CompSci 516
Data Intensive Computing Systems

Lecture 7
Indexing and Query Evaluation

Instructor: Sudeepa Roy

Announcement

- Homework 1
  - Due on Feb 9 (Tuesday), 11:59 pm
  - Check out clarifications and Q/A on Piazza
  - You are doing a great job!
  - Keep asking and answering questions!

What will we learn?

- Last lecture:
  - Storage and tree-based indexing
- Next:
  - Hash-based indexing
    - Static and dynamic (extendible hashing, linear hashing)

Reading Material

- [RG]
  - Hash-based index: Chapter 11
  - Query evaluation: Chapter 12
- [GUW]
  - Hash-based index: Chapter 14.3
  - Query evaluation: Chapter 15

Acknowledgement:
The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.

Hash-based Index

Recall from the previous lecture….

- For an index,
  - Search key = k
  - Data entry (stored in the index file) = k*
  - Data entry points to the data record with search key k
- Three alternatives for data entries **k***:
  1. The entire data record with key value **k**
  2. <**k**, rid>
  3. <**k**, list-of-rids>
- The above choice is orthogonal to the indexing technique used to locate data entries **k*** given k
  - Tree-based (Lecture 6)
  - Hash-based (this lecture)

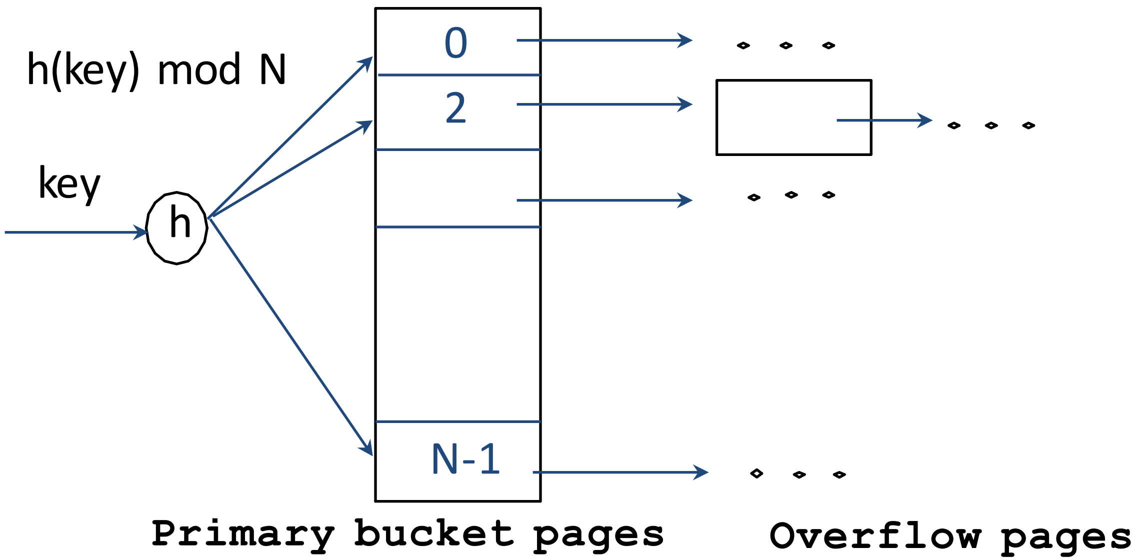
Introduction

- Hash-based indexes are best for equality selections
  - Cannot support range searches
  - But useful in implementing relational operators like join (later)

- Static and dynamic hashing techniques exist
  - trade-offs similar to ISAM vs. B+ trees

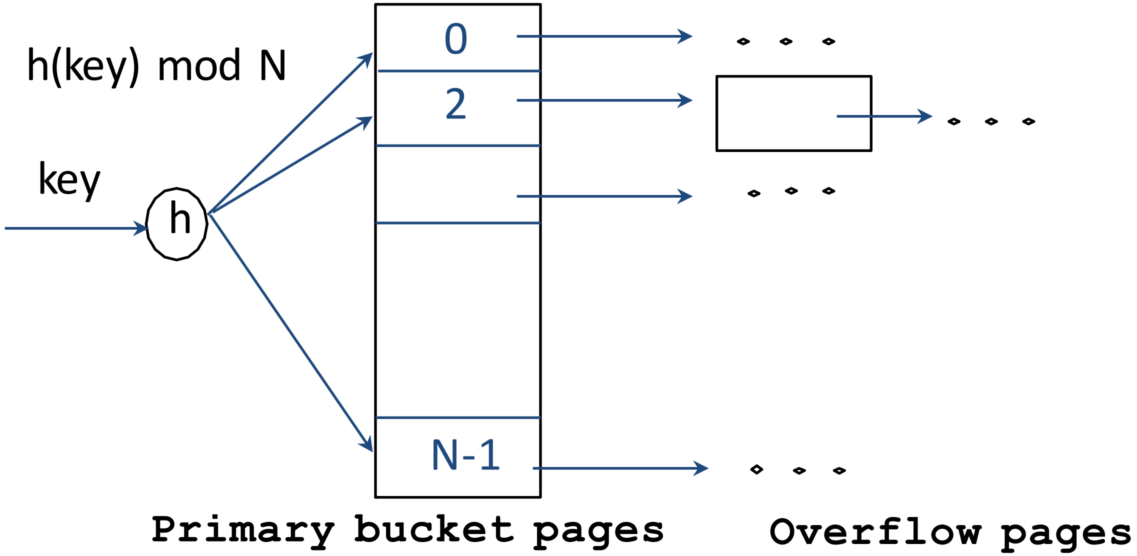
Static Hashing

- Pages containing data = a collection of buckets
  - each bucket has one primary page, also possibly overflow pages
  - buckets contain data entries k*



Static Hashing

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



Static Hashing

- Hash function works on search key field of record r
  - Must distribute values over range 0 ... N-1.
  - h(key) = (a * key + b) usually works well.
  - a and b are constants – chosen to tune h
- Advantage:
  - #buckets known – pages can be allocated sequentially
  - search needs 1 I/O (if no overflow page)
  - insert/delete needs 2 I/O (if no overflow page)
- Disadvantage:
  - Long overflow chains can develop and degrade performance
- Solutions:
  - keep some pages say 80% full initially
  - Rehash if overflow pages (can be expensive)
  - or use Dynamic Hashing

