CompSci 516
Data Intensive Computing Systems

Lecture 5 and 6
Storage and Indexing

Instructor: Sudeepa Roy
Announcements

• Homework 1
  – Due on September 16 (Friday), 11:55 pm

• Homework 2: AWS account set up instructions
  • carefully use the credit
  • remember to turn off instances (to avoid additional charges)

• Conflict with CS graduate students retreat for September 30 (Friday) class
  – Midterm moved to October 12 (Wednesday)
  – A piazza poll will be posted today for a make-up class for those who are participating – for others, there will be a class on Sept 30
  – Please fill out by tomorrow
Where are we now?

We learnt
✓ Relational Model and Query Languages
  ✓ SQL, RA, RC
  ✓ Postgres (DBMS)
    ▪ HW1
✓ Map-reduce and spark
  ▪ HW2

Next
• DBMS Internals
  – Storage
  – Indexing
  – Query Evaluation
  – Operator Algorithms
  – External sort
  – Query Optimization
Reading Material

• [RG]
  – Storage: Chapters 8.1, 8.2, 8.4, 9.4-9.7
  – Index: 8.3, 8.5
  – Tree-based index: Chapter 10.1-10.7
  – Hash-based index: Chapter 11

Additional reading
• [GUW]
  – Chapters 8.3, 14.1-14.4

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

• How does a DBMS organize files?
  – Record format, Page format

• What is an index?

• What are different types of indexes?
  – Tree-based indexing:
    • B+ tree
    • insert, delete
  – Hash-based indexing
    • Static and dynamic (extendible hashing, linear hashing)

• How do we use index to optimize performance?
Storage
A typical DBMS has a layered architecture.

The figure does not show the concurrency control and recovery components — to be done in “transactions”.

This is one of several possible architectures — each system has its own variations.

These layers must consider concurrency control and recovery.
Data on External Storage

• Data must persist on disk across program executions in a DBMS
  – Data is huge
  – Must persist across executions
  – But has to be fetched into main memory when DBMS processes the data

• The unit of information for reading data from disk, or writing data to disk, is a page

• Disks: Can retrieve random page at fixed cost
  – But reading several consecutive pages is much cheaper than reading them in random order
Disk Space Management

- Lowest layer of DBMS software manages space on disk

- Higher levels call upon this layer to:
  - allocate/de-allocate a page
  - read/write a page

- Size of a page = size of a disk block
  = data unit

- Request for a sequence of pages often satisfied by allocating contiguous blocks on disk

- Space on disk managed by Disk-space Manager
  - Higher levels don’t need to know how this is done, or how free space is managed
Suppose

- 1 million pages in db, but only space for 1000 in memory
- A query needs to scan the entire file
- DBMS has to
  - bring pages into main memory
  - decide which existing pages to replace to make room for a new page
  - called Replacement Policy

- Managed by the Buffer manager
  - Files and access methods ask the buffer manager to access a page mentioning the “record id” (soon)
  - Buffer manager loads the page if not already there
Buffer Management

Buffer pool = main memory is partitioned into frames either contains a page from disk or is a free frame

Page Requests from Higher Levels

BUFFER POOL

disk page

free frame

MAIN MEMORY

DISK

DB

choice of frame dictated by replacement policy

• Data must be in RAM for DBMS to operate on it
• Table of <frame#, pageid> pairs is maintained
When a Page is Requested ...

For every frame, store

- **a dirty bit:**
  - whether the page has been modified since it has been brought to memory
  - initially 0 or off

- **a pin-count:**
  - the number of times a page has been requested but not released (and no. of current users)
  - initially 0
  - when a page is requested, the count in incremented
  - when the requestor releases the page, count is decremented
  - buffer manager only reads a page into a frame when its pin-count is 0
  - if no page with pin-count 0, buffer manager has to wait (or a transaction is aborted -- later)
When a Page is Requested ...

