Data Structures

B-Tree Structure

Jing Ming

物联网19级

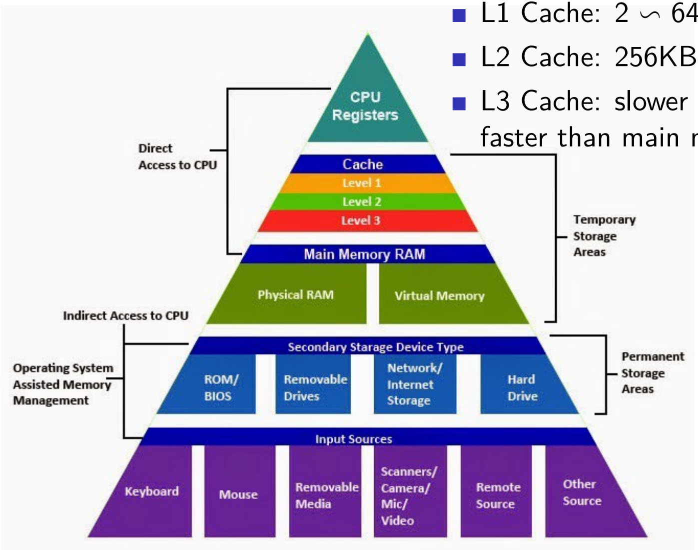
Spring2021

Outline

Code Example

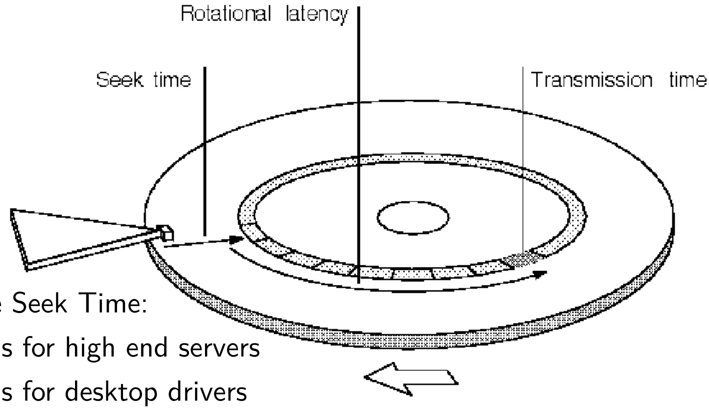
```
for(int i = 0; i < 4000; i++){
    for(int j = 0; j < 4000; j++){
        sum += arr[i * 4000 + j];
    }
}

// the code block above runs 10x faster than the one below

for(int i = 0; i < 4000; i++){
    for(int j = 0; j < 4000; j++){
        sum += arr[i + 4000 * j];
    }
}
```

Memory Hierarchy



- L1 Cache: 2 ∽ 64KB
- L2 Cache: 256KB ∽ 2MB
- L3 Cache: slower than L2, faster than main memory

Traditional HDD (Hard Disk Drive)



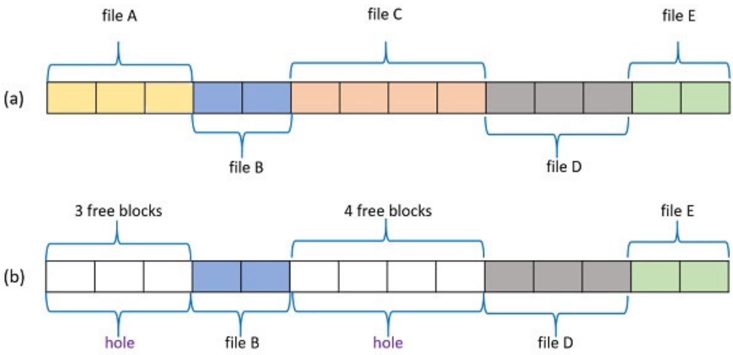
Average Seek Time:

- 4ms for high end servers
- 9ms for desktop drivers
- .1ms for SSD

Latency Comparison Numbers (2012)

L1 cache reference	0.5	ns				
Branch mispredict	5	ns				
L2 cache reference	7	ns				
Mutex lock/unlock	25	ns				
Main memory reference	100	ns				
Compress 1K bytes with Zippy	3,000	ns	3	us		
Send 1K bytes over 1 Gbps network	10,000	ns	10	us		
Read 4K randomly from SSD*	150,000	ns	150	us		
Read 1 MB sequentially from memory	250,000	ns	250	us		
Round trip within same datacenter	500,000	ns	500	us		
Read 1 MB sequentially from SSD*	1,000,000	ns	1,000	us	1	ms
Disk seek	10,000,000	ns	10,000	us	10	ms
Read 1 MB sequentially from disk	20,000,000	ns	20,000	us	20	ms
Send packet CA->Netherlands->CA	150,000,000	ns	150,000	us	150	ms