Dynamic Hashing Techniques

- Extendible Hashing
- Linear Hashing

Extendible Hashing

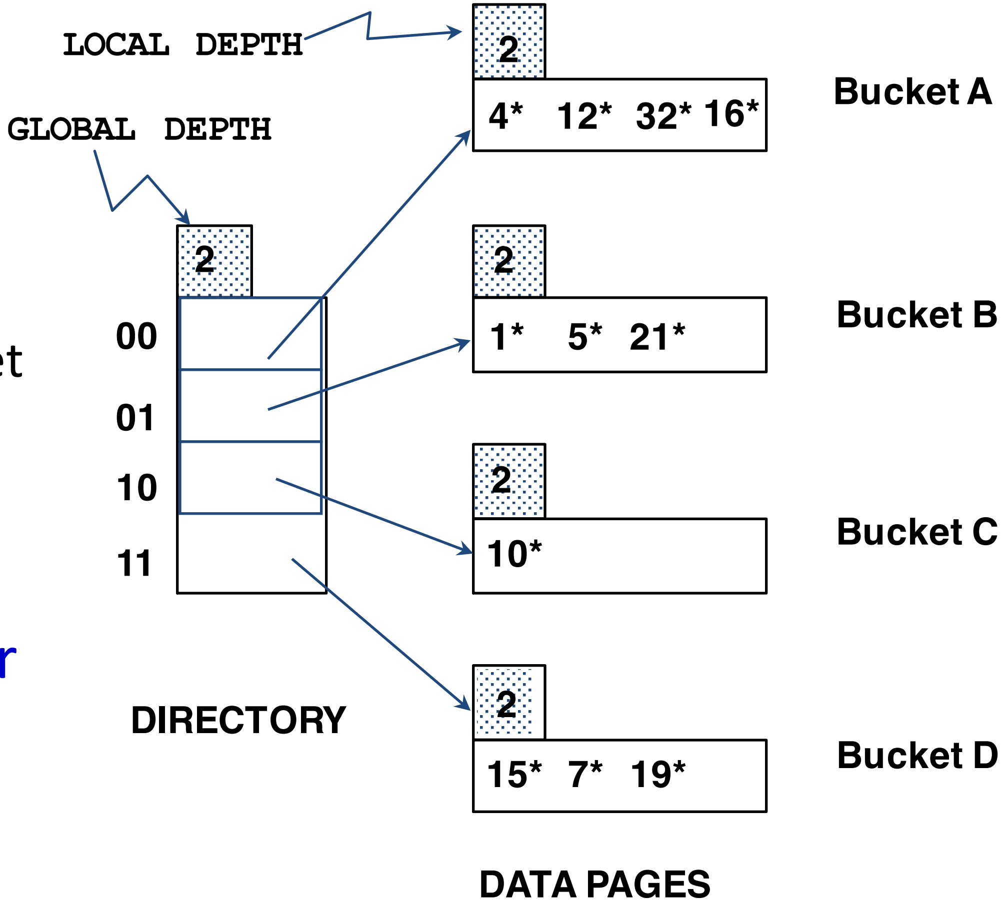
- Consider static hashing
- Bucket (primary page) becomes full

- Why not re-organize file by doubling # of buckets?
  - Reading and writing (double #pages) all pages is expensive

- Idea: Use directory of pointers to buckets
  - double # of buckets by doubling the directory, splitting just the bucket that overflowed
  - Directory much smaller than file, so doubling it is much cheaper
  - Only one page of data entries is split
  - No overflow page (new bucket, no new overflow page)
  - Trick lies in how hash function is adjusted

Example

- Directory is array of size 4
  - each element points to a bucket
  - #bits to represent = log 4 = 2 = global depth
- To find bucket for search key r
  - take last global depth # bits of **h**(r)
  - assume h(r) = r
  - If **h**(r) = 5 = binary 101
  - it is in bucket pointed to by 01.



