- ✓ Track Capacity = Number Of Sectors Per Track * Bytes Per Sector
- ✓ Cylinder Capacity = Number Of Tracks Per Cylinder * Track Capacity
- ✓ Driver Capacity = Number Of Cylinder * Cylinder Capacity
- ✓ Access Time = Seek Time + Rotational Delay
- ✓ Time To Read = Seek Time + Rotational Delay + Transfer Time
- ✓ Transfer Time = (Number Of Sectors Transferred / Number Of Sectors On a Track) * Rotation Time

- 20 plate, 800 track/plate, 25 sector/track, 512 byte/sector, 3600 rpm
- 7 ms from track to track, 28 ms avg. seek time, 50 msec max seek time

Avg. Rotational time	=1/2 * (60000/3600) = 16.7/2 = 8.3msec
Disk capacity	= 25 * 512 * 800 * 20 = 204.8 MB
Time to read track	= 1 rotation = 16.7 msec
Time to read cylinder	= 20 * 16.7 = 334msec
Time to read whole disc	= 800 * time to read cylinder + 799* time to pass from one cylinder to another =800*334 + 799*7msec =267 sec + 5.59 sec = 272.59 sec

- Disc specs: avg. seek time 8 msec, 10.000 rpm, 170 sector/track, 512 byte/sector

Time to read one sector = avg. seek time + rotational delay + time to transfer one sector
= 8 + (0.5 * 60.000 / 10.000) + (6* 1/170) = 8 + 3+ 0.035 = 11.035 msec

Time to read 10 sequential blocks = avg. seek time + rotational delay + 10 * time to transfer one sector = 8 + 3 + 10 * 0.035 = 11.35 msec

Time to read random 10 block= 10 * (avg. seek time + rotational delay + time to transfer one sector)= 10 * (8 + 3 + 0.035) = 113.5 msec

You have total 200.000 records stored in heap file and length of each record is 250 byte.

Here are the specifications of disk used for storage: 512 byte/sector, 20 sector/track, 5 sector/cluster, 3000 track/platter, one sided total 10 plate, rotational time 7200 rpm, seek time 8 msec. Records do not span between sectors

- Blocking factor of heap file: 10
- Number of blocks to store whole file: 20.000 blocks
- Number of blocks to estimate disk capacity: 120.000 blocks
- Cylinder number to store the data with cylinder based approach : 500
- Time to read one block : 14,225 ms
- Time to read file from beginning to end (in worst case): 284.500sec, 4741,67 min, 79,02 hour

- ✓ **pin_count** is used to keep track of number of transactions that are using the page. Zero means nobody is using it.
- ✓ **dirty** is used as a flag (dirty bit) to indicate that a page has been modified since read from disk. Need to flush it to disk if the page is to be evicted from pool.

Page_no = 1 Pin_count = 3 Dirty = 1 Last Used: 12:34:05	Page_no = 2 Pin_count = 0 Dirty = 1 Last Used: 12:35:05	Page_no = 3 Pin_count = 1 Dirty = 0 Last Used: 12:36:05	Page_no = 4 Pin_count = 2 Dirty = 0 Last Used: 12:37:05	Page_no = 5 Pin_count = 0 Dirty = 0 Last Used: 12:38:05
Page_no = 6 Pin_count = 0 Dirty = 0 Last Used: 12:29:05	Page_no = 7 Pin_count = 1 Dirty = 1 Last Used: 12:20:05	Page_no = 8 Pin_count = 0 Dirty = 1 Last Used: 12:40:05	Page_no = 9 Pin_count = 2 Dirty = 0 Last Used: 12:27:05	Page_no = 10 Pin_count = 0 Dirty = 1 Last Used: 12:39:05

Which page should be removed if LRU is used as the policy:.................6.......................

Which page should be removed if MRU is used as the policy :...............8...............

Which pages do not need to be written to disc, if it is removed:...............5,6...........

Which pages could not be removed in this situation:.....P.c.. = 0.................

- ✓ In general, (bf is blocking factor. N is the size of the file in terms of the number of records) :
- ✓ At least 1 block is accessed (I/O cost : 1)
- ✓ At most N/bf blocks are accessed.
- ✓ On average N/2bf.
- ✓ Time To Fetch One Record = (N/2bf) * Time To Read One Block
- ✓ Time To Read One Block = Seek Time + Rotational Delay + Block Transfer Time

- ✓ Time To Read All Records = N/bf * Time To Read Per Block
- ✓ Time To Add New Record = Time To Read One Block (For Last Block) + Time To Write One Block (For Last Block)
- ✓ If the last block is full; Time To Add New Record= Time To Read One Block (For Last Block) + Time To Write New One Block (For New Last Block)
- ✓ Time To Update One Fixed Length Record = Time To Fetch One Record + Time To Write One Block
- ✓ Time To Update One Variable Length Record = Time To Delete One Record + Time To Add New Record

- FileA: 10000 records , BF = 100, 4 extents
- File B: 5000 records, BF = 150, 3 extents

- Time to find the number of common records of FileA and B

  Time to read FileA= 4 * (seek time + rotational delay) + (10000/100) * block transfer time

  Time to read FileB = 3 * (seek time + rotational delay) + (5000/150) * block transfer time

  = Time to read FileA + 100 * Time to read FileB

  (imagine you've got only two frames in the buffer pool.)
- Read FileA and compare each record of FileA with whole records in FileB

- We can do binary search (assuming fixed-length records) in the sorted part.

m records	k records	
Sorted part	overflow	(m + k = N)

- Worst case to fetch a record :

$$T_F = \log_2 (m/bf) \ * \text{time to read per block.}$$

- If the record is not found, search the overflow area too. Thus total time is:

$$T_F = \log_2 (m/bf) \ * \text{time to read per block} + k/bf * \text{time to read per block}$$

Dense & Sparse Index:

Dense and Sparse Indices

1. There are Two types of ordered indices:

   Dense Index:

   - An index record appears for **every** search key value in file.
   - This record contains search key value and a pointer to the actual record.

   Sparse Index:

   - Index records are created only for **some** of the records.
   - To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.
   - We start at that record pointed to by the index record, and proceed along the pointers in the file (that is, sequentially) until we find the desired record.
2. Figures 11.2 and 11.3 show dense and sparse indices for the deposit file.

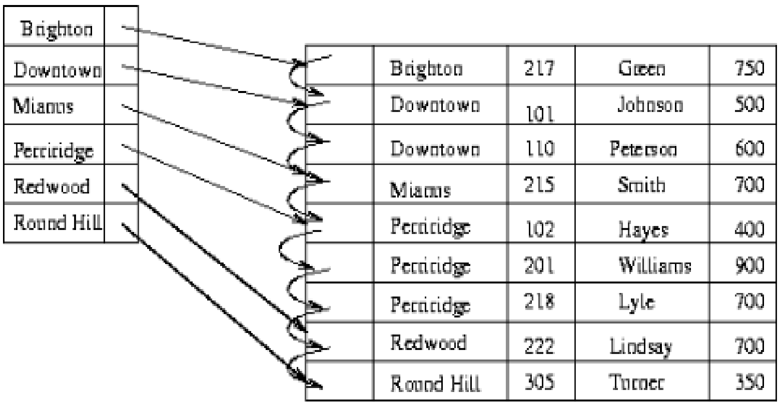


Figure 11.2: Dense index.

3. Notice how we would find records for Perryridge branch using both methods. (Do it!)

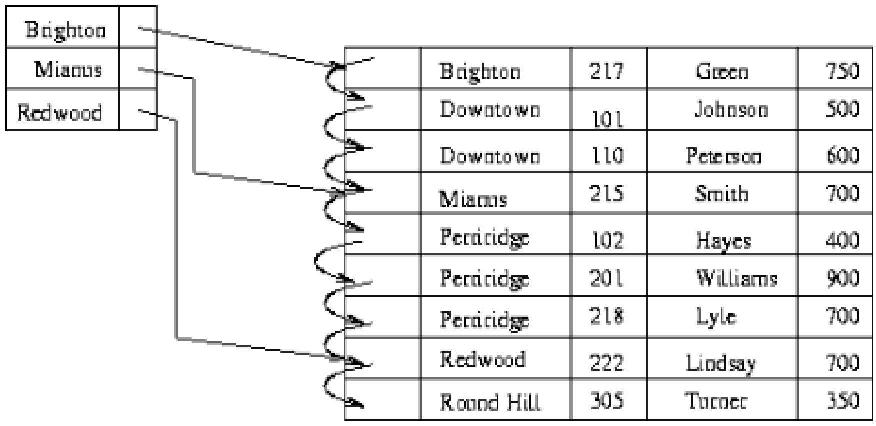


Figure 11.3: Sparse index.

4. Dense indices are faster in general, but sparse indices require less space and impose less maintenance for insertions and deletions. (Why?)
5. A good compromise: to have a sparse index with one entry per block.

Why is this good?

- Biggest cost is in bringing a block into main memory.
- We are guaranteed to have the correct block with this method, unless record is on an overflow block (actually could be **several** blocks).
- Index size still small.

- Suppose there is a data file of 4 GB (2^{32}) in a system with blocks of 1KB and fixed length records of 256Bytes.
- The records are stored in sorted order with respect to the key Student ID.
- Index stores a search key of 4 Bytes and a 4 Bytes of pointer. So, an index entry is 8 bytes.
- How many disk accesses do we need to find a record with a given Student ID:
  - Using sorted data file
  - Using dense index
  - Using sparse index

- $2^{32} / 2^{10} = 2^{22}$ blocks are in the data file.
- Blocking Factor of data file = $2^{10}/2^{8} = 2^{2}$
- 2^{22} x BF of data file = 2^{24} records in the file
- With binary search we can have 22 disk accesses to find the record we are searching for in the worst case.
- If an index entry is 8 bytes we can fit into a block $2^{10}/2^{3}=2^{7}$ entries
  - Dense Index should be: $2^{24}/ 2^{7}=2^{17}$ blocks = 2^{17} x 2^{10} = 2^{27} = 128MB index file. So, a binary search is 17 disk accesses.
  - Sparse Index should be: $2^{22}/2^{7}=2^{15}$ blocks = 2^{15} x 2^{10} = 2^{25} = 32MB index file. So, a search is 15 disk accesses.
- What would happen if we had a two-level sparse index?

Cost:

✓ B: The number of data pages. ✓ R: Number of records per page. ✓ D: (Average) time to read or write disk page.

	Heap File	Sorted File	Hashed File
Scan All Records	BD	BD	1.25 BD
Equality Search	0.5 BD	D log2B	D
Range Search	BD	D (log2B + # of pages with matches)	1.25 BD
Insert	2D	Search + BD	2D
Delete	Search + D	Search + BD	2D

Load Factor

- Loading factor (LF), α = n / m
  n: number of keys
  m: number of slots
- If uniform distribution (1/m) to get mapped to a slot, a slot will have an expectation of α elements.
- If m increases
  - Collision decreases
  - LF decreases
  - 0.5 > LF > 0.8 is unacceptable
  - Storage requirements increases.
- Reduce collisions while keeping storage requirements low.

Linear Probing

$$h(k,i) = (h'(k) + i) \bmod m$$

- Always check the next index
- Increments index linearly with respect to i.
- Clustering problem

hash(10) = 2
hash(5) = 5
hash(15) = 7

0	72	72	72	72
1				15
2	18	18	18	18
3	43	43	43	43
4	36	36	36	36
5		10	10	10
6	6	6	6	6
7			5	5

Open Addressing – Quadratic Probing

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

- Instead of moving by one, move i^2

$c_1=0$, $c_2=1$
hash(89)=9
hash(18)=8
hash(49)=9
hash(49, 1) = 0
hash(58) = 8
hash(58, 1) = 9
hash(58,2) = 2
hash(69) = 9
hash(69,1) = 0
hash(69,2) = 3

0			49	49	49
1					
2				58	58
3					69
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Example 2 – Double Hashing

	insert(76)	insert(93)	insert(40)	insert(47)	insert(10)	insert(55)
	76%7 = 6	93%7 = 2	40%7 = 5	47%7 = 5 5 - (47%5) = 3	10%7 = 3	55%7 = 6 5 - (55%5) = 5
0						
1				47	47	47
2		93	93	93	93	93
3					10	10
4						55
5			40	40	40	40
6	76	76	76	76	76	76
probes:	1	1	1	2	1	2

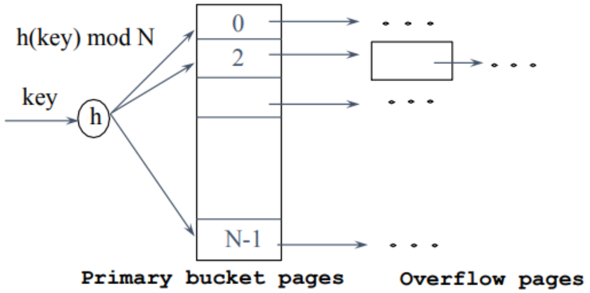
Closed Hashing >>> https://www.cs.usfca.edu/~galles/visualization/ClosedHash.html **Open Hashing >>> https://www.cs.usfca.edu/~galles/visualization/OpenHash.html**

Dynamic Hashing Methods

- *As for any index, 2 alternatives for data entries* **k***:
  - ☐ <**k**, rid of data record with search key value **k**>
  - ☐ <**k**, list of rids of data records with search key **k**>
  - Choice orthogonal to the *indexing technique*
- *Hash-based* indexes are best for *equality selections*. ***Cannot*** support range searches.

Static Hashing

- \# primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- **h**(*k*) mod M = bucket to which data entry with key *k* belongs. (M = # of buckets)



Extendible Hashing >>> https://www.youtube.com/watch?v=TtkN2xRAgv4&t=519s

Extendible Hashing Example