ADB Lecture 1

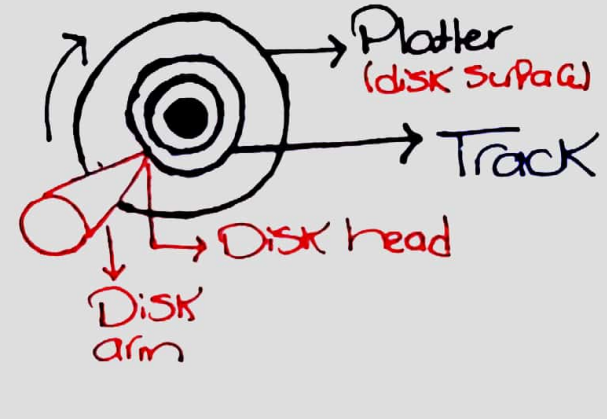
⇒ In this lecture, we'll understand how the DBMS (database management server) stores records

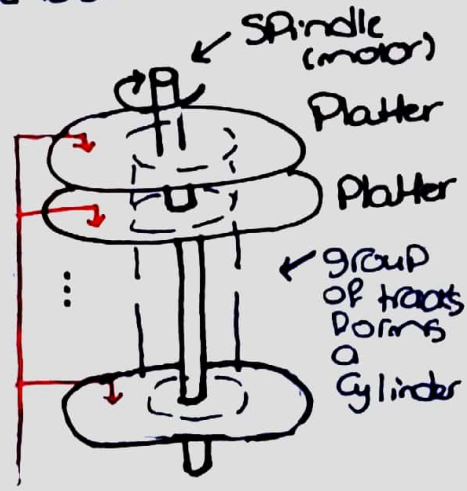
→ Data corresponding to records is huge & must persist across program executions

Hence, the DBMS uses a HDD (Hard Disk Drive)

→ Many layers of

- magnetic disk
- Stores 0s and 1s by using 2 diff magnetic polarity





2. Every Platter has a group of concentric circles (tracks)
   → its capacity is usually in the 4-50 KB or more range. Notice that outer tracks → more data #Zoning

1. The group of Platters (magnetic disks) connected to the rotating spindle is called a disk pack

3. Each track is further divided into sectors. // smallest addressable unit on HDD

* The OS can choose to deal with the disk at a different basic unit (logical block)

→ In this case, it must be a multiple of the sector size. // intuitive

* Often set as large as the Page Size (basic unit for main memory)

- Consequently, in a DBMS a Page
  - → has the same size as a block
  - → is the smallest unit for read/write from disk
  - → corresponds to a set of consecutive records in the DB

- To retrieve a random Page

1. Disk Arm Positions the disk head on the track that has the Page (via a seeking motion).
2. It then has to wait for the disk to rotate so that the head is on the right block
3. Then wait until reading (or writing) the block is done (by rotating over it)

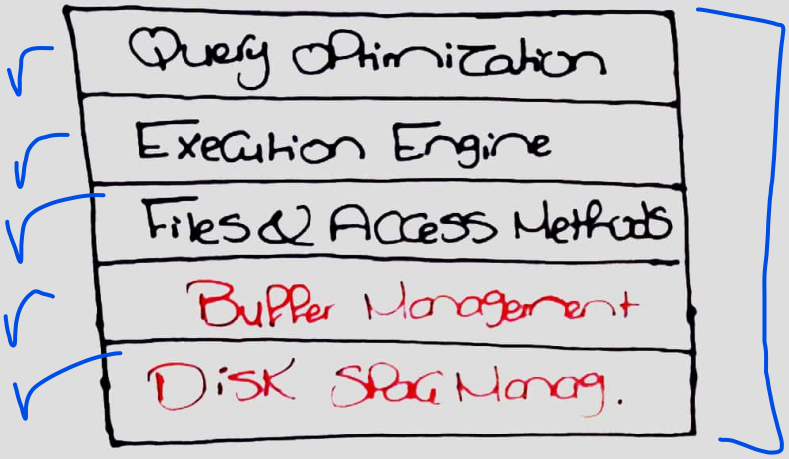
Hence, total access time is given by

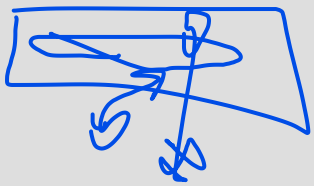
$$T_{Access} = t_{seek} + t_{Rot\ delay} + t_{transfer}$$

