most $\log_2 n$ of them)
  - Index may be small enough to reside in main memory

Sparse Indexes

- Usually has only one key-pointer pair per block of the data file
  - Keys are for the first records of each data block
  - Index therefore requires less space
- Can only use a sparse index if the data file is already sorted



Using a sparse index



- Goal is to find the tuple corresponding to search-key value K
- Search index for largest key less than or equal to K
  - Follow the associated pointer to a data-file block.
  - Now search this block for tuple with search-key value K

Observation: The index for a data file may be very large (i.e., cover many blocks). How might we handle this?

Multiple Levels of Index

- Even binary searches through dense or sparse indexes may result in many disk-block transfers
- If we put an index on the index, we can reduce some of this further
- First-level index: can be dense or sparse
- Higher-level indexes: must be sparse (or else there is no point!)