- Check if the page is already in the buffer pool
- If yes, increment the pin-count of that frame
- If no,
  - Choose a frame for replacement using the replacement policy
  - If the chosen frame is dirty (has been modified), write it to disk
  - Read requested page into chosen frame
- Pin (increase pin-count of) the page and return its address to the requestor

- If requests can be predicted (e.g., sequential scans), pages can be pre-fetched several pages at a time
- Concurrency Control & recovery may entail additional I/O when a frame is chosen for replacement
  - e.g. Write-Ahead Log protocol : when we do Transactions
Buffer Replacement Policy

• Frame is chosen for replacement by a replacement policy

• Least-recently-used (LRU)
  – add frames with pin-count 0 to the end of a queue
  – choose from head

• Clock
  – an efficient implementation of LRU
  – Assign 1 to N (=#frames) to frames
  – choose next frame with pin-count 0

• First In First Out (FIFO)
• Most-Recently-Used (MRU) etc.
Buffer Replacement Policy

- Policy can have big impact on # of I/O’s
- Depends on the access pattern
- **Sequential flooding:** Nasty situation caused by LRU + repeated sequential scans
  - What happens with 10 frames and 9 pages?
  - What happens with 10 frames and 11 pages?
  - # buffer frames < # pages in file means each page request in each scan causes an I/O
  - MRU much better in this situation (but not in all situations, of course)
DBMS vs. OS File System

- Operating Systems do disk space and buffer management too:
- Why not let OS manage these tasks?

- DBMS can predict the page reference patterns much more accurately
  - can optimize
  - adjust replacement policy
  - pre-fetch pages – already in buffer + contiguous allocation
  - pin a page in buffer pool, force a page to disk (important for implementing Transactions concurrency control & recovery)

- Differences in OS support: portability issues

- Some limitations, e.g., files can’t span disks
Files of Records

- Page or block is OK when doing I/O, but higher levels of DBMS operate on records, and files of records
- **FILE**: A collection of pages, each containing a collection of records
- **Must support:**
  - insert/delete/modify record
  - read a particular record (specified using record id)
  - scan all records (possibly with some conditions on the records to be retrieved)
File Organization

• **File organization:** Method of arranging a file of records on external storage
  – One file can have multiple pages
  – Record id (rid) is sufficient to physically locate the page containing the record on disk
  – Indexes are data structures that allow us to find the record ids of records with given values in index search key fields

• **NOTE:** Several uses of “keys” in a database
  – Primary/foreign/candidate/super keys
  – Index search keys
Alternative File Organizations

Many alternatives exist, each ideal for some situations, and not so good in others:

• **Heap (random order) files**: Suitable when typical access is a file scan retrieving all records

• **Sorted Files**: Best if records must be retrieved in some order, or only a “range” of records is needed.

• **Indexes**: Data structures to organize records via trees or hashing
  – Like sorted files, they speed up searches for a subset of records, based on values in certain (“search key”) fields
  – Updates are much faster than in sorted files
Unordered (Heap) Files

• Simplest file structure contains records in no particular order
• As file grows and shrinks, disk pages are allocated and de-allocated
• To support record level operations, we must:
  – keep track of the pages in a file
  – keep track of free space on pages
  – keep track of the records on a page
• There are many alternatives for keeping track of this
Heap File Implemented as a List

- The header page id and Heap file name must be stored someplace.
- Each page contains 2 `pointers` plus data.
- Problem:
  - to insert a new record, we may need to scan several pages on the free list to find one with sufficient space.
The entry for a page can include the number of free bytes on the page.

The directory is a collection of pages
  – linked list implementation of directory is just one alternative
  – Much smaller than linked list of all heap file pages!
How do we arrange a collection of records on a page?