Contiguous Memory



(a) Contiguous memory allocation of 5 files
(b) When the file A and C terminates and release the memory creating hole

Arrays

Zero-Based Index

```
int primes[] = {1, 3, 5, 7, 11};

primes[0] // return 1st prime number
primes[1] // return 2nd prime number
primes[2] // return 3rd prime number
...       // and so on
```

Arrays: Pros and Cons

Pros:

- fixed length data structures
- offer great memory locality

For large data sets stored in an array, two issues arise:

- Dynamic array resizing.
- Search. O(n) without keeping data sorted.
- Insert. Require rearranging large sorted data set.

Large Data Set Storage

Figure: Large Data Set cannot be stored in one contiguous block of external memory

Figure: Sequentially break big array into multiple small ones. Linear search inefficiently.

B-Tree Solution

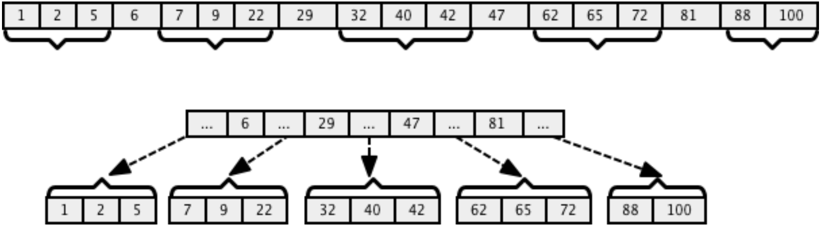


Figure: B-Tree Example

A B-Tree of order k (children) is an k-ary search tree

- The root node is either a leaf or has at least two children.
- Each node, except for the root and the leaves, has between k/2 and k children. This is to make sure that tree is making optimal use of space and is not skewed.
- Each path from the root to a leaf has the same length. In other words, all leaf are at same level.
- The root, each internal node, and each leaf is typically a disk block.
- Each internal node has up to (k - 1) key values and up to k pointers to children, as k is the order of tree (order=maximum children).
- The records are typically stored in leaves. In some cases, they are also stored in internal nodes.

B+Tree Solution

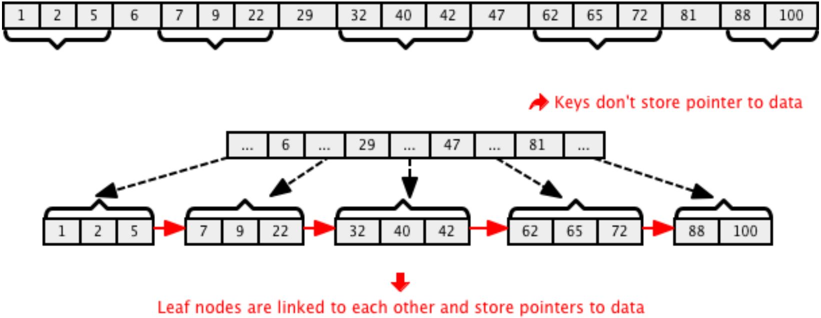


Figure: B+Tree Example

B+ Trees are different from B Trees

- B+ trees don't store data pointer in interior nodes, they are ONLY stored in leaf nodes. This is not optional as in B-Tree. This means that interior nodes can fit more keys on block of memory and thus fan out better.
- The leaf nodes of B+ trees are linked, so doing a linear scan of all keys will requires just one pass through all the leaf nodes. A B tree, on the other hand, would require a traversal of every level in the tree. This property can be utilized for efficient search as well, since data is stored only in leafs.

B+Tree for Index

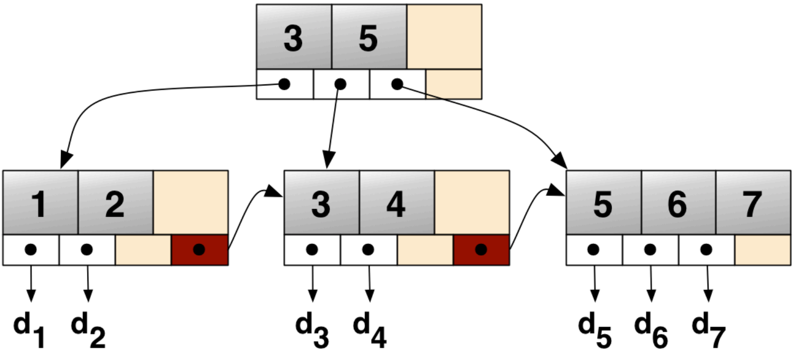


Figure: B+Tree Index. $(d_1, d_2, ..., d_7)$ corresponds to the no of the pages.