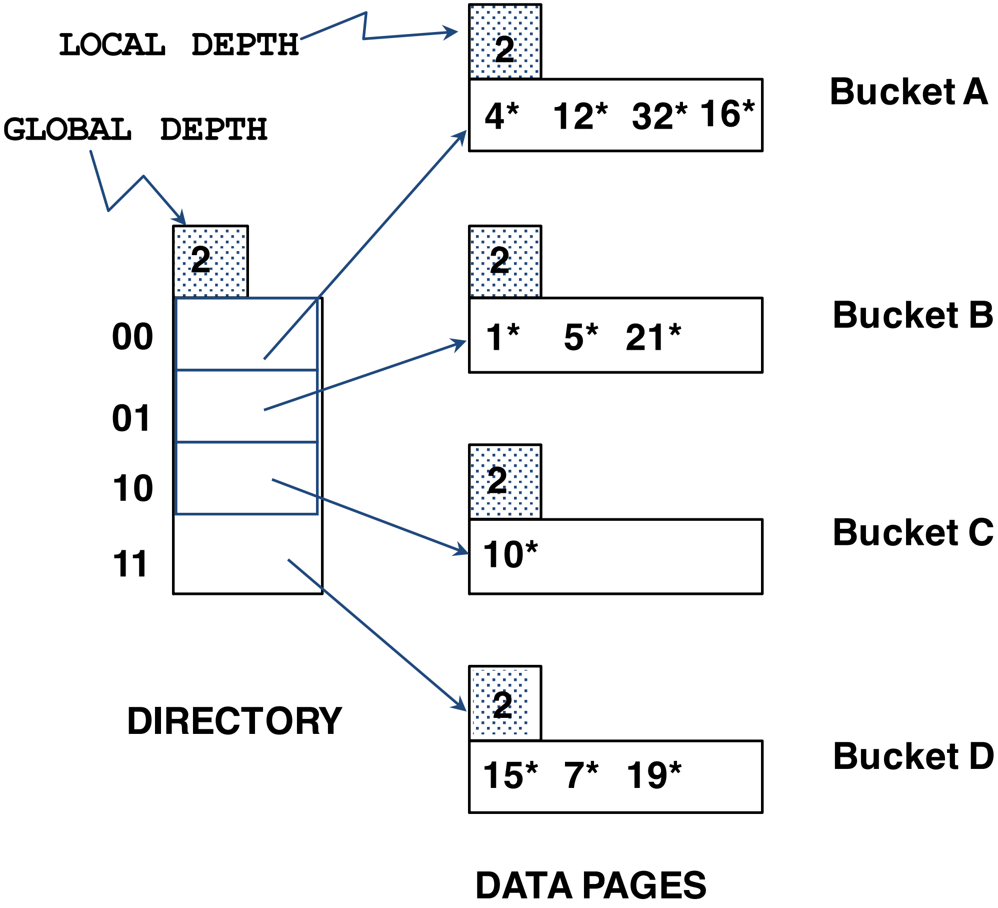
Example

Insert:

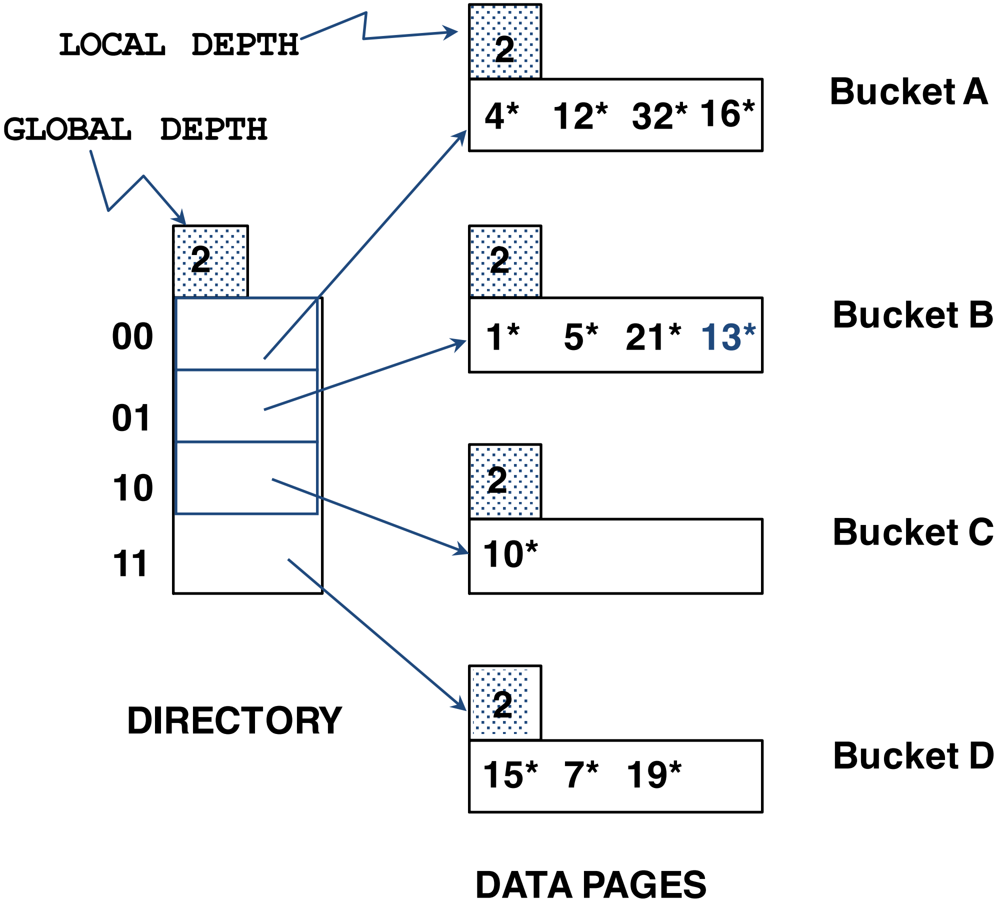
- If bucket is full, split it
- allocate new page
- re-distribute

Suppose inserting 13*

- binary = 1101
- bucket 01
- Has space, insert



Example



Insert:

- If bucket is full, split it
- allocate new page
- re-distribute

Suppose inserting 20*

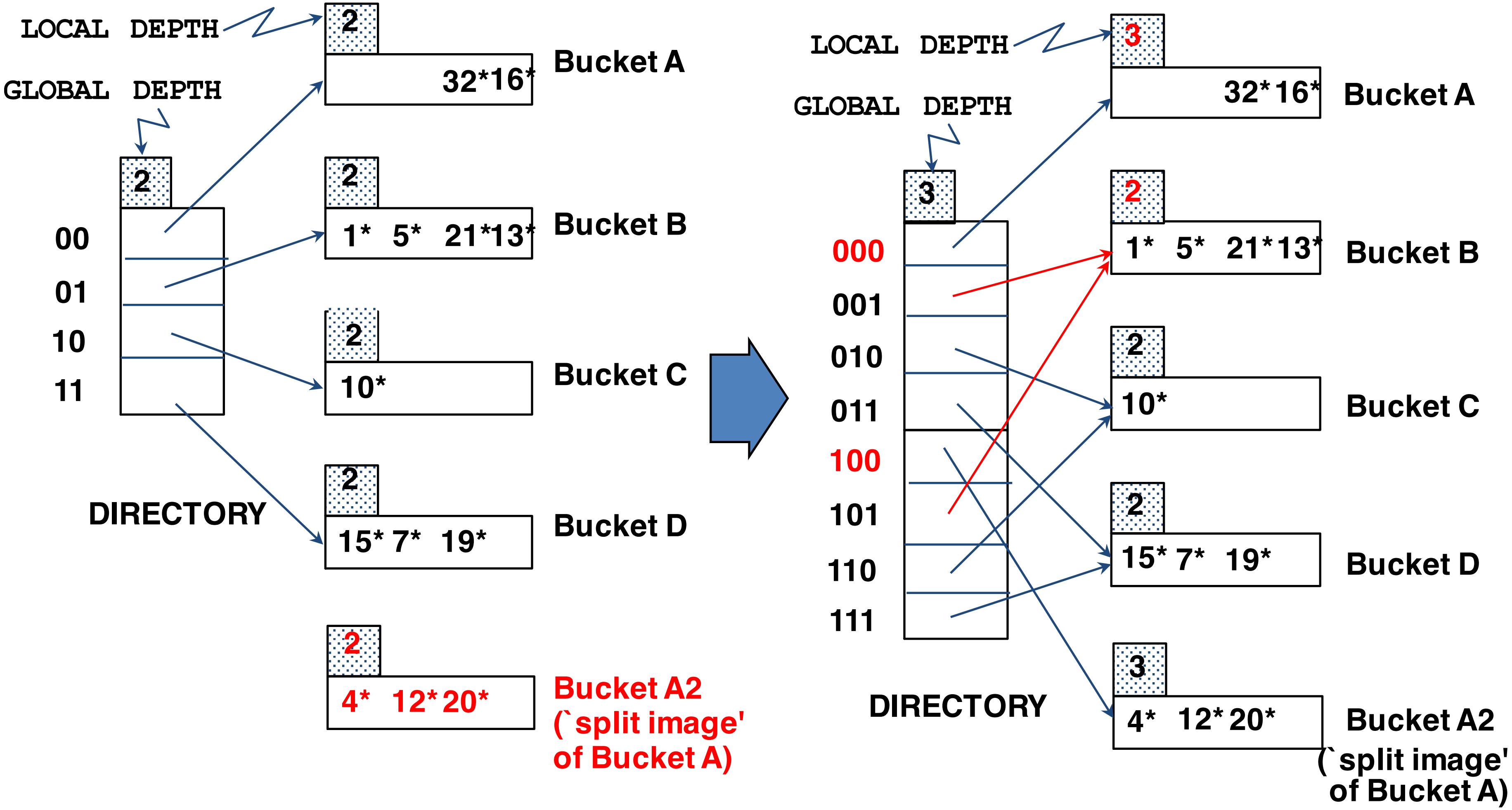
- binary = 10100
- bucket 00
- Already full
- To split, consider last three bits of 10100
- Last two bits the same 00 – the data entry will belong to one of these buckets
- Third bit to distinguish them

Example

Global depth: Max # of bits needed to tell which bucket an entry belongs to

Local depth: # of bits used to determine if an entry belongs to this bucket

- denotes whether a directory doubling is needed while splitting
- no directory doubling needed when 9* = 1001 is inserted

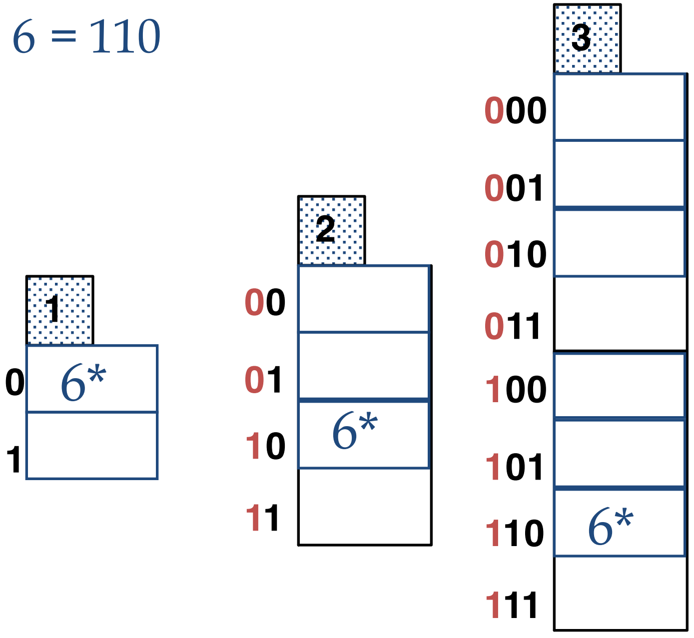


When does bucket split cause directory doubling?

- Before insert, local depth of bucket = global depth
- Insert causes local depth to become > global depth
- directory is doubled by copying it over and `fixing' pointer to split image page

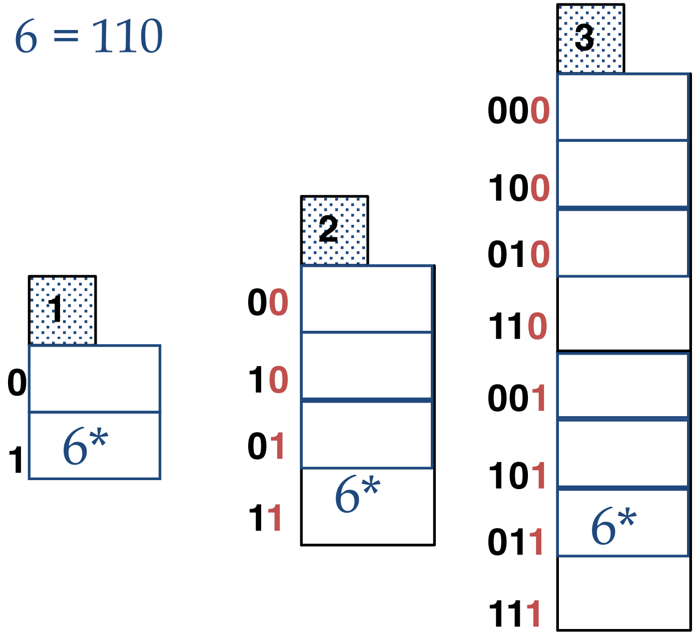
Directory Doubling

Why use least significant bits in directory?
Allows for doubling via copying!



Least Significant

vs.



Most Significant

Comments on Extendible Hashing

- If directory fits in memory, equality search answered with one disk access (to access the bucket); else two.
  - 100MB file, 100 bytes/rec, 4KB page size, contains 1,000,000 records (as data entries) and 25,000 directory elements; chances are high that directory will fit in memory.
  - Directory grows in spurts, and, if the distribution *of hash values* is skewed, directory can grow large.
  - Multiple entries with same hash value cause problems
- Delete:
  - If removal of data entry makes bucket empty, can be merged with `split image’
  - If each directory element points to same bucket as its split image, can halve directory.

