Chapter 5 - Tree Indexes

Given a dynamic file (many insertions and deletions)
we would like to do frequent independent fetches, consider

- an unsorted file
- a sorted file
- having an index (look up table)

Inverted Files:

- A simplest index structure that is in the form of an ordered list where each each entry is a (key, ptr) pair.
- difficult to maintain
  - After insertion and deletions, whole file needs to be shifted.

Most DBMSs use B+-trees and hash table utilities.

- we must learn how they work and what performance to expect.

ISAM (Indexed Sequential Access Method)

- the most extensively used indexing method in last decade.
- mostly promoted by IBM and INGRES DBMS, but obsolete today.

- ISAM is simple and efficient as long as no new records are added
  It contains
  - a memory-resident cylinder index that keeps the highest valued key for each cylinder
  - each cylinder contains an index that keeps the highest valued key for each block

memory-resident cylinder index

cylinder	high value	cylinder	high value	...
1	1001	2	2878	

index at cylinder 1

block	high value	block	high value	...
1	100	2	170	

$$T_F = \underbrace{r + s + btt}_{\text{Time to fetch the index on cylinder}} + \underbrace{r + btt}_{\text{Time to fetch correct block}}$$

$$T_x = \text{same as the sorted file}$$

(Actually a little bit longer since some space left on each cylinder for overflow.)

Disadvantages of ISAM:

- As new records are added, the ISAM file degrades in performance.
- It has to be reorganized at high cost.

Overflow Chains in ISAM

- We start with some empty tracks in each cylinder for overflow
- When a new record is added, old records are shifted to make place for the new one.
- The record which had the largest key in the block is moved to the overflow area.
- When the overflow area fills up, overflow is written to another cylinder
- Eventually the performance gets very slow.

Performance

- performance gets really poor when the distribution of new records could not be predicted in advance - very long overflow chains may occur
- With good prediction, enough space can be reserved in areas which are expected to grow

B+-Trees

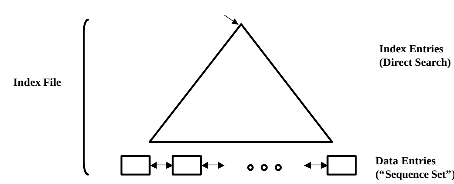
- Most used indexing method today.
- In B+-Trees:
  - nodes tend to have over 100 children
  - all leaves are on the same level
  - leaves contain the actual pointers to data on disk

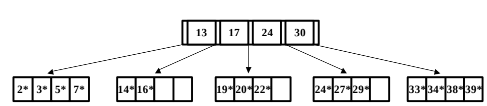
Any indexing structure which supports an ordering on a large file is likely to be implemented by a B+-tree.

- we can make efficient range queries.

- We shall show how a B+-tree can be used as a secondary or primary indexing method.
- We will look at the costs of fetching, sequential operations, and insertion/deletion.

Structure of a B+-Tree





Example of a B+-Tree, Order v = 2

Definition of a B+-Tree of Order *v*

- The root has at least two children unless it is a leaf.
- No internal node has more than $2v$ keys.
  - Root may have less keys
  - Internal nodes contain only keys and addresses of nodes on the next lower level.
- All leaves are on the same level.
  - When B+-tree is used as a primary index, the leaves contain the data records.
  - When B+-tree is used as a secondary index, the leaves contain the keys and record addresses.
- An internal node with k keys has $k+1$ children.

Bucket factor (Bkfr) : the # records that can fit in a leaf node.
Fan-out: the average # children of an internal node.

- B+-trees are short and wide.
- The records take up more space than the keys and addresses.
  - Typically internal nodes carry on 100-200 keys, leaves carry on 15 records.
- A primary index determines the way the records are actually stored.
- Clustering index: records are stored together in buckets acc.to the values of the key.
  - The records in a given bucket will have nearby key values.
  - The index only note the lowest or the highest key in a given bucket.
    - For this reason, clustering index, is often called a *sparse index* (e.g., ISAM, a B+-tree with data in the leaves)

- A B+-tree can also be used for a secondary index.
  - The records in the file are not grouped in buckets according to the keys of secondary indexes.
  - A secondary index is also a dense index where an entry exists for each record in the file (e.g., a B+-tree where leaves contain keys and addresses of records)
- There may be many secondary indexes for the same file.
- Why not have a secondary index on each field in the file?
  - this would need repeating all the information in the file in the leaves of the trees.
  - with many indexes, update costs becomes high.