Indexing Large Data

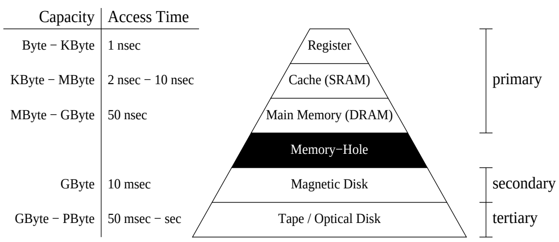
Victoria University of Wellington
Te Whare Wānanga o te Ūpoko o te Ika a Māui
Capital City University

Introduction

- Files, file structures, DB files, indexes
- B-trees and B+-trees
  - Reference Book (in VUW Library): *Fundamentals of Database Systems* by Elmasri & Navathe (2011), (Chapters 16 and 17 only!)

The Memory Hierarchy

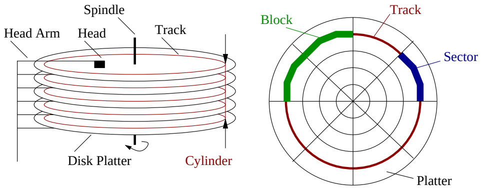
- Memory in a computer system is arranged in a hierarchy:



The Memory Hierarchy

- Registers
  - in CPU.
- Memory caches
  - Very fast, small
  - Copy of bits of primary memory
- Primary memory:
  - Fast, large
  - Volatile: lost in power outage
- Secondary Storage (hard disks)
  - Slow, very large
  - Persistent
  - Data cannot be manipulated directly,
    - data must be copied to main memory,
    - modified
    - written back to disk
- Need to know how files are organised on disk

Disk Storage Devices



- Disk controller: interfaces disk drive to computer
- Implements commands to read or write a sector

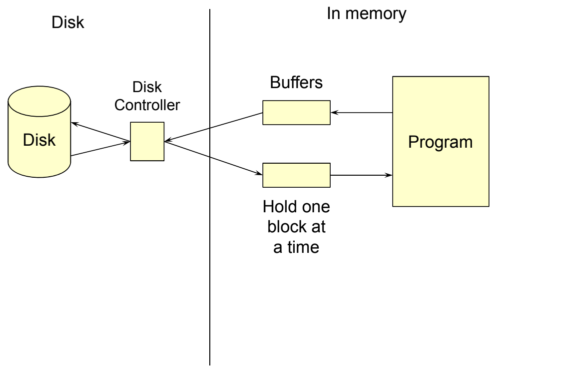
Disk Organisation

Data on a disk is organised into chunks:

- Sectors:
  - physical organisation.
  - hardware disk operations work on sector at a time
  - traditionally: 512 bytes
  - modern disks: 4096 bytes
- Blocks:
  - logical organisation
  - operating system retrieves and writes a block at a time
  - 512 bytes to 8192 bytes,

⇒ For all efficient file operations, minimise *block* accesses

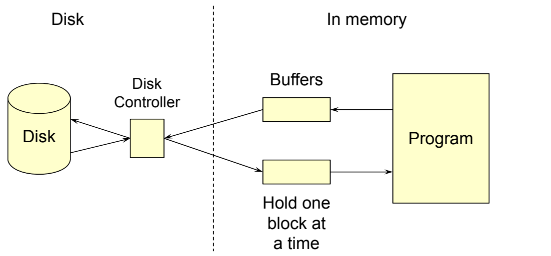
Reading/Writing from files



File Organisation

Files typically much larger than blocks

- A file must be stored in a collection of blocks on disk
  - Must organise the file data into blocks
  - Must record the collection of blocks associated with the file
  - Must enable access to the blocks (sequential or random access)



Files of records

- File may be a sequence of records (especially for DB files)
  - record = logical chunk of information
    - eg a row from a DataBase table
    - an entry in a file system description
    - …
  - May have several records per block
    - "blocking factor" = # records per block
    - can calculate block number from record number (& vice versa) (if records all the same size)
    - # records = # blocks × bf

Using B+ trees for organising Data

- B+ tree is an efficient index for very large data sets
- The B+ tree must be stored on disk (ie, in a file)
  - ⇒ costly actions are accessing data from disk
- Retrieval action in the B+ tree is accessing a node
  - ⇒ want to make one node only require one block access
  - ⇒ want each of the nodes to be as big as possible ⇒ fill the block

B+ tree in a file:

- one node (internal or leaf) per block.
- links to other nodes = index of block in file.
- need some header info.
- need to store keys and values as bytes.

Implementing B+ Tree in a File

- To store a B Tree in a block-structured file
  - Need a block for each node of the tree
    - Can refer to blocks by their index in the file.
  - Need an initial block (first block in file) with meta data:
    - index of the root block
    - number of items
    - information about the item sizes and types?
  - Need a block for each internal node
    - type
    - number of items
    - child key child key.... key child
  - Need a block for each leaf node
    - type
    - number of items
    - link to next
    - key-value key-value .... key-value

index of block containing child node

Cost of B+ tree

- If the block size is 1024 bytes, how big can the nodes be?
- Node requires
  - some header information
    - leaf node or internal node
    - number of items in node,
  - internal node:
    - m_N x key size
    - m_N+1 x pointer size
  - leaf node
    - m_L x item size
    - pointer to next leaf

Must specify which node, ie which block of the file

Leaf nodes could hold more values than internal nodes!