Linear Hashing

- This is another dynamic hashing scheme
  - an alternative to Extendible Hashing
- LH handles the problem of long overflow chains
  - without using a directory
  - handles duplicates and collisions
  - very flexible w.r.t. timing of bucket splits

Linear Hashing: Basic Idea

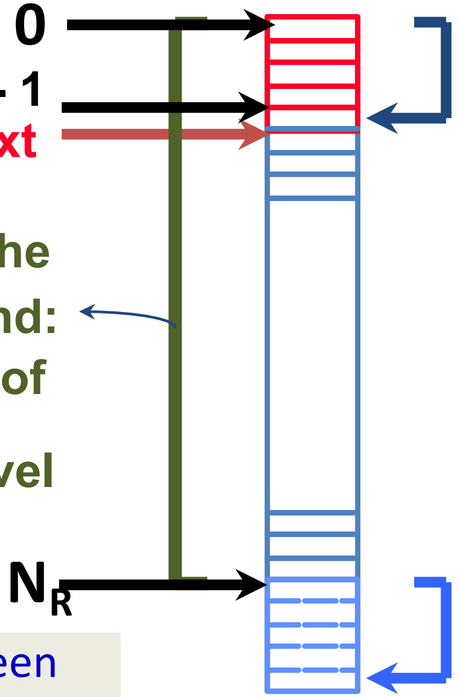
- Use a family of hash functions h_0, h_1, h_2, ...
  - $h_i(key) = h(key) \mod (2^i N)$
  - N = initial # buckets
  - h is some hash function (range is not 0 to N-1)
  - If $N = 2^{d_0}$, for some d_0, h_i consists of applying h and looking at the last d_i bits, where $d_i = d_0 + i$
    - Note: $h_i(key) = h(key) \mod (2^{d_0+i})$
  - h_{i+1} doubles the range of h_i
    - if h_i maps to M buckets, h_{i+1} maps to 2M buckets
    - similar to directory doubling

Linear Hashing: Rounds

- Directory avoided in LH by using overflow pages, and choosing bucket to split round-robin
- During round Level, only h_{Level} and $h_{Level+1}$ are in use
- The buckets from start to last are split sequentially
  - this doubles the no. of buckets
- Therefore, at any point in a round, we have
  - buckets that have been split
  - buckets that are yet to be split
  - buckets created by splits in this round

Overview of LH File

- In the middle of a round Level



Bucket to be split

Buckets that existed at the beginning of this round: this is the range of h_{Level}

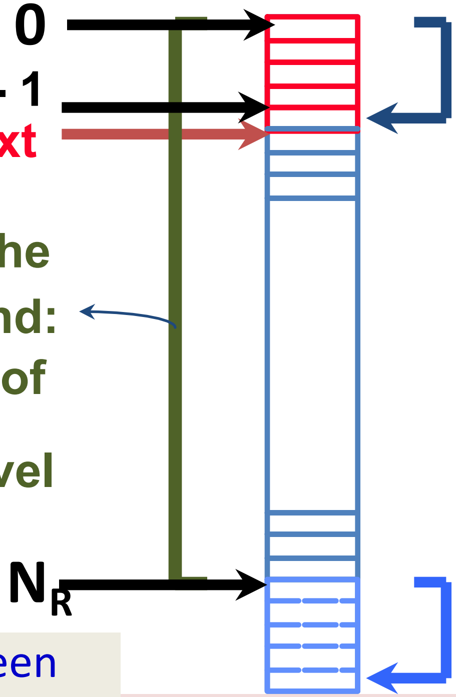
Buckets split in this round: If $h_{Level}(r)$ is in this range, must use $h_{Level+1}(r)$ to decide if entry is in `split image' bucket.

`split image' buckets: created (through splitting of other buckets) in this round

- Buckets 0 to Next-1 have been split
- Next to N_R yet to be split
- Round ends when all N_R initial (for round R) buckets are split

Linear Hashing: Search

- In the middle of a round Level



Bucket to be split

Buckets that existed at the beginning of this round: this is the range of h_{Level}

Buckets split in this round: If $h_{Level}(r)$ is in this range, must use $h_{Level+1}(r)$ to decide if entry is in `split image' bucket.

`split image' buckets: created (through splitting of other buckets) in this round

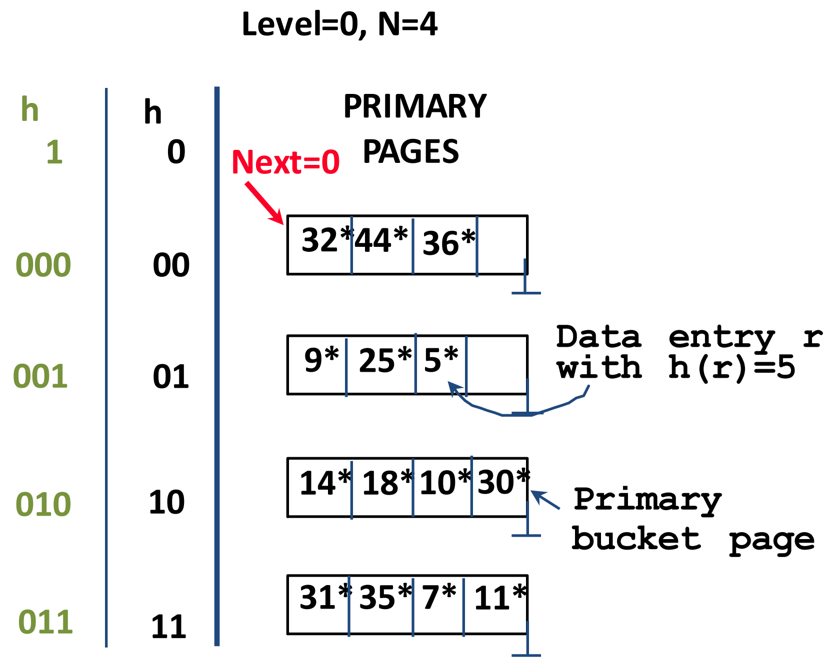
- Buckets 0 to Next-1 have been split
- Next to N_R yet to be split
- Round ends when all N_R initial (for round R) buckets are split

- Search: To find bucket for data entry r, find $h_{Level}(r)$:
- If $h_{Level}(r)$ in range `Next to N_R', r belongs here.
- Else, r could belong to bucket $h_{Level}(r)$ or $h_{Level}(r)+N_R$
- must apply $h_{Level+1}(r)$ to find out.