• Each page contains several slots
  – one for each record

• Record is identified by <page-id, slot-number>

• Fixed-Length Records
• Variable-Length Records

• For both, there are options for
  – Record formats (how to organize the fields within a record)
  – Page formats (how to organize the records within a page)
**Page Formats: Fixed Length Records**

- **Record id = <page id, slot #>**
- **Packed:** moving records for free space management changes rid; may not be acceptable
- **Unpacked:** use a bitmap – scan the bit array to find an empty slot
- **Each page also may contain additional info like the id of the next page (not shown)**
Page Formats: Variable Length Records

• Need to find a page with the right amount of space
  – Too small – cannot insert
  – Too large – waste of space

• if a record is deleted, need to move the records so that all free space is contiguous
  – need ability to move records within a page

• Can maintain a directory of slots (next slide)
  – <record-offset, record-length>
  – deletion = set record-offset to -1

• Record-id rid = <page, slot-in-directory> remains unchanged
Page Formats: Variable Length Records

- Can move records on page without changing rid
  - so, attractive for fixed-length records too
- Store (record-offset, record-length) in each slot
- rid-s unaffected by rearranging records in a page
Record Formats: Fixed Length

- Each field has a fixed length
  - for all records
  - the number of fields is also fixed
  - fields can be stored consecutively
- Information about field types same for all records in a file
  - stored in system catalogs
- Finding i-th field does not require scan of record
  - given the address of the record, address of a field can be obtained easily

\[ \text{Address} = B + L1 + L2 \]
Record Formats: Variable Length

- Cannot use fixed-length slots for records
- Two alternative formats (# fields is fixed):
  - Second offers direct access to i-th field, efficient storage of nulls (special don’t know value); small directory overhead
  - Modification may be costly (may grow the field and not fit in the page)
Indexes
Announcements

• Homework 1
  – Due TODAY: September 16 (Friday), 11:55 pm

• Conflict with CS graduate students retreat for September 30 (Friday) class
  – Midterm moved to October 12 (Wednesday)
  – Regular class on September 30
  – Make up lecture on September 29 (Thursday), 4:40-5:55 pm, LSRC D309: only for students going to CS grad retreat (room accommodates ~10 people)
Indexes

• An index on a file speeds up selections on the search key fields for the index
  – Any subset of the fields of a relation can be the search key for an index on the relation.
  – “Search key” is not the same as “key”
    key = minimal set of fields that uniquely identify a tuple

• An index contains a collection of data entries, and supports efficient retrieval of all data entries \( k^* \) with a given key value \( k \)
Alternatives for Data Entry $k^*$ in Index $k$

- In a data entry $k^*$ we can store:
  1. (Alternative 1) The actual data record with key value $k$, or
  2. (Alternative 2) $<k, \text{rid}>$
     - $\text{rid} =$ record of data record with search key value $k$, or
  3. (Alternative 3) $<k, \text{rid-list}>$
     - list of record ids of data records with search key $k$

- Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value $k$
Alternatives for Data Entries: Alternative 1

- In a data entry $k^*$ we can store:
  1. The actual data record with key value $k$
  2. $<k, \text{rid}>$
     - rid = record of data record with search key value $k$
  3. $<k, \text{rid-list}>$
     - list of record ids of data records with search key $k$\n
Advantages/Disadvantages?

- Index structure is a file organization for data records
  - instead of a Heap file or sorted file

- How many different indexes can use Alternative 1?

- At most one index can use Alternative 1
  - Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency

- If data records are very large, #pages with data entries is high
  - Implies size of auxiliary information in the index is also large
Alternatives for Data Entries: Alternative 2, 3

- In a data entry $k^*$ we can store:
  1. The actual data record with key value $k$
  2. $<k, \text{rid}>$
     - $\text{rid} =$ record of data record with search key value $k$
  3. $<k, \text{rid-list}>$
     - list of record ids of data records with search key $k$

- **Data entries typically much smaller than data records**
  - So, better than Alternative 1 with large data records
  - Especially if search keys are small.

