Indexing techniques for databases

Hans Philippi

December 19, 2023

Why indexing?

- Suppose you are a police officer
- A suspicious car is passing by at high speed
- You want to check the license plate

Why indexing?

- Around 10.000.000 cars in the Netherlands
- Query: search a car based on license plate
- Assumptions:
  - A tuple (record) takes 400 bytes
  - A hard disk block contains 16 kbyte, so we have 40 tuples on a block
  - A disk IO takes 5 msec
- Maximum search time (complete table scan)
- 10.000.000 / 40 = 250.000 disk IO
- Search time : 1250 sec = 21 minutes
- Required search time : $<$ 1 sec

Memory characteristics (figures from 2023)

Main memory

- Typical size : 4 – 256 GB
- Access time: 100 nsec (10^{-7} sec)
- Volatile

Harddisk (block size: 2 – 32 kbyte)

- Typical size : 2 – 14 TB
- Access time: 5–10 msec (10^{-2} sec)
- Some speedup possible with clustering
- Non-volatile

SSD (expensive)

- Typical size : 256 GB – 8 TB
- Access time: 0,1 msec (10^{-4} sec)
- Non-volatile

Indexing

- Indexing enables a quick table search, based on the value of a specific attribute
- Indexing also supports query processing and optimization
- Indexing supports primary key maintenance and uniqueness constraints (other candidate keys)
- Syntax for SQL DDL:

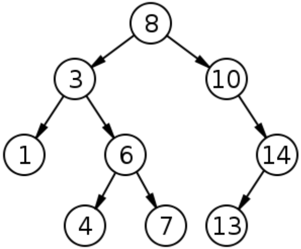
```
CREATE INDEX Person_dob_ndx
ON Person (date_of_birth);
CREATE UNIQUE INDEX Person_ppn_ndx
ON Person (passport_number);
```

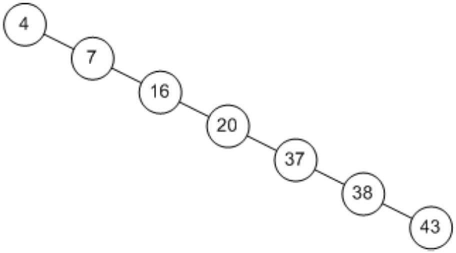
Indexing techniques

- Two fundamental techniques
  - Indexing based on search trees
  - Indexing based on hashing
- Both techniques are applicable to main memory as well as external memory
- Both techniques deal with block sized memory traffic

Search trees

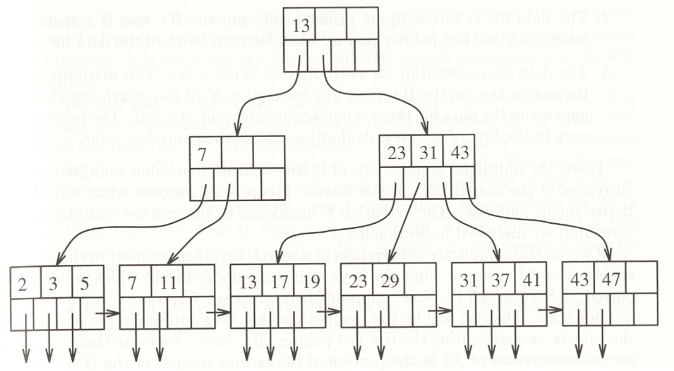
- The most well known search tree is the binary search tree
- Search time is $O(log(n))$ for n entries, when balanced
- Problem: maintaining balance under updates





B-tree

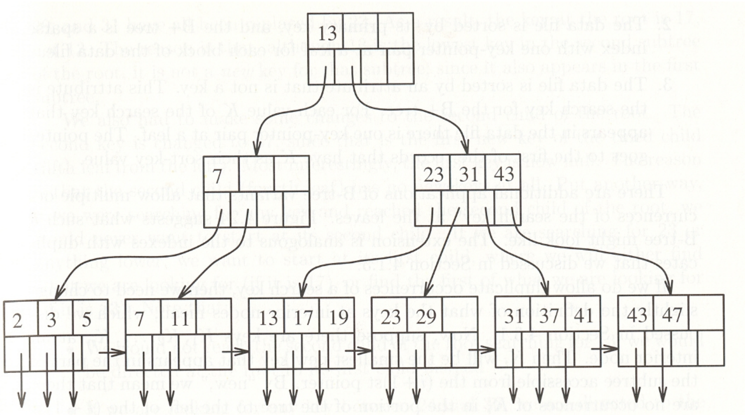
- Standard multiway search tree applied in relational databases
- Sophisticated updating techniques to keep it balanced
- Guarantees at least 50% filling of nodes
- Nodes correspond to disk blocks



Source: *Garcia-Molina e.a: Database Systems, The Complete Book*

B-tree

- Lowest level contains all attribute values and pointers to corresponding tuples
- Lowest level contains sibling pointers supporting range queries



Source: *Garcia-Molina e.a: Database Systems, The Complete Book*

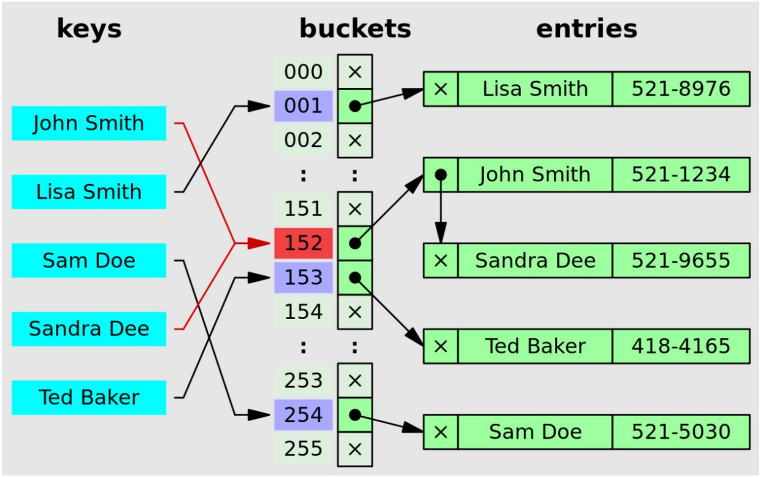
B-tree: an example

- Attribute value: 4 byte integer
- Pointer: 8 bytes
- Block size: 16 kbyte
- Content: $683 - 1365$ entries per block
- 2 levels: minimum nr of entries $= 466000$
- 3 levels : minimum nr of entries $= 318$ million
- 4 levels : minimum nr of entries $= 217$ billion
- Number of pointer traces is limited by $\lceil ^k log(n) \rceil$, with $k = 683$
- Search time in our example: $<< 1$ sec

Hash table

- Memory reservation of N buckets: virtual addresses 0..N-1
- Hashfunction f
  - Domain: all possible attribute values
  - Codomain: 0..N-1
- The hash function calculates the bucket address from the current attribute value
- Hashfunction f should distribute the addresses evently
- More info: `https://en.wikipedia.org/wiki/Hash_function`

Hash table



Source: `https://en.wikipedia.org/wiki/Hash_function`

Final words

- Hash indexing has a theoretical advantage: one disk access versus ${}^{k}log(n)$ for B-tree
- Hash indexing has a fundamental disadvantage: range queries are not supported ...
- ... while B-trees support range queries by horizontal links on the lowest level
- The k of ${}^{k}log(n)$ is very large, so ${}^{k}log(n)$ hardly exceeds 3 ...
- ... while the root of the B-tree is (and possibly the second level nodes are) often kept in main memory
- Overall, the B-tree is the winner