Linear Hashing: Insert

- Insert: Find bucket by applying h_{Level} / $h_{Level+1}$:
  - If bucket to insert into is full:
    - Add overflow page and insert data entry.
    - Split Next bucket and increment Next
- Note: We are going to assume that a split is `triggered' whenever an insert causes the creation of an overflow page, but in general, we could impose additional conditions for better space utilization ([RG], p.380)

Example of Linear Hashing



Level=0, N=4

Next=0

h_1	h_0	PRIMARY PAGES
000	00	32* 44* 36*
001	01	9* 25* 5*
010	10	14* 18* 10* 30*
011	11	31* 35* 7* 11*

Data entry r with h(r)=5

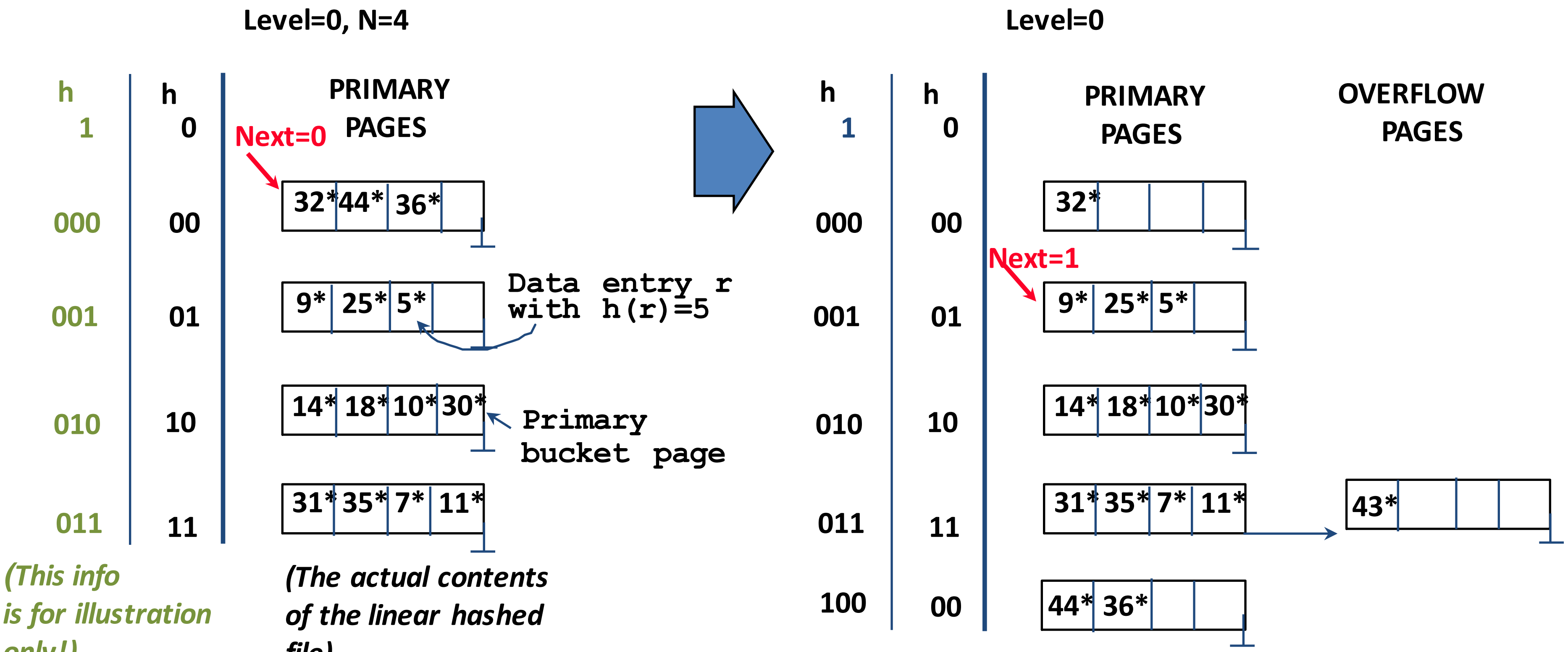
Primary bucket page

(This info is for illustration only!)

(The actual contents of the linear hashed file)

- Insert 43* = 101011
- $h_0(43) = 11$
- Full
- Insert in an overflow page
- Need a split at Next (=0)
- Entries in 00 is distributed to 000 and 100

Example of Linear Hashing



(This info is for illustration only!)

(The actual contents of the linear hashed file)

- Next is incremented after split
- Note the difference between overflow page (11) and split image (000 and 100)

Example of Linear Hashing

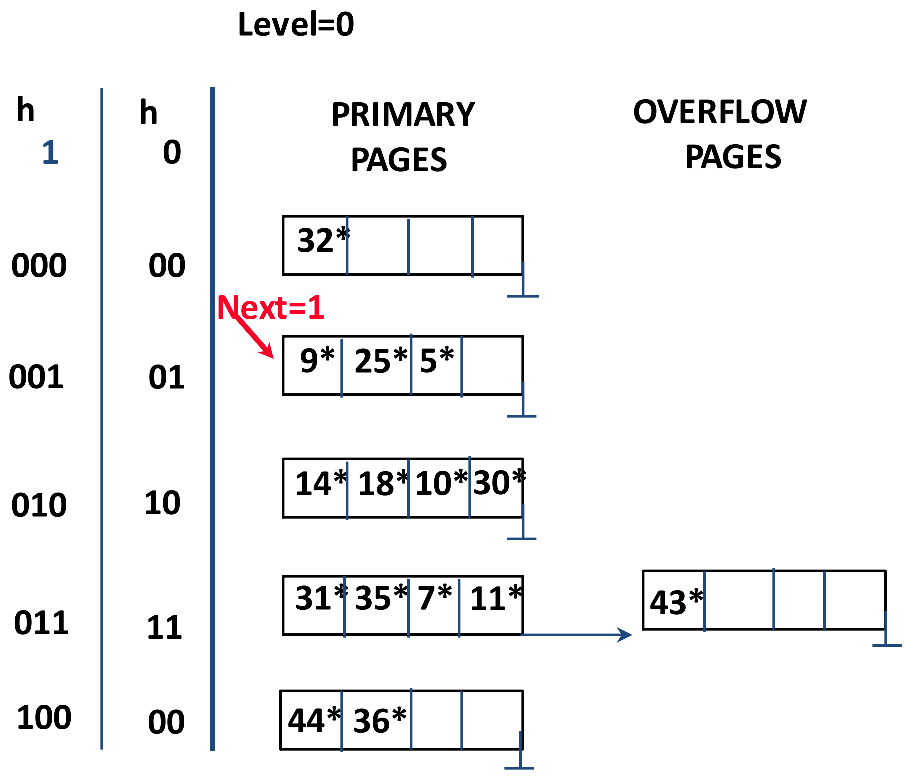
- Search for 18* = 10010
  - between Next (=1) and 4
  - this bucket has not been split

- Search for 32* = 100000
  or 44* = 101100
- Between 0 and Next-1
- Need h_1

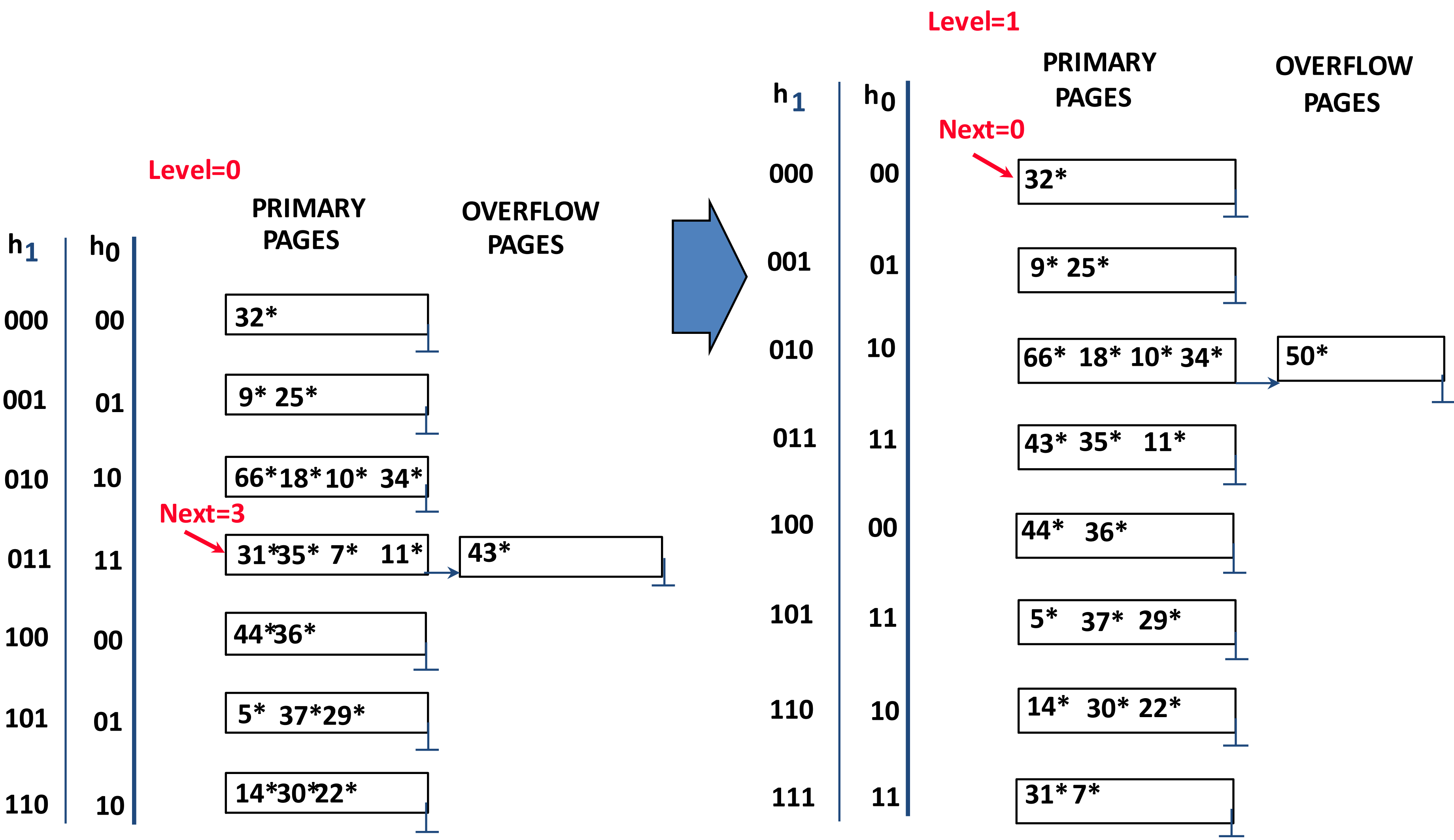
- Not all insertion triggers split
  - Insert 37* = 100101
  - Has space

- Splitting at Next?
  - No overflow bucket needed
  - Just copy at the image/original

- Next = N_{level}-1 and a split?
  - Start a new round
  - Increment Level
  - Next reset to 0



Example: End of a Round



Level=0

h_1	h_0	PRIMARY PAGES	OVERFLOW PAGES
000	00	32*	
001	01	9* 25*	
010	10	66* 18* 10* 34*	
011	11	Next=3 31* 35* 7* 11*	43*
100	00	44* 36*	
101	01	5* 37* 29*	
110	10	14* 30* 22*	

Level=1

h_1	h_0	PRIMARY PAGES	OVERFLOW PAGES
000	00	Next=0 32*	
001	01	9* 25*	
010	10	66* 18* 10* 34*	50*
011	11	43* 35* 11*	
100	00	44* 36*	
101	11	5* 37* 29*	
110	10	14* 30* 22*	
111	11	31* 7*	

LH Described as a Variant of EH

- The two schemes are actually quite similar:
- Begin with an EH index where directory has *N* elements.
  - Use overflow pages, split buckets round-robin.
  - First split is at bucket 0
    - Imagine directory being doubled at this point
  - But elements <1,*N*+1>, <2,*N*+2>, ... are the same. So, need only create directory element *N*, which differs from 0, now.
    - When bucket 1 splits, create directory element *N*+1, etc.
- So, directory can double gradually
- Also, primary bucket pages are created in order
- If they are *allocated* in sequence too (so that finding i'th is easy), we actually don't need a directory
- Voila, LH.

LH vs. EH

- Uniform distribution: LH has lower average cost
  - No directory level
- Skewed distribution
  - Many empty/nearly empty buckets in LH
  - EH may be better

Summary

- Hash-based indexes: best for equality searches, cannot support range searches.
- Static Hashing can lead to long overflow chains.
- Extendible Hashing avoids overflow pages by splitting a full bucket when a new data entry is to be added to it
  - Duplicates may still require overflow pages
  - Directory to keep track of buckets, doubles periodically
  - Can get large with skewed data; additional I/O if this does not fit in main memory

Summary

- Linear Hashing avoids directory by splitting buckets round-robin, and using overflow pages
  - Overflow pages not likely to be long
  - Duplicates handled easily
  - Space utilization could be lower than Extendible Hashing, since splits not concentrated on `dense' data areas
  - Can tune criterion for triggering splits to trade-off slightly longer chains for better space utilization.
- For hash-based indexes, a *skewed* data distribution is one in which the *hash values* of data entries are not uniformly distributed

Overview of Query Evaluation

Overview of Query Evaluation

- How queries are evaluated in a DBMS
  - How DBMS describes data (tables and indexes)
- Recall Relational Algebra = Logical Query Plan
- Now Algorithms will be attached to each operator = Physical Query Plan
- Plan: Tree of R.A. ops, with choice of alg for each op.
  - Each operator typically implemented using a `pull' interface
  - when an operator is `pulled' for the next output tuples, it `pulls' on its inputs and computes them

Overview of Query Evaluation

- Two main issues in query optimization:

1. For a given query, what plans are considered?
   - Algorithm to search plan space for cheapest (estimated) plan.
2. How is the cost of a plan estimated?

- Ideally: Want to find best plan
- Practically: Avoid worst plans!

Some Common Techniques

- Algorithms for evaluating relational operators use some simple ideas extensively:
- Indexing:
  - Can use WHERE conditions to retrieve small set of tuples (selections, joins)
- Iteration:
  - Examine all tuples in an input tuple
  - Sometimes, faster to scan all tuples even if there is an index
  - And sometimes, we can scan the data entries in an index instead of the table itself
  - Does not use the index structure (hash or tree structure – can iterate over leaves in a tree)
- Partitioning:
  - By using sorting or hashing, we can partition the input tuples and replace an expensive operation by similar operations on smaller inputs

Watch for these techniques as we discuss query evaluation!

System Catalog

- Stores information about the relations and indexes involved
- Also called Data Dictionary

- Catalogs typically contain at least:
  - Size of the buffer pool and page size
  - # tuples (NTuples) and # pages (NPages) for each relation
  - # distinct key values (NKeys) and NPages for each index.
  - Index height, low/high key values (Low/High) for each tree index

- More detailed information (e.g., histograms of the values in some field) are sometimes stored

- Catalogs updated periodically.
  - Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok.

Access Paths

- A way of retrieving tuples from a table
- Consists of
  - a file scan
  - or, an index + a matching condition
- The access method contributes significantly to the cost of the operator
  - Any relational operator accepts one or more table as input

Index “matching” a search condition

- A tree index *matches* (a conjunction of) terms that involve only attributes in a *prefix* of the search key.
  - E.g., Tree index on <*a, b, c*> matches the selection
  - *a*=*5 AND b*=*3,*
  - and *a*=*5 AND b*>*6,*
  - but not *b*=*3*

- A hash index *matches* (a conjunction of) terms that has a term *attribute* = *value* for every attribute in the search key of the index.
  - E.g., Hash index on <*a, b, c*> matches
  - *a*=*5 AND b*=*3 AND c*=*5;*
  - but it does not match *b*=*3,*
  - *or a*=*5 AND b*=*3,*
  - *or a*>*5 AND b*=*3 AND c*=*5*

A Note on Complex Selections

- If index (hash or tree) on
  - search key <bid, sid>
- Selection condition
  - rname = ‘Joe’ AND bid = 5 AND sid = 3
- <bid, sid> can be used to retrieve all tuples with bid = 5 and sid = 3
  - then apply rname = ‘Joe’ to each such tuple to eliminate more

A Note on Complex Selections

- Suppose two indexes
  - B+ tree index on day
  - index on search key <bid, sid>
- Selection condition
  - day<8/9/94 AND bid = 5 AND sid = 3
  - Two choices
  - Part of the index not matched – check for each retrieved tuple
- We only discuss case with no ORs

Access Paths: Selectivity

- Selectivity:
  - the number of pages retrieved for an access path
  - includes data pages + index pages
- If there is an index, many options:
  1. Scan the data file
  2. Use the index to retrieve tuples
  3. (possible sometimes) Just scan the index, rather than scanning the data file or using the index to probe

Most Selective Access Paths

- An index or file scan that we estimate will require the fewest page I/Os.
  - Terms that match this index reduce the number of tuples *retrieved*
  - other terms are used to discard some retrieved tuples, but do not affect number of tuples/pages fetched.

To be continued in the next lecture