WEEK 3

ADDING RECORDS TO THE HARD DRIVE USING A HASH ORGANIZATION

STUDENT OBJECTIVES

- Upon completion of this video, you should be able to:
  - Explain how the records are added to the disk when using a hash organization
  - Given adding, modifying and deleting records, determine which operations are efficient and which operations are costly
  - Determine when an hash organization is appropriate
  - Given a number of records, record size and block size, figure out the average number of searches needed to find a record and the worst case scenario for searching for a given record

HASH ORGANIZATION

- **External Hashing**
  - Blocks are divided into M equal sized buckets → Bucket 0, Bucket 1, …Bucket M-1 (usually a bucket corresponds to 1 or a fixed number of blocks)
  - One field (attribute) is the hash key
  - The record with the hash key value K is stored in bucket i, where i=h(K) and h is the hashing function.

key.

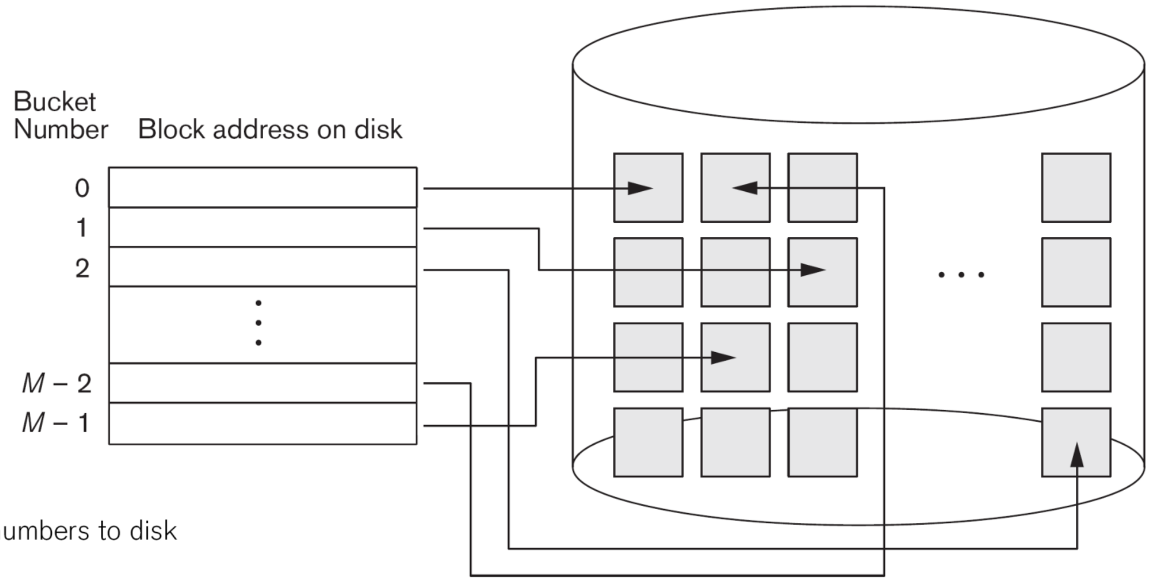


Figure 17.9
Matching bucket numbers to disk block addresses.

HASHING CONTINUED…

- A method of distributing data evenly (almost randomly) to different areas of memory
- Excellent if you need to get a record using its key field

credit card is a hashing.

- Credit card numbers are checked using a hashing function

A SAMPLE HASHING FUNCTION

- You want/need an even distribution, here is a good function to use with hashing:
  - Assume you have M buckets and you want to hash a key K to the bucket is K MOD M. This will give you a number between 0 and M-1, so label your M buckets with those numbers. Then, for each key, work out K MOD M and whatever number you get, put it into that bucket.

EXAMPLE:

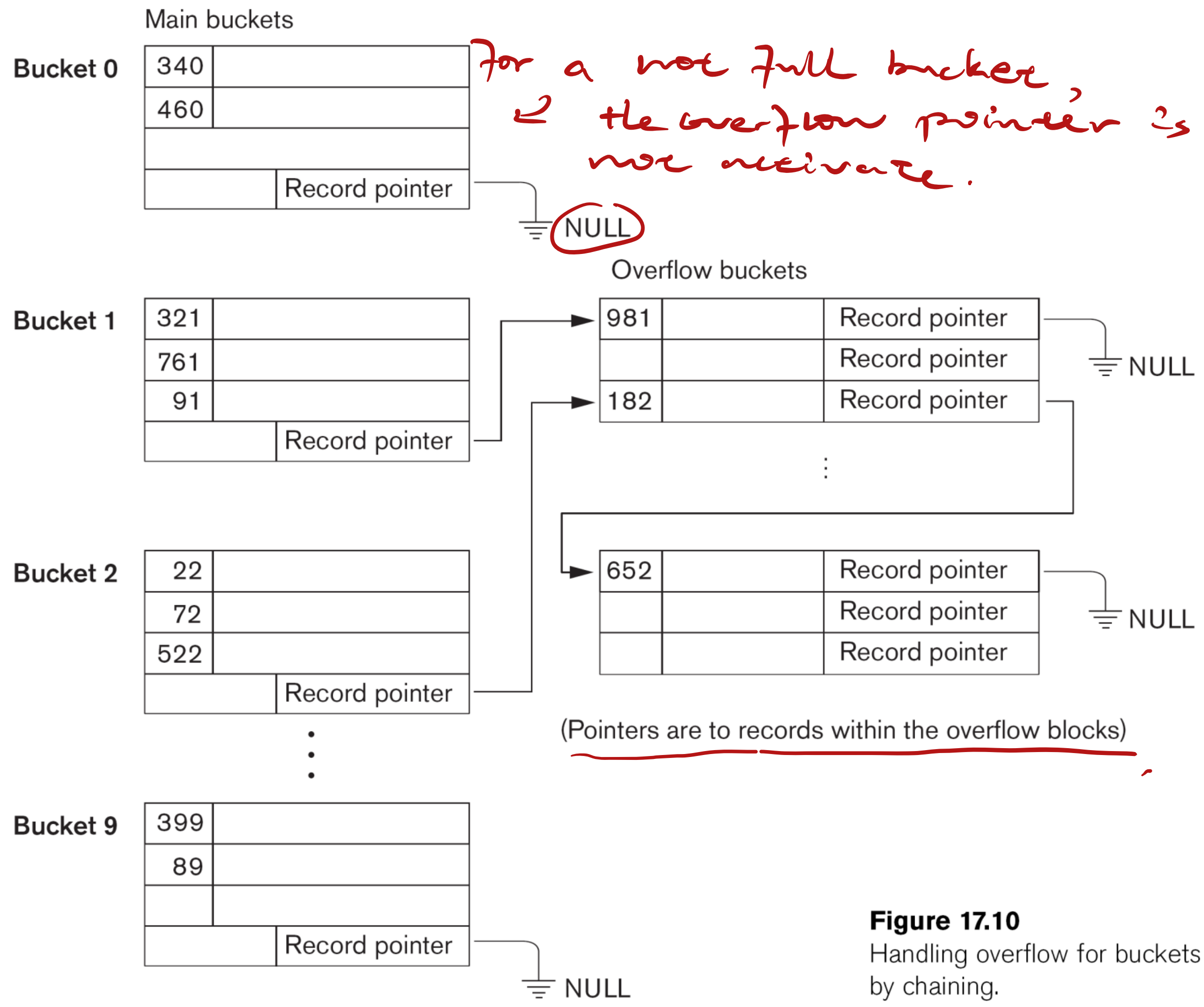
- QUESTION: Assume you have the records with the following values for their key attributes:
  34, 44, 22, 24, 23, 100, 46, 50, 32, 61 → NOTE: WE HAVE 10 RECORDS!
- You have 4 buckets, a bucket is 1024 bytes, each record is 333 bytes thus each bucket can hold __3__ records. <- this is the number that would contain in each bucket
- We have 10 records to put into the 4 buckets, where each bucket holds 3 records, DO WE HAVE ENOUGH SPACE? Yes, we SHOULD have enough space!
- Steps to hash these records into the bucket using the formula K MOD 4 (because of 4 buckets).

Bucket 0	44	24	100
Bucket 1			
Bucket 2	34	22	46 50
Bucket 3 m-1	23		

- QUESTION: What problem occurred here?

NO MORE ROOM!
Called a COLLISION

- General rule is to pick big enough slots so that all slots are always about 80% full, this will avoid most collisions
- Handling collisions (occurs when 2 records has to the same slot or Bucket is full):
  - **Open addressing:** Find the first open position following the position that is full   i.e. move 50 to bucket 3 . .
  - **Chaining:** Overflow area is kept and a pointer to the overflow area that is use
  - **Multiple Hashing:** another hash function is applied to the record if the first results in a collision
- For External Hashing on disks, only something based on ***Chaining*** would be used



For a not full bucket, the overflow pointer is not activate.

Figure 17.10
Handling overflow for buckets by chaining.

PROBLEMS WITH HASHING:

- If we want to order on the key that has been used for the hashing function, the records aren't in order
- Requires a fixed amount of space, for example if we have M buckets and each bucket holds m records then at most we can hold M*m records, but what if we have substantially fewer records? or substantially more records?
- Not good if we want to retrieve records in a range

then we have to go through all records, like a linear search.

- Not good when retrieval is based on an attribute other than the hashed one.

But it would response fast if we retrived based on hash key attribute.

EXAMPLE

QUESTION: Find the average search time to find a record if you use a heap organization for the following scenario:

- r = 100,000 records stored on a disk with block size B = 2048 bytes.
- Records are fixed size of R = 500 bytes.
- Blocking Factor = 2048/500 = __4__ records per block (fill in the blank)
- # of blocks needed is __100,000/4__ = __25,000__ blocks
- Hash to a Record __1__ block accesses (What assumption have we made→ __NO COLLISIONS OCCURRED__) so it has a perfect time complexity.

if we have a collision, we have to go to the overflow table