- **Alternative 3 more compact than Alternative 2**
  - but leads to variable-size data entries even if search keys have fixed length.
Index Classification

- Primary vs. secondary
- Clustered vs. unclustered
- Tree-based vs. Hash-based
Primary vs. Secondary Index

• If search key contains primary key, then called primary index, otherwise secondary
  – Unique index: Search key contains a candidate key

• Duplicate data entries:
  – if they have the same value of search key field $k$
  – Primary/unique index never has a duplicate
  – Other secondary index can have duplicates
Clustered vs. Unclustered Index

• If order of data records in a file is the same as, or `close to’, order of data entries in an index, then clustered, otherwise unclustered
  – Alternative 1 implies clustered – 2, 3 are typically unclustered
    • unless sorted according to the search key
  – In practice, clustered also implies Alternative 1 (since sorted files are rare)
  – A file can be clustered on at most one search key
  – Cost of retrieving data records (range queries) through index varies greatly based on whether index is clustered or not
Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file.

To build clustered index, first sort the Heap file:
- with some free space on each page for future inserts
- Overflow pages may be needed for inserts
- Thus, data records are ‘close to’, but not identical to, sorted

Clustered vs. Unclustered Index

Clustered

Index entries
direct search for
data entries

Data entries

Data Records

(Index File)

(Data file)

Unclustered
Methods for indexing

• Tree-based
• Hash-based

• (in detail later)
System Catalogs

• For each index:
  – structure (e.g., B+ tree) and search key fields
• For each relation:
  – name, file name, file structure (e.g., Heap file)
  – attribute name and type, for each attribute
  – index name, for each index
  – integrity constraints
• For each view:
  – view name and definition
• Plus statistics, authorization, buffer pool size, etc.
• (described in [RG] 12.1)

Catalogs are themselves stored as relations!
Remember Terminology

• **Index search key (key):** k
  – Used to search a record

• **Data entry :** k*
  – Pointed to by k
  – Contains record id(s) or record itself

• **Records or data**
  – Actual tuples
  – Pointed to by record ids
Tree-based Index and $B^+$-Tree
Range Searches

• "Find all students with gpa > 3.0"
  – If data is in sorted file, do binary search to find first such student, then scan to find others.
  – Cost of binary search can be quite high.
Index file format

- Simple idea: Create an “index file”
  - <first-key-on-page, pointer-to-page>, sorted on keys

Can do binary search on (smaller) index file but may still be expensive: apply this idea repeatedly
Indexed Sequential Access Method (ISAM)

- Leaf-pages contain data entry – also some overflow pages
- DBMS organizes layout of the index – a static structure
- If a number of inserts to the same leaf, a long overflow chain can be created
  – affects the performance

Leaf pages contain data entries.
B+ Tree

- Most Widely Used Index
  - a dynamic structure
- Insert/delete at $\log_F N$ cost = height of the tree
  - $F = \text{fanout}$, $N = \text{no. of leaf pages}$
  - tree is maintained height-balanced
- Minimum 50% occupancy
  - Each node contains $d \leq m \leq 2d$ entries
  - Root contains $1 \leq m \leq 2d$ entries
  - The parameter $d$ is called the order of the tree
- Supports equality and range-searches efficiently

The index-file

Index Entries
(Direct search)

Data Entries
(“Sequence set”)

Duke CS, Fall 2016
CompSci 516: Data Intensive Computing Systems
B+ Tree Indexes

Leaf pages contain data entries, and are chained (prev & next)
- Non-leaf pages have index entries; only used to direct searches:

```
index entry
```

```
P0  K1  P1  K2  P2  .  .  .  K_m  P_m
```

Leaf Pages (Sorted by search key)

Non-leaf Pages
Example B+ Tree

• Search begins at root, and key comparisons direct it to a leaf

• Search for 5*, 15*, all data entries >= 24* ...

Based on the search for 15*, we know it is not in the tree!
Example B+ Tree

- **Find**
  - 28*
  - 29*
  - All > 15* and < 30*

Note how data entries in leaf level are sorted
B+ Trees in Practice

• Typical order: \( d = 100 \). Typical fill-factor: 67%.
  – average fanout \( F = 133 \)

• 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
Inserting a Data Entry into a B+ Tree

• Find correct leaf L
• Put data entry onto L
  – If L has enough space, done
  – Else, must split L
    • into L and a new node L2
    • Redistribute entries evenly, copy up middle key.
    • Insert index entry pointing to L2 into parent of L.