- How big is an item?
- How big is a pointer?

Cost of B+ tree

- If block has 1024 bytes
- each node has header $\Rightarrow$ 5 bytes
- If key is a string of up to 10 characters $\Rightarrow$ 10 bytes
- if value is a string of up to 20 characters $\Rightarrow$ 20 bytes
- If child pointer is an int $\Rightarrow$ 4 bytes

- Internal node (m_N keys, m_N+1 child pointer)
  - size = 5 + (10 + 4) m_N + 4
    $\Rightarrow$ $m_N \leq (1024 - 9)/ 14 = 72.5$ $\Rightarrow$ 72 keys in internal nodes
- Leaf node (with pointer to next)
  - size = 5 + 4 + (10 + 20) m_L
    $\Rightarrow$ $m_L \leq (1024 - 9)/ 30 = = 33.8$ $\Rightarrow$ 33 key-value pairs in leaves

B+ Tree: Find

To find value associated with a key:

<u>Find(key)</u>:

```
    if root is empty    return null
    else return Find(key, root)
```

$C_0,\ K_1\ C_1,\ K_2\ C_2,\ \cdots\ C_{node.size-1}\ K_{node.size}\ C_{node.size}$

<u>Find(key, node)</u>:

```
    if node is a leaf
        for i from 0 to node.size-1
            if key = node.keys[ i ]   return node.values[ i ]
        return null
    if node is an internal node
        for i from 1 to node.size
            if key < node.keys[ i ] return Find(key, getNode(node.child[i-1]))
        return Find(key, getNode(child[node.size] ))
```

Could use binary search

K_0-V_0, K_1-V_1, K_2-V_2, …, $K_{leaf.size-1}$-$V_{leaf.size-1}$

B+ Tree Add (1)

<u>Add(key, value)</u>:

```
    if root is empty
        create new leaf, add key-value,
        root ← leaf
    else
        (newKey, rightChild) ← Add(key, value, root)
        if (newKey, rightChild) ≠ null
            node ← create new internal node
            node.size ← 1
            node.child[0] ← root
            node.keys[1] ← newKey
            node.child[1] ← rightChild
            root ← node
```

If root was full: returns new key and new leaf node,

Make a new root node

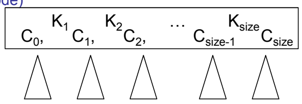
B+ Tree Add (2)

Add(key, value, node):

if node is a leaf
  if node.size < maxLeafKeys
    insert key and value into leaf in correct place
    return null
  else
    return SplitLeaf(key, value, node)

K_0-V_0, K_1-V_1, K_2-V_2, …, K_{size-1}-V_{size-1}

Returns new key and new leaf node,



if node is an internal node
  for i from 1 to node.size
    if key < node.keys[i]
      (k, rc) ← Add(key, value, node.child[i-1])
      if (k, rc)=null return null
      else return dealWithPromote(k,rc,node)
  (k, rc) ← Add(key, value, node.child[node.size])
  if (k, rc)= null return null
  else return dealWithPromote(k, rc, node)

Inserts new key and child into node,

B+ Tree Add (3)

Could make the array one larger than necessary to give room for this.

SplitLeaf(key, value, node):
  insert key and value into leaf in correct place (spilling over end)
  sibling ← create new leaf
  mid ← ⌊(node.size+1)/2⌋
  move keys and values from mid … size out of node into sibling.
  sibling.next ← node.next   node.next ← sibling
  return (sibling.keys[0] , sibling)

= ⌊(max_L+2)/2⌋ since size is now max_L+1

☆

K_0-V_0, K_1-V_1, K_2-V_2, …, K_{size-1}-V_{size-1}

K_0-V_0, K_1-V_1, K_2-V_2, … key-value … K_{size-1}-V_{size-1}

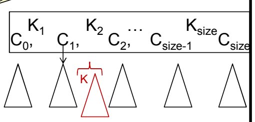
K_0-V_0, …, K_{mid-1}-V_{mid-1}

K_{mid}-V_{mid}, …, K_{size-1}-V_{size-1}

B+ Tree Add (4)

Nothing was promoted

DealWithPromote(newKey, rightChild, node):



if (newKey, rightChild) = null return null
if newKey > node.keys[node.size]
  insert newKey at node.keys[node.size+1]
  insert rightChild at node.child[node.size+1]
else for i from 1 to node.size
  if newKey < node.keys[i]
    insert newKey at node.keys[i]
    insert rightChild at node.child[i]
if size ≤ maxNodeKeys return null

No need to promote further

sibling ← create new node
mid ← ⌊size/2⌋ +1
move node.keys[mid+1… node.size] to sibling.node [1… node.size-mid]
move node.child[mid … node.size] to sibling.child [0 … node.size-mid]
promoteKey ←node.keys[mid],
remove node.keys[mid]
return (promoteKey, sibling)

Node is overfull:
Have to split and promote