CSE 544: Lectures 13 and 14
Storing Data, Indexes

Monday, 5/10/2004

Wednesday, 5/12/2004

Outline

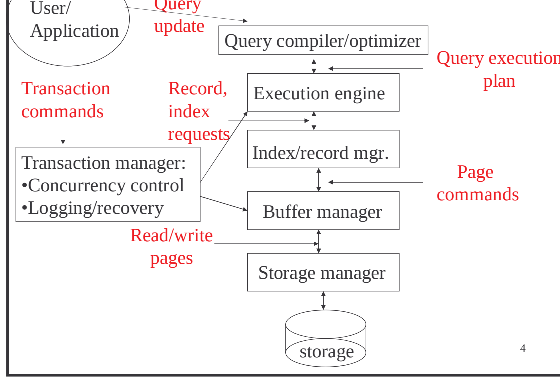
- Overview of a RDBMS
- Storing data: disks and files - Chapter 9
- Types of Indexes - Chapter 8.3
- B-trees - Chapter 10
- Hash-tables - Chapter 11

What Should a DBMS Do?

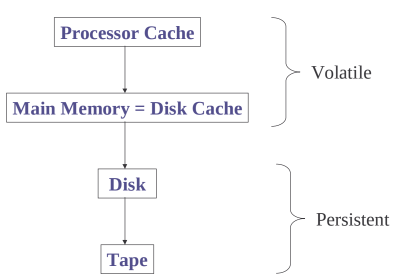
- Store large amounts of data
- Process queries efficiently
- Allow multiple users to access the database concurrently and safely.
- Provide durability of the data.

- How will we do all this??

Generic Architecture



The Memory Hierarchy



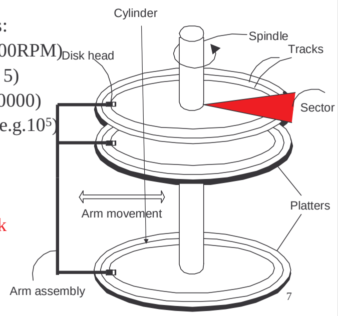
Disk

- The unit of disk I/O = ***block***
  - Typically 1 block = 4k
- Used with a main memory buffer

The Mechanics of Disk

Mechanical characteristics:
- Rotation speed (e.g. 7200RPM)
- Number of platers (e.g. 5)
- Number of tracks (<=10000)
- Number of bytes/track(e.g.10^5)

Important:
- Logical R/W unit: block
  typical 2KB - 32KB



7

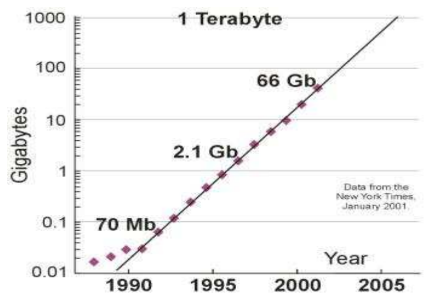
Important Disk Access Characteristics

Disk latency =
= seek time + rotational latency + transfer time

- Seek time:
  - e.g. min=2.2ms, max=15.5ms, avg=9.1ms
- Rotational latency:
  - e.g. avg = 4.17ms
- Transfer rate
  - E.g. 13MB/s

8

How Much Storage for $200



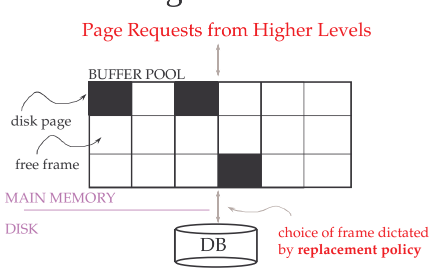
9

RAIDs

- = “Redundant Array of Independent Disks”
  - Was “inexpensive” disks
- Idea: use more disks, increase reliability
- Recall:
  - Database *recovery* helps after a systems crash, not after a disk crash
- 6 ways to use RAIDs. More important:
  - Level 4: use N-1 data disks, plus one parity disk
  - Level 5: same, but alternate which disk is the parity
  - Level 6: use Hamming codes instead of parity

10

Buffer Management in a DBMS



- Need a table of <frame#, pageid> pairs

Buffer Manager

- Page request --> read it in a free frame
- **pin_count** = how many processes requested it pinned
- **dirty** flag = if the page in the frame has been changed

- Replacement policies:
  - LRU, Clock, MRU, etc.
  - Only consider frames with **pin_count**=0

12

Buffer Manager

Why not use the Operating System for the task??