• This can happen recursively
  – To split index node, redistribute entries evenly, but push up middle key
    • Contrast with leaf splits

• Splits “grow” tree; root split increases height.
  – Tree growth: gets wider or one level taller at top.
Inserting 8* into Example B+ Tree

STEP-1

• Copy-up: 5 appears in leaf and the level above
• Observe how minimum occupancy is guaranteed

Entry to be inserted in parent node. (Note that 5 is copied up and continues to appear in the leaf.)
Inserting 8* into Example B+ Tree

• Note difference between copy-up and push-up
• What is the reason for this difference?
• All data entries must appear as leaves
  – (for easy range search)
• no such requirement for indexes
  – (so avoid redundancy)

Need to split parent

Entry to be inserted in parent node. (Note that 17 is pushed up and only appears once in the index. Contrast this with a leaf split.)
Notice that root was split, leading to increase in height.

In this example, we can avoid split by re-distributing entries (insert 8 to the 2nd leaf node from left and copy it up instead of 13)
  • however, this is usually not done in practice – since need to access 1-2 extra pages always (for two siblings), and average occupancy may remain unaffected as the file grows
Deleting a Data Entry from a B+ Tree

Each non-root node contains \( d \leq m \leq 2d \) entries

• Start at root, find leaf \( L \) where entry belongs

• Remove the entry
  – If \( L \) is at least half-full, done!
  – If \( L \) has only \( d-1 \) entries,
    • Try to re-distribute, borrowing from sibling (adjacent node with same parent as \( L \))
    • If re-distribution fails, merge \( L \) and sibling

• If merge occurred, must delete entry (pointing to \( L \) or sibling) from parent of \( L \)

• Merge could propagate to root, decreasing height

See this slide later,
First, see examples on the next few slides
Example Tree: Delete 19*

- We had inserted 8*
- Now delete 19*
- Easy
Example Tree: Delete 19*

Root

After deleting 19*
Example Tree: Delete 20*

Before deleting 20*
Example Tree: Delete 20*

- < 2 entries in leaf-node
- Redistribute

After deleting 20* - step 1
Example Tree: Delete 20*

- Notice how middle key is copied up
Example Tree: ... And Then Delete 24*

Before deleting 24*

Root

Before deleting 24*

Duke CS, Fall 2016
CompSci 516: Data Intensive Computing Systems
Once again, imbalance at leaf
- Can we borrow from sibling(s)?
- No – d-1 and d entries (d = 2)
- Need to merge
Example Tree: ... And Then Delete 24*

- Imbalance at parent
- Merge again
- But need to “pull down” root index entry

After deleting 24*
- Step 2

Observe ‘toss’ of old index entry 27

because, three index 5, 13, 30 but five pointers to leaves
Final Example Tree
Example of Non-leaf Re-distribution

- An intermediate tree is shown
- In contrast to previous example, can re-distribute entry from left child of root to right child
After Re-distribution

• Intuitively, entries are re-distributed by `pushing through’ the splitting entry in the parent node.
  – It suffices to re-distribute index entry with key 20; we’ve re-distributed 17 as well for illustration.
Duplicates

• **First Option:**
  – The basic search algorithm assumes that all entries with the same key value resides on the same leaf page
  – If they do not fit, use overflow pages (like ISAM)

• **Second Option:**
  – Several leaf pages can contain entries with a given key value
  – Search for the left most entry with a key value, and follow the leaf-sequence pointers
  – Need modification in the search algorithm

• if $k^* = <k, \text{rid}>$, several entries have to be searched
  – Or includerid in $k$ – becomes unique index, no duplicate
  – If $k^* = <k, \text{rid-list}>$, some solution, but if the list is long, again a single entry can span multiple pages
A Note on `Order`

• **Order (d)**
  - denotes minimum occupancy

• **replaced by physical space criterion in practice (`at least half-full`)**
  - Index pages can typically hold many more entries than leaf pages
  - Variable sized records and search keys mean different nodes will contain different numbers of entries.
  - Even with fixed length fields, multiple records with the same search key value (duplicates) can lead to variable-sized data entries (if we use Alternative (3))
Summary