• Slow due to moving parts

- We assume that a random Page can be retrieved at a fixed cost (using average access time (avg. seek time and T/2 for rotation delay))
  - → Still true that reading consecutive Pages is much cheaper than reading them in random → implies consecutive blocks and thus records.
    - Seek and rotation time only once added.

- DBMS follows a layered architecture





1. Disk Space Management

→ Lowest layer of DBMS software
→ Called by higher layers to read/write or allocate/deallocate a Page
  - They only call the routines (no need to know how its done or how free space is managed)

2. Buffer Management

→ Brings Pages to main memory (by calling lower layer) if they aren't already there.
→ More importantly, decides which Pages in the main memory should be replaced to make room for new Pages
→ Uses a replacement Policy (such as FIFO)

- This is important because some queries (e.g., statistical) require going over many pages (or all). But at any time the main memory will never have enough room.

* Note that "File and access methods" (the layer above) can ask the buffer manager to access a page based on a record id (rid) it contains.

→ Even higher* levels just ask for records and files of records
- A file is a logical collection of pages (each with its records)

The DBMS is expected to support
- → insertion, deletion, modification of a record
- → Reading a particular record
- → Scanning all records (or a specific set)

(record id should be sufficient)

- The way such files are organized on external storage is called File Organization

More next lecture: As we'll see next lecture, sometimes storing an extra index datastructure based on some index search key can speed up specific queries
- → Index Search Key doesn't have to be a Primary, Foreign, Candidate or SuperKey (can be any other column)

- Many File Organization methods (alternatives) exist:
  - Heap Files (random order)
  - Sorted Files
  - Hash or tree-based indexes

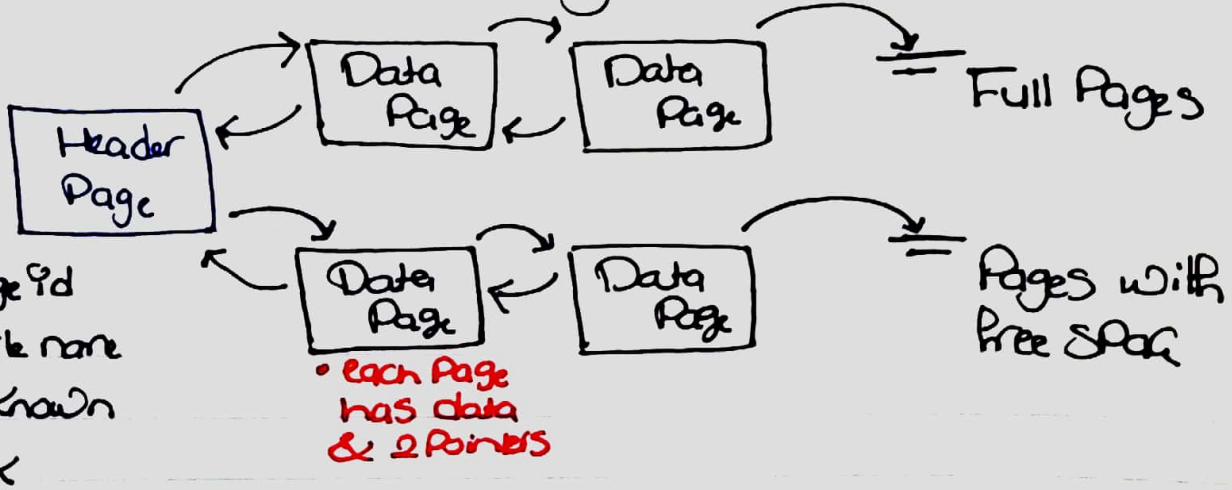
1. Heap Files (simplest)

→ Has nothing to do with Priority queues / heap tree (heap ↔ random)*

→ The records in the Pages of a heap file are not ordered in any way.

- To support record level operations, we need
  - → To keep track of Pages in a file
  - → To keep track of records in a Page
  - → To keep track of free space on Pages

- One way to achieve this is to implement it as two doubly-linked lists



- header Page id & heap file name stored at known loc. in disk
- Each Page has data & 2 Pointers

notice that Search time Per Page is const.

* To Search
  - → just go over all Pages
  - → avg. time taken is b/2 (b = # Pages in File)

Full → Pages with free space

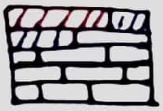
* To Insert
  → Start with the First Free Page
  → Keep going until you find a Page with enough free space to fit your record.
    → This can be a Problem especially when records have varying fields and thus are of varying lengths
    - may need to scan several Pages in that case
    → allocate a new one on the disk (and add its Pointer) if there's none with enough space (and deallocate if 100% free).

* Generally good when we tend to retrieve all the data (e.g. for statistics)

- Note: there are multiple methods to arrange the records themselves on a Page
  - Page contains a collection of slots (one for each record)
  - Each record is hence identified by <Page_id, Slot_num> = rid
  → Depend on whether data is made of fixed or varying length records
  - In each case there are different options for

How fields are organized in a record (Record Format)

- e.g., if they are varying we can decide to put columns next to each other & use delimeters (to separate)



How to organize the records within a page (Page Format)

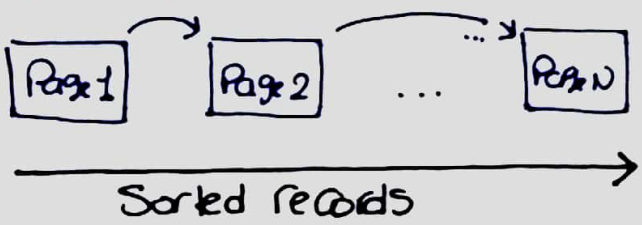
- whether to span or unspan
  + Don't allow records to span more than one page
  - can lead to unused space

2. Ordered Files (Sequential Files)

- Defined by an ordering field (records are always sorted w.r.t of it)

~~Does not involve data structure overhead~~

→ gives you quick search ($\log_2 b$ for b pages in file) (binary S.)



- less data structure overhead

- Clearly, insertion can be a headache (search for insertion position, change pointers after inserting page)

→ Solutions

1. Maintain an overflow file and keep insertions there and merge periodically (searching done on both)

* Use ordered Files when data is not expected to change frequently (class roster).

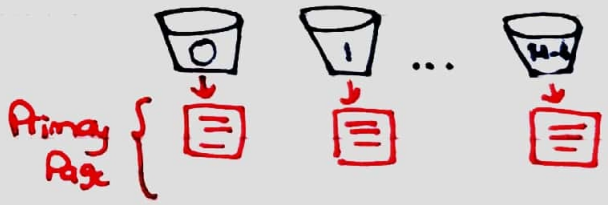
- Notice that sequential scan still takes $\frac{b}{2}$ even if file is ordered.

3. Hashed Files

(called external hashing)
- i.e., for disk files

- Choose a field in the data to be a hash key

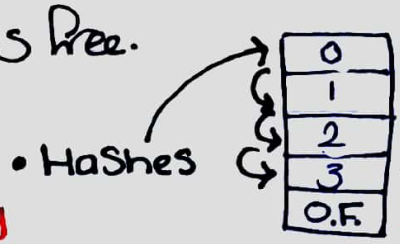
→ Divide file blocks into M ones of equal size



- For record in data:
  h = K mod M # K is chosen field
  insert_at_bucket(h) # goes to Primary Page

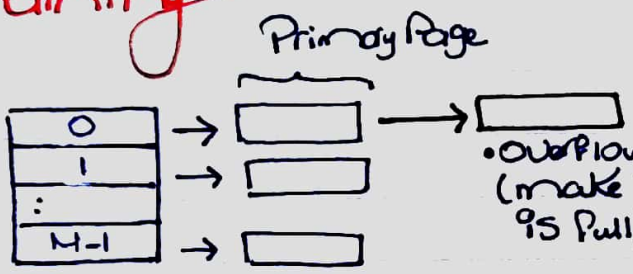
O(1) {
- They clearly speed up search (when its in the form .... WHERE Chosen_Field='...')
  ↖ hash & search in the relevant bucket.
- Same applies to insertion (hash Chosen Field)
  * Nothing special for other fields

- When a collision occurs (a record hashes to a bucket that's already full)
  - Open addressing
    → Keep considering the next bucket until you find one that's free.



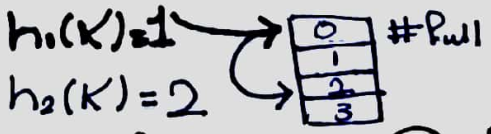
    * in this case, it's called linear probing

  - Chaining



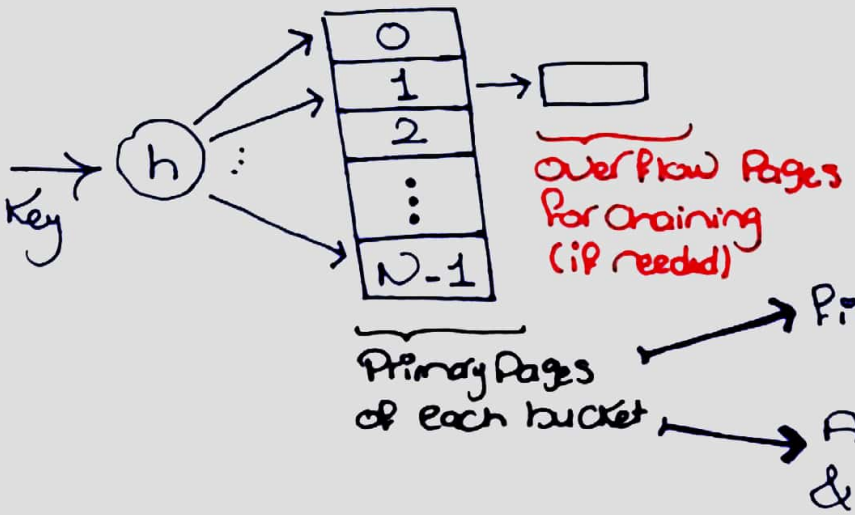
    • more complex versions might exist

  - Multiple Hashing
    - instead of choosing the bucket based on $h_1(K)$ everytime, if it results in a collision use another hash function $h_2(K)$ as an offset value



      → if there's a collision at the bucket ③ then we can use a 3rd hash function as offset or often use the 2nd again (open addressing in general)

Static Hashing



* Need the hash function to distribute keys over the range 0, 1, ..., N-1

$$h(K) = (aK + b) \% N$$

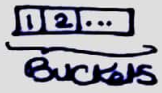
a, b → Chosen Constants (to tune h)

Works good for that.

- Advantages
  - → # Buckets are fixed and known → Stored Sequentially

Thus (assuming no overflows):

1. Search takes 1 I/O (read from disk)



2. Insertion takes 2 I/O
   (read from disk → insert → write to disk)
   ① ②

// Only Count disk IOs
(insertion is on main memory)

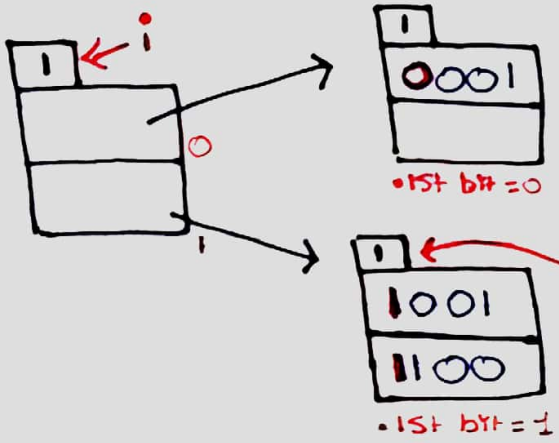
- Same applies for deletion

- Disadvantages
  - → If File Shrinks, can't make use of the allocated disk space that's free
  - → A lot of Collisions (depends on h, K) can lead to long overflow chains (data Skew)
    - Search/ insertion no longer takes const. time
- Solutions
  - → Assuming file doesn't grow much, just try to assure that Pages are 80% full initially (Prevent overflow)
  - → When overflow Pages build up, Periodically rehash (i.e., increase* N (no. of buckets) then reapply the hash function)
  - → Use dynamic hashing

Extendible Hashing (type of dynamic hashing)

- → Buckets (Primary Pages) are not fixed.
- One idea to implement is to reorganize the file by doubling the no. of buckets when an overflow will happen.
  - → need to read Previous configuration, double buckets and write again (rehash) . expensive operation

Rather, use a directory (list) of pointers to buckets



① * hash function has typically takes form $h_i = h_k \% 2^i$

→ Thus, to decide bucket look at last(i) bits (global depth)

"local depth" = no. of bits to look at for this bucket.

① unlike book, slides takes first (i) bits

→ Here h_k is 4 bits (clearly)
→ Each bucket holds 2 keys

Algorithm:

- if an overflow is about to happen upon insertion
  1. Make another bucket besides the one that was about to overflow and increase their local depth by 1.
     - Redistribute entries on both according to new local depth.

  2. If ① requires that the directory size is doubled then do that and fix pointers

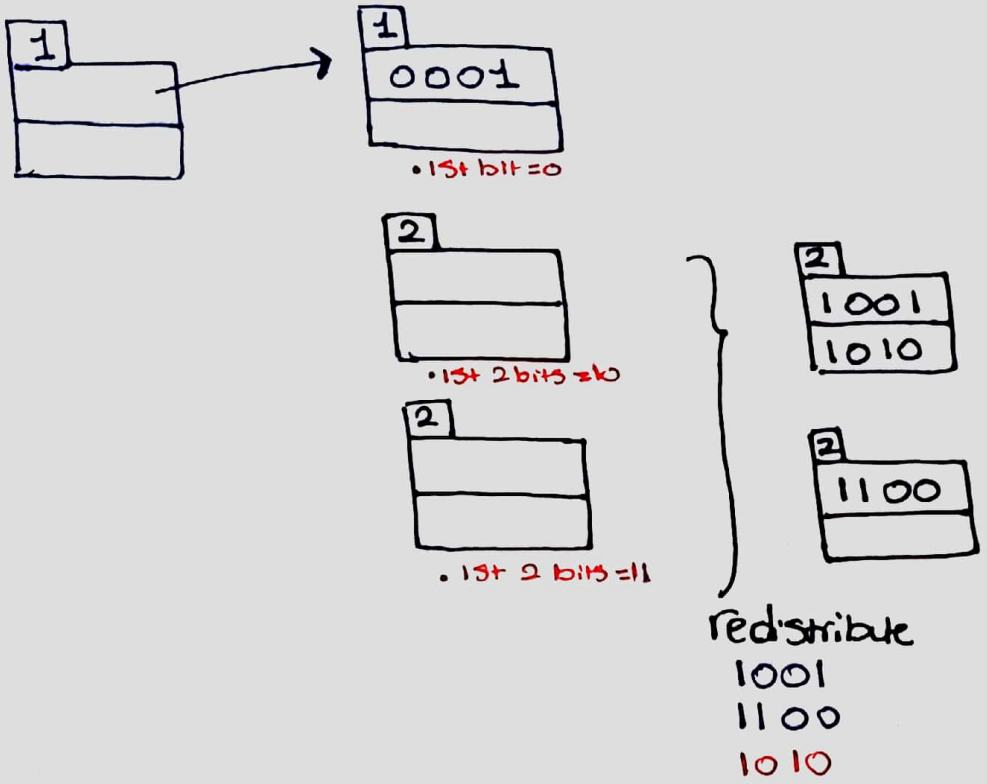
* Let's try to insert '1010' for the previous setup

→ 1st bit is 1 so it should go to 2nd bucket
→ will overflow!

This,

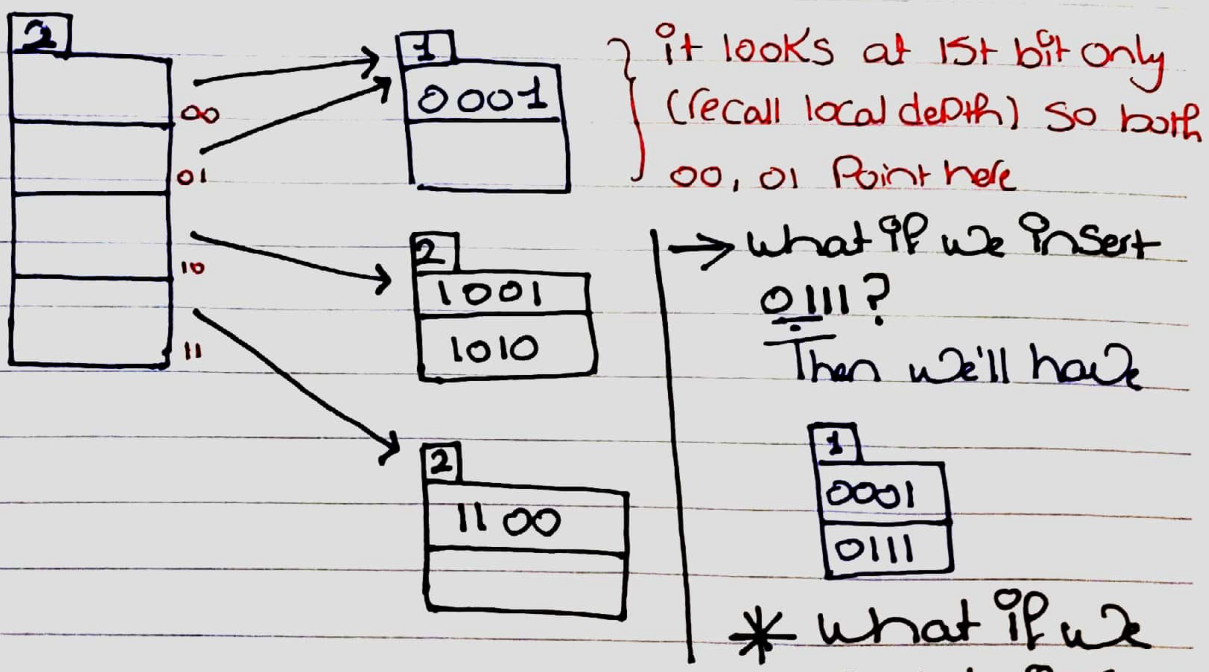
Called 'Split Image'

→ let's make a new bucket and let both have increased local depth.

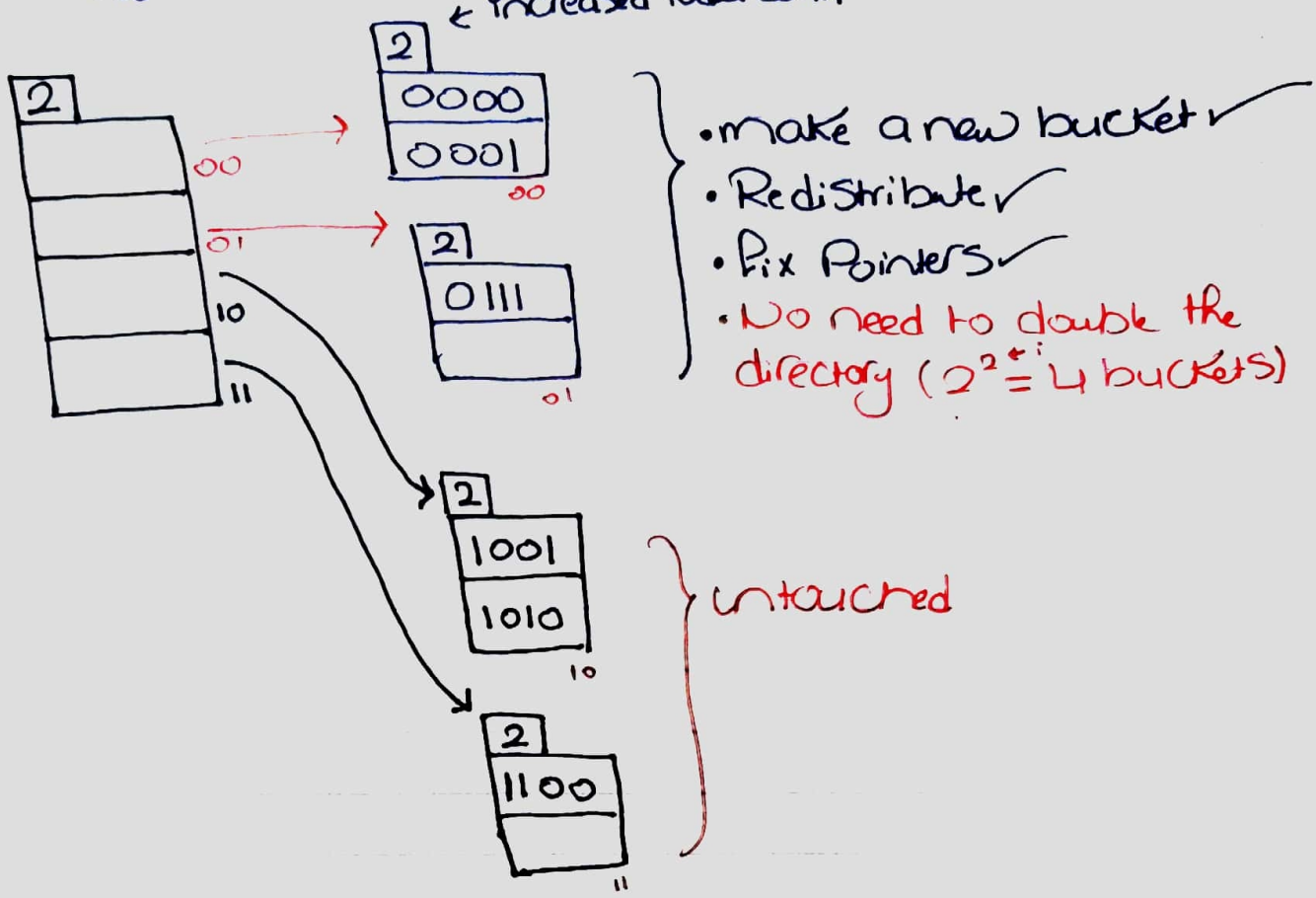


→ Now clearly, a directory of i = 1 can't correspond to 3 buckets so we need to double i

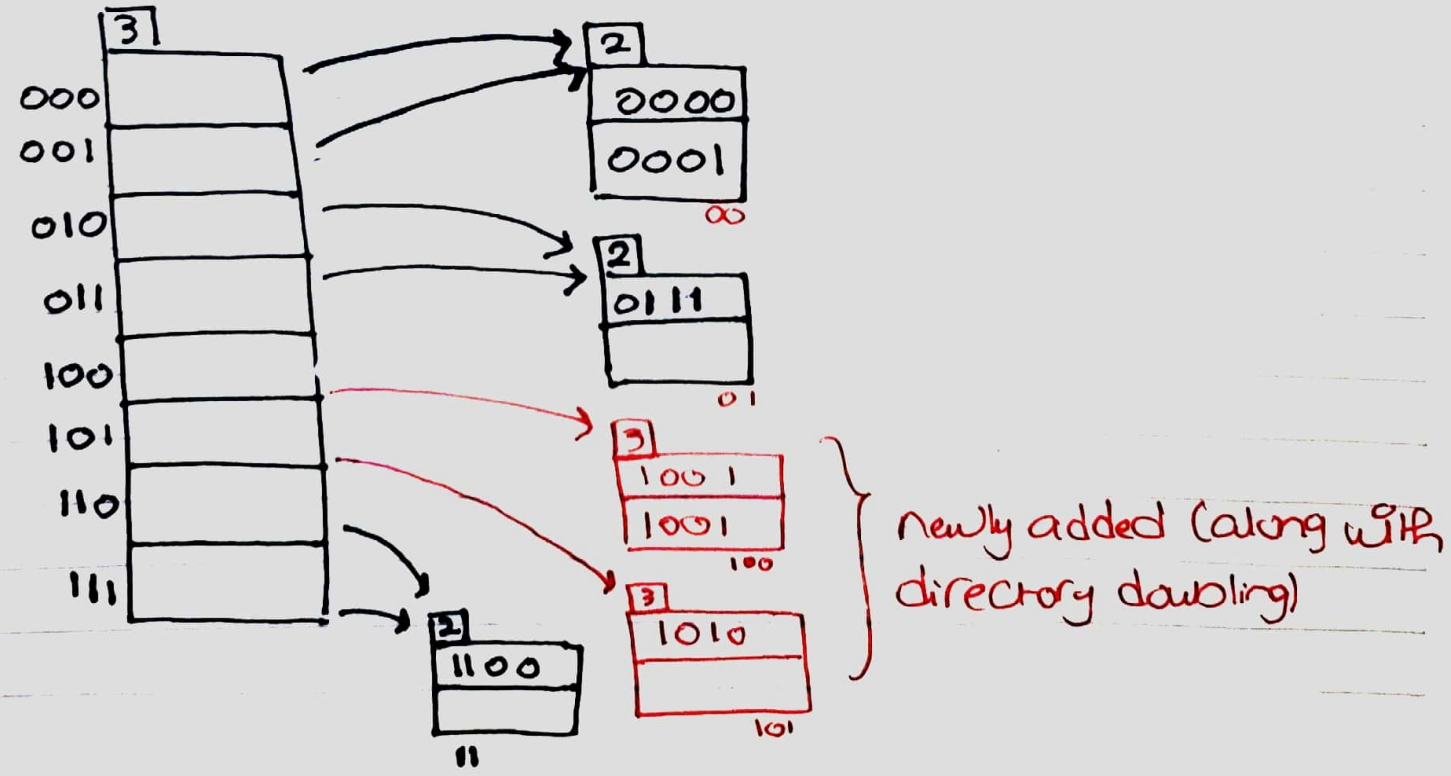
- This will be true whenever local depth of some bucket(s) > global depth (i)



→ 1st bucket will overflow, let's make another as well



• Now suppose we want to insert 1001 (again)
→ Clearly need another bucket (3rd will overflow)
→ Clearly will need to double directory



- Comments on extendible hashing

→ one disk access for sure when doing equality search (WHERE = ' ')
  - Because No overflows is guaranteed
  - assuming directory fits in main mem.

→ Sudden / quick growth can occur due to many collisions (skewed hash values)
  - entries with same hash aggravate the problem

* Note

→ upon deletion, an empty bucket can be merged with its split image
  → The directory can be halved again if all directory elements point to both bucket and its split image (intuitive).

→ Dynamic hashing also possible via a tree data structure (list of pointers at leaves and path to determine the binary corresponding to leave element)