- DBMS may be able to anticipate access patterns
- Hence, may also be able to perform prefetching
- DBMS needs the ability to force pages to disk.

Managing Free Blocks

- By the OS
- By the RDBMS (typical: why ?)
  - Linked list of free blocks
  - Bit map

Files of Records

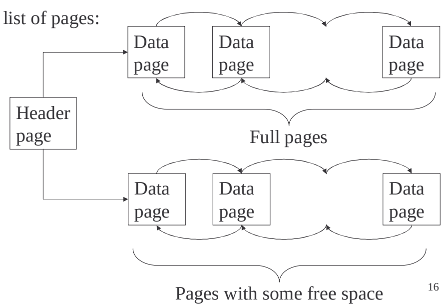
Types of files:

- Heap file - unordered
- Sorted file
- Clustered file - sorted, plus a B-tree

Will discuss heap files only; the others are similar, only sorted by the key

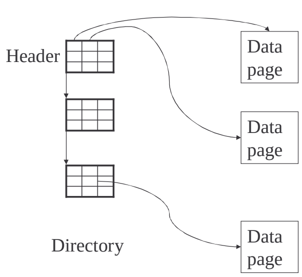
Heap Files

Linked list of pages:



Heap Files

Better: directory of pages



Page Formats

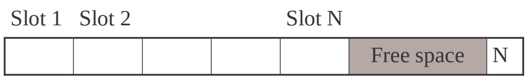
Issues to consider

- 1 page = fixed size (e.g. 8KB)
- Records:
  - Fixed length
  - Variable length
- Record id = RID
  - Typically RID = (PageID, SlotNumber)

Why do we need RID's in a relational DBMS ?

Page Formats

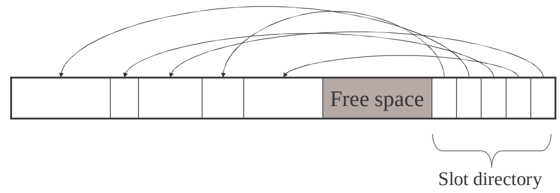
Fixed-length records: packed representation



Problems ?

19

Page Formats

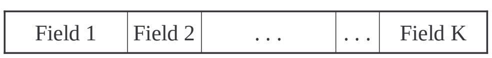


Variable-length records

20

Record Formats

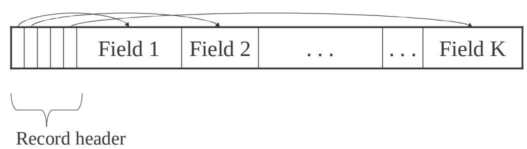
Fixed-length records --> all fields have fixed length



21

Record Formats

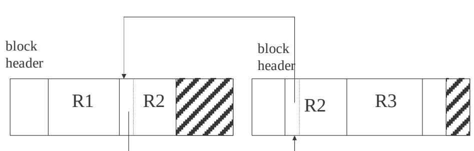
Variable length records



Remark: NULLS require no space at all (why ?)

22

Spanning Records Across Blocks



- When records are very large
- Or even medium size: saves space in blocks
- Commercial RDBMS avoid this

23

LOB

- Large objects
  - Binary large object: BLOB
  - Character large object: CLOB
- Supported by modern database systems
- E.g. images, sounds, texts, etc.
- Storage: attempt to cluster blocks together

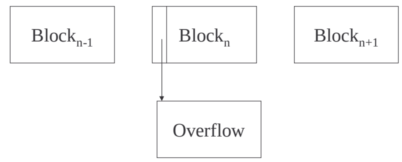
24

Modifications: Insertion

- File is unsorted (= ***heap file***)
  - add it to the end (easy J)
- File is sorted:
  - Is there space in the right block ?
    - Yes: we are lucky, store it there
  - Is there space in a neighboring block ?
    - Look 1-2 blocks to the left/right, shift records
  - If anything else fails, create *overflow block*

25

Overflow Blocks



- After a while the file starts being dominated by overflow blocks: time to reorganize

26

Modifications: Deletions

- Free space in block, shift records
- Maybe be able to eliminate an overflow block
- Can never really eliminate the record, because others may point to it
  - Place a tombstone instead (a NULL record)

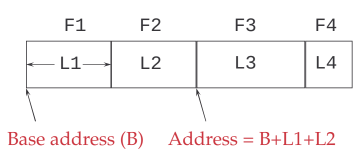
27

Modifications: Updates

- If new record is shorter than previous, easy J
- If it is longer, need to shift records, create overflow blocks

28

Record Formats: Fixed Length



- Information about field types same for all records in a file; stored in *system catalogs.*
- Finding *i'th* field requires scan of record.
- **Note the importance of schema information!**

Indexes

- **Search key** = can be any set of fields
  - not the same as the primary key, nor a key
- **Index** = collection of data entries
- **Data entry** for key k can be:
  - The actual record with key k
  - (k, RID)
  - (k, list-of-RIDs)

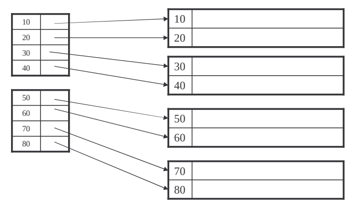
Index Classification

- Primary/secondary
  - Primary = may reorder data according to index
  - Secondary = cannot reorder data
- Clustered/unclustered
  - Clustered = records close in the index are close in the data
  - Unclustered = records close in the index may be far in the data
- Dense/sparse
  - Dense = every key in the data appears in the index
  - Sparse = the index contains only some keys
- B+ tree / Hash table / …

31

Primary Index

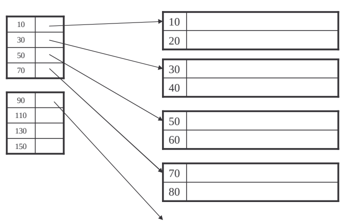
- File is sorted on the index attribute
- *Dense* index: sequence of (key,pointer) pairs



32

Primary Index

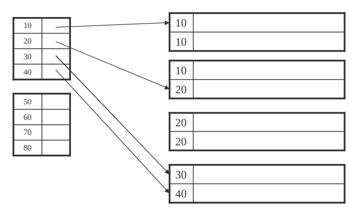
- *Sparse* index



33

Primary Index with Duplicate Keys

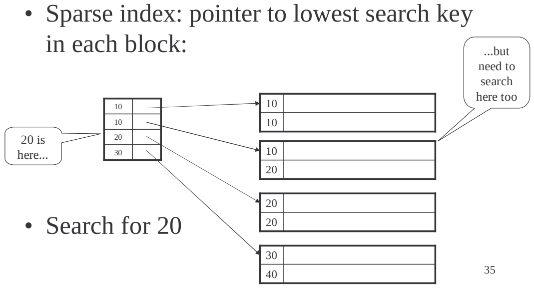
- Dense index:



34

Primary Index with Duplicate Keys

- Sparse index: pointer to lowest search key in each block:

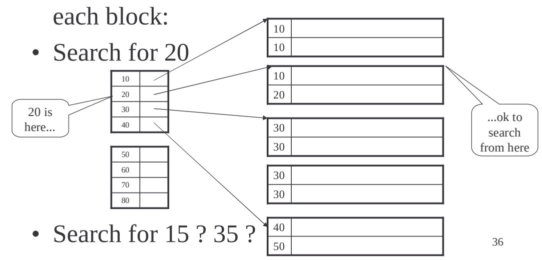


- Search for 20

35

Primary Index with Duplicate Keys

- Better: pointer to ***lowest new search key*** in each block:
- Search for 20

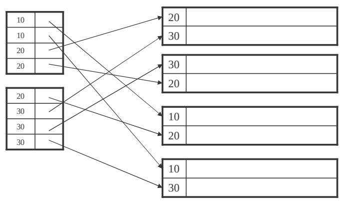


- Search for 15 ? 35 ?

36

Secondary Indexes

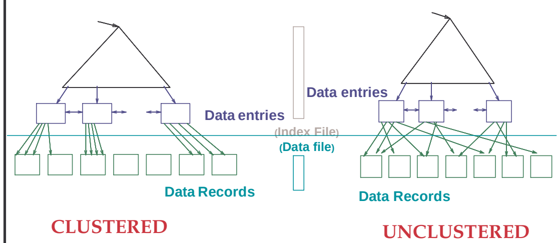
- To index other attributes than primary key
- Always dense (why ?)



Clustered/Unclustered

- Primary indexes = usually clustered
- Secondary indexes = usually unclustered

Clustered vs. Unclustered Index



Secondary Indexes

- Applications:
  - index other attributes than primary key
  - index unsorted files (heap files)
  - index clustered data

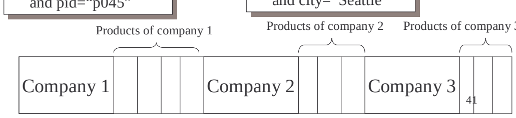
Applications of Secondary Indexes

- Secondary indexes needed for *heap files*
- Also for *Clustered data:*

Company(name, city), Product(pid, maker)

```
Select city
From Company, Product
Where name=maker
    and pid="p045"
```

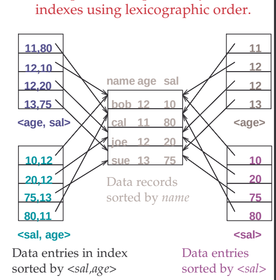
```
Select pid
From Company, Product
Where name=maker
    and city="Seattle"
```



Composite Search Keys

- *Composite Search Keys*: Search on a combination of fields.
  - Equality query: Every field value is equal to a constant value. E.g. wrt <sal,age> index:
    - age=20 and sal =75
  - Range query: Some field value is not a constant. E.g.:
    - age =20; or age=20 and sal > 10

Examples of composite key indexes using lexicographic order.

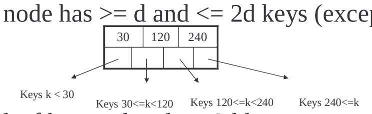


B+ Trees

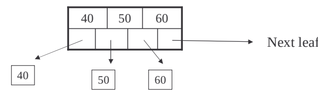
- Search trees
- Idea in B Trees:
  - make 1 node = 1 block
- Idea in B+ Trees:
  - Make leaves into a linked list (range queries are easier)

B+ Trees Basics

- Parameter d = the *degree*
- Each node has >= d and <= 2d keys (except root)

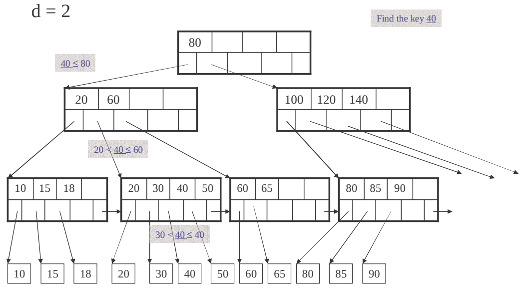


- Each leaf has >=d and <= 2d keys:



B+ Tree Example

d = 2



Searching a B+ Tree

- Exact key values:
  - Start at the root
  - Proceed down, to the leaf

```
Select name
From people
Where age = 25
```

- Range queries:
  - As above
  - Then sequential traversal

```
Select name
From people
Where 20 <= age
  and  age <= 30
```


B+ Tree Design

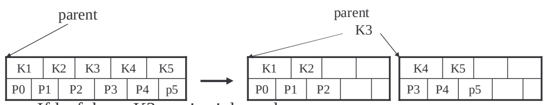
- How large d ?
- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 byes
- 2d x 4 + (2d+1) x 8 <= 4096
- d = 170

B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

Insertion in a B+ Tree

Insert (K, P)
- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:

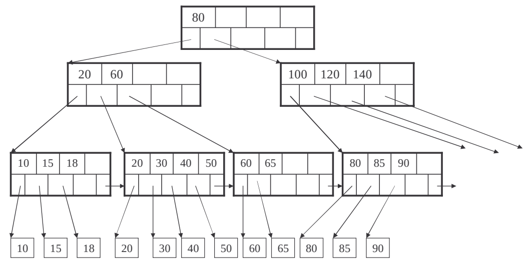


- If leaf, keep K3 too in right node
- When root splits, new root has 1 key only

49

Insertion in a B+ Tree

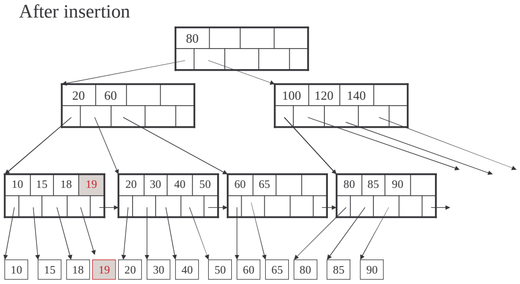
Insert K=19



50

Insertion in a B+ Tree

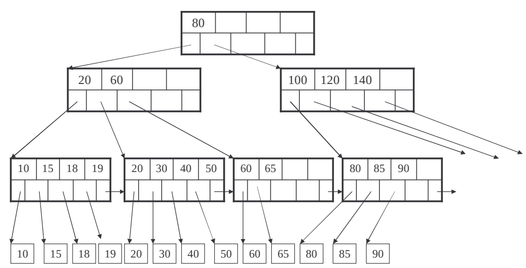
After insertion



51

Insertion in a B+ Tree

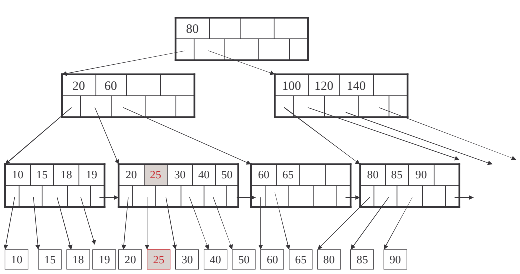
Now insert 25



52

Insertion in a B+ Tree

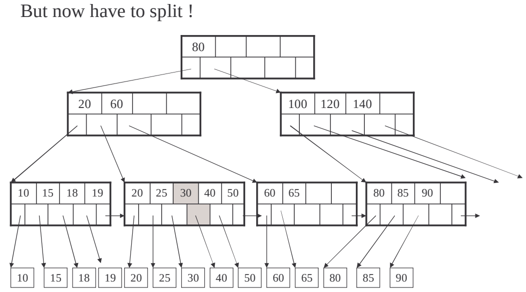
After insertion



53

Insertion in a B+ Tree

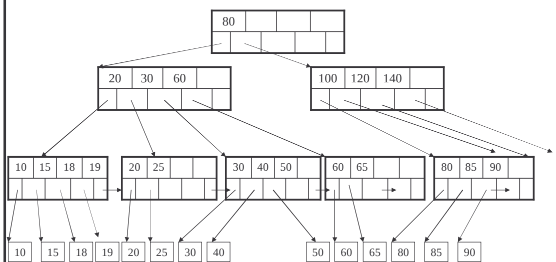
But now have to split !



54

Insertion in a B+ Tree

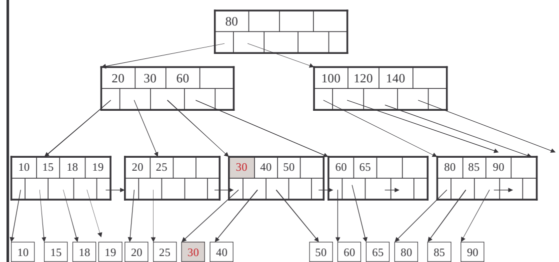
After the split



55

Deletion from a B+ Tree

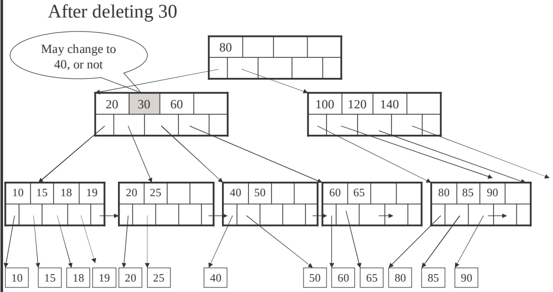
Delete 30



56

Deletion from a B+ Tree

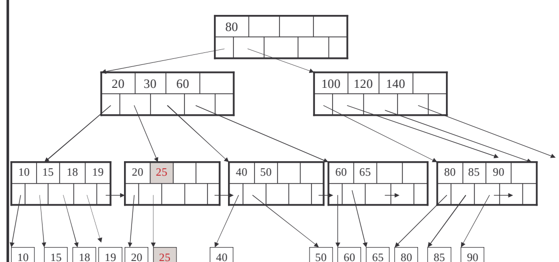
After deleting 30



57

Deletion from a B+ Tree

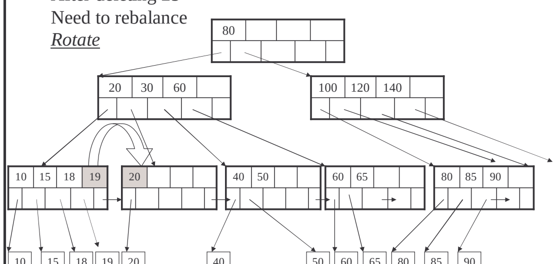
Now delete 25



58

Deletion from a B+ Tree

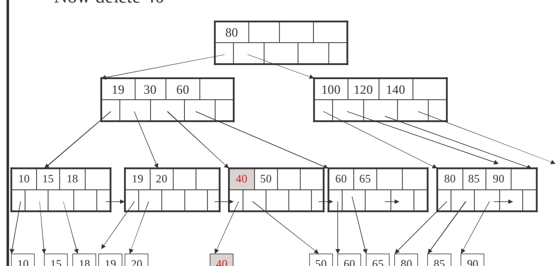
After deleting 25
Need to rebalance
Rotate



59

Deletion from a B+ Tree

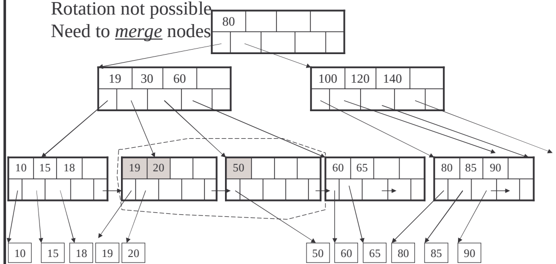
Now delete 40



60

Deletion from a B+ Tree

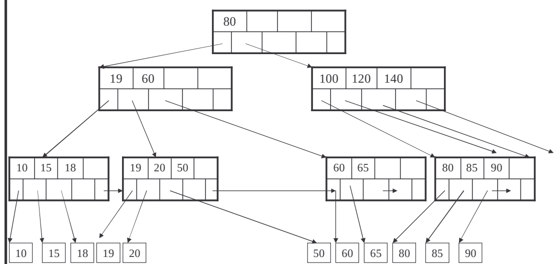
After deleting 40
Rotation not possible
Need to *merge* nodes



61

Deletion from a B+ Tree

Final tree



62

In Class

- Suppose the B+ tree has depth 4 and degree d=200
- How many records does the relation have (maximum) ?
- How many index blocks do we need to read and/or write during:
  - A key lookup
  - An insertion
  - A deletion

63

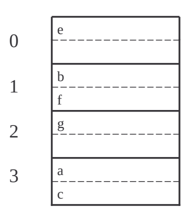
Hash Tables

- Secondary storage hash tables are much like main memory ones
- Recall basics:
  - There are n *buckets*
  - A hash function f(k) maps a key k to {0, 1, …, n-1}
  - Store in bucket f(k) a pointer to record with key k
- Secondary storage: bucket = block, use overflow blocks when needed

64

Hash Table Example

- Assume 1 bucket (block) stores 2 keys + pointers
- h(e)=0
- h(b)=h(f)=1
- h(g)=2
- h(a)=h(c)=3

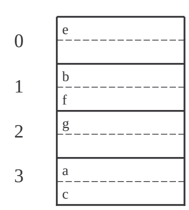


Here: h(x) = x mod 4

65

Searching in a Hash Table

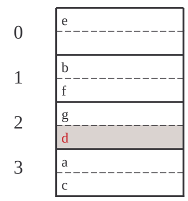
- Search for a:
- Compute h(a)=3
- Read bucket 3
- 1 disk access



66

Insertion in Hash Table

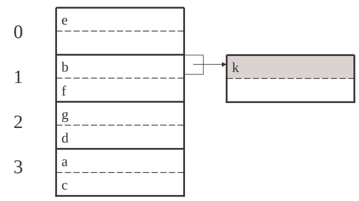
- Place in right bucket, if space
- E.g. h(d)=2



67

Insertion in Hash Table

- Create overflow block, if no space
- E.g. h(k)=1



- More overflow blocks may be needed

68

Hash Table Performance

- Excellent, if no overflow blocks
- Degrades considerably when number of keys exceeds the number of buckets (I.e. many overflow blocks).

69

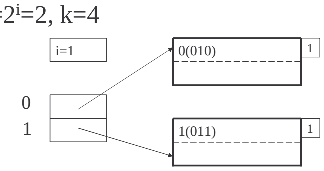
Extensible Hash Table

- Allows has table to grow, to avoid performance degradation
- Assume a hash function h that returns numbers in $\{0, \ldots, 2^k - 1\}$
- Start with $n = 2^i << 2^k$, only look at first i most significant bits

70

Extensible Hash Table

- E.g. i=1, $n=2^i=2$, k=4

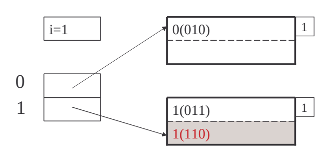


- Note: we only look at the first bit (0 or 1)

71

Insertion in Extensible Hash Table

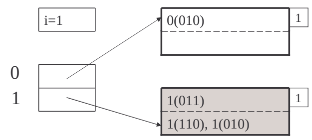
- Insert 1110



72

Insertion in Extensible Hash Table

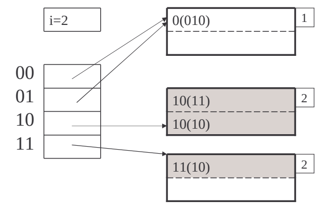
- Now insert 1010



- Need to extend table, split blocks
- i becomes 2

73

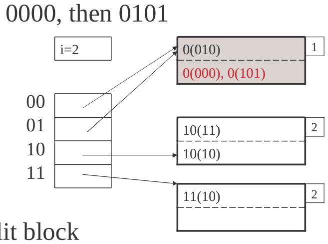
Insertion in Extensible Hash Table



74

Insertion in Extensible Hash Table

- Now insert 0000, then 0101

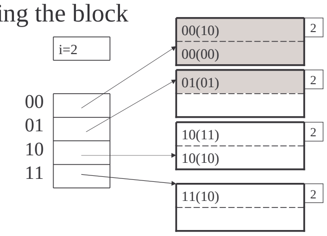


- Need to split block

75

Insertion in Extensible Hash Table

- After splitting the block



76

Extensible Hash Table

- How many buckets (blocks) do we need to touch after an insertion ?
- How many entries in the hash table do we need to touch after an insertion ?

77

Performance Extensible Hash Table

- No overflow blocks: access always one read
- BUT:
  - Extensions can be costly and disruptive
  - After an extension table may no longer fit in memory

78

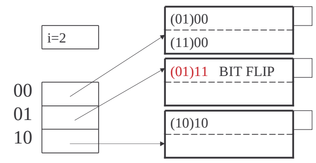
Linear Hash Table

- Idea: extend only one entry at a time
- Problem: n= no longer a power of 2
- Let i be such that $2^i <= n < 2^{i+1}$
- After computing h(k), use last i bits:
  - If last i bits represent a number > n, change msb from 1 to 0 (get a number <= n)

79

Linear Hash Table Example

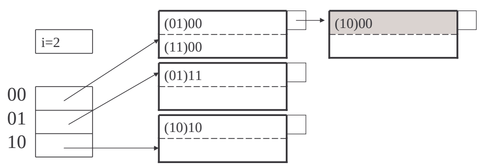
- n=3



80

Linear Hash Table Example

- Insert 1000: overflow blocks…



81

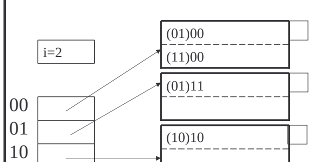
Linear Hash Tables

- Extension: independent on overflow blocks
- Extend n:=n+1 when average number of records per block exceeds (say) 80%

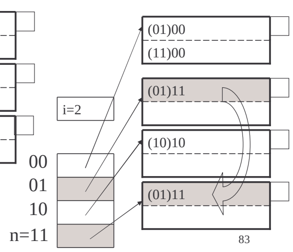
82

Linear Hash Table Extension

- From n=3 to n=4



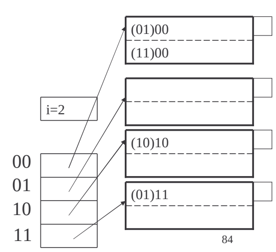
- Only need to touch one block (which one ?)



83

Linear Hash Table Extension

- From n=3 to n=4 finished
- Extension from n=4 to n=5 (new bit)
- Need to touch every single block (why ?)



84