• Tree-structured indexes are ideal for range-searches, also good for equality searches

• ISAM is a static structure
  – Only leaf pages modified; overflow pages needed
  – Overflow chains can degrade performance unless size of data set and data distribution stay constant

• B+ tree is a dynamic structure
  – Inserts/deletes leave tree height-balanced; \( \log_F N \) cost
  – High fanout \((F)\) means depth rarely more than 3 or 4
  – Almost always better than maintaining a sorted file
  – Most widely used index in database management systems because of its versatility.
  – One of the most optimized components of a DBMS
Hash-based Index
Hash-Based Indexes

• Records are grouped into buckets
  – Bucket = primary page plus zero or more overflow pages

• Hashing function $h$:
  – $h(r)$ = bucket in which (data entry for) record $r$ belongs
  – $h$ looks at the search key fields of $r$
  – No need for “index entries” in this scheme
Example: Hash-based index

Index organized file hashed on AGE, with Auxiliary index on SAL

Employee File hashed on AGE

Alternative 1

Alternative 2

File of <SAL, rid> pairs hashed on SAL
Introduction

• Hash-based indexes are best for equality selections
  – Find all records with name = “Joe”
  – 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
  – \( h(key) = (a \times key + b) \) usually works well
    • bucket = \( h(key) \mod N \)
  – 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) (why 2?)

• Disadvantage:
  – Long overflow chains can develop if file grows and degrade performance
  – Or waste of space if file shrinks

• Solutions:
  – keep some pages say 80% full initially
  – Periodically 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 = \( \log_2 4 = 2 \) = 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.
- also denotes whether a directory doubling is needed while splitting
- no directory doubling needed when 9* = 1001 is inserted (LD < GD)

Example
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
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 $10^6$ 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 $h_0$, $h_1$, $h_2$, ...
  - $h_i(key) = h(key) \mod(2^iN)$
  - $N = \text{initial \# buckets}$
  - $h$ 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 $h_{\text{Level}}$ and $h_{\text{Level}+1}$ 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

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

Next to $N_R$ yet to be split

Round ends when all $N_R$ initial (for round R) buckets are split

Buckets split in this round:
If $h_{\text{Level}}(r)$ is in this range, must use $h_{\text{Level}+1}(r)$ 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

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

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

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

**Buckets that existed at the beginning of this round:**
- this is the range of $h_{\text{Level}}$

- **Bucket to be split**
- **Next - 1**
- **Next**
- **$N_R$**

- **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: Insert

- **Insert**: Find bucket by applying $h_{\text{Level}} / h_{\text{Level}+1}$:
  - 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

<table>
<thead>
<tr>
<th>h</th>
<th>h</th>
<th>PRIMARY PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>32<em>44</em>36*</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>9<em>25</em>5*</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>14<em>18</em>10<em>30</em></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>31<em>35</em>7<em>11</em></td>
</tr>
</tbody>
</table>

**Data entry r with h(r)=5**

- 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

(This info is for illustration only!)

(The actual contents of the linear hashed file)
Example of Linear Hashing

Level=0, N=4

<table>
<thead>
<tr>
<th>h</th>
<th>h</th>
<th>PRIMARY PAGES</th>
<th>h</th>
<th>h</th>
<th>PRIMARY PAGES</th>
</tr>
</thead>
</table>
| 01  | 1 | 00
|     |   | 32*44*36*     | 00  | 0 | 32*           |
| 00  | 0 | 00
|     |   | 9*25*5*       | 01  | 0 | 9*25*5*       |
| 01  | 0 | 10             | 01  | 0 | 14*18*10*30* |
| 00  | 1 | 11             | 01  | 1 | 31*35*7*11*  |
|     |   |                | 00  | 1 | 44*36*        |
| 01  | 1 |                | 10  | 0 | 14*18*10*30* |
|     |   |                | 11  | 0 | 43*           |

(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

Level=1

Next=0

Next=3

32*

9* 25*

66*18*10* 34*

31*35* 7* 11*

43*

44*36*

5* 37*29*

14*30*22*

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 \(<1, N+1>, <2, N+2>, \ldots\) 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 \emph{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
Selection of Indexes
Different File Organizations

Search key = <age, sal>

- **Heap files**
  - random order; insert at end-of-file
- **Sorted files**
  - sorted on <age, sal>
- **Clustered B+ tree file**
  - search key <age, sal>
- **Heap file with unclustered B⁺-tree index**
  - on search key <age, sal>
- **Heap file with unclustered hash index**
  - on search key <age, sal>
Possible Operations

- **Scan**
  - Fetch all records from disk to buffer pool

- **Equality search**
  - Find all employees with age = 23 and sal = 50
  - Fetch page from disk, then locate qualifying record in page

- **Range selection**
  - Find all employees with age > 35

- **Insert a record**
  - Identify the page, fetch that page from disk, insert record, write back to disk (possibly other pages as well)

- **Delete a record**
  - Similar to insert
Understanding the Workload

• A workload is a mix of queries and updates

• For each query in the workload:
  – Which relations does it access?
  – Which attributes are retrieved?
  – Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?

• For each update in the workload:
  – Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
  – The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected
Choice of Indexes

• What indexes should we create?
  – Which relations should have indexes? What field(s) should be the search key? Should we build several indexes?

• For each index, what kind of an index should it be?
  – Clustered? Hash/tree?
More on Choice of Indexes

• One approach:
  – Consider the most important queries
  – Consider the best plan using the current indexes
  – See if a better plan is possible with an additional index.
  – If so, create it.
  – Obviously, this implies that we must understand how a DBMS evaluates queries and creates query evaluation plans
  – We will learn query execution and optimization later - For now, we discuss simple 1-table queries.

• Before creating an index, must also consider the impact on updates in the workload!
• Trade-off: Indexes can make queries go faster, updates slower. Require disk space, too.
Index Selection Guidelines – 1/3

• Attributes in WHERE clause are candidates for index keys.
  – Exact match condition suggests hash index
  – Range query suggests tree index
  – Clustering is especially useful for range queries
    • can also help on equality queries if there are many duplicates.
Index Selection Guidelines – 2/3

- Multi-attribute search keys should be considered when a \texttt{WHERE} clause contains several conditions.
  - Order of attributes is important for range queries.
  - Such indexes can sometimes enable \texttt{index-only} strategies for important queries.
  - For index-only strategies, clustering is not important.
Index Selection Guidelines – 3/3

• Try to choose indexes that benefit as many queries as possible
  – Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering

• Note: clustered index should be used judiciously
  – expensive updates, although cheaper than sorted files
Examples of Clustered Indexes

- B+ tree index on E.age can be used to get qualifying tuples

- How selective is the condition?
  - everyone > 40, index not of much help, scan is as good
  - Suppose 10% > 40. Then?

- Depends on if the index is clustered
  - otherwise can be more expensive than a linear scan
  - if clustered, 10% I/O (+ index pages)

What is a good indexing strategy?

```sql
SELECT E.dno
FROM Emp E
WHERE E.age > 40
```
Examples of Clustered Indexes

Group-By query

• Use $E.age$ as search key?
  – Bad If many tuples have $E.age > 10$ or if not clustered....
  – ...using $E.age$ index and sorting the retrieved tuples by $E.dno$ may be costly

• Clustered $E.dno$ index may be better
  – First group by, then count tuples with age > 10
  – good when age > 10 is not too selective

• Note: the first option is good when the WHERE condition is highly selective (few tuples have age > 10), the second is good when not highly selective

SELECT $E.dno$, COUNT (*)
FROM Emp $E$
WHERE $E.age > 10$
GROUP BY $E.dno$

What is a good indexing strategy?
Examples of Clustered Indexes

Equality queries and duplicates

• Clustering on \textit{E.hobby} helps
  – hobby not a candidate key, several tuples possible

• Does clustering help now?
  – Not much
  – at most one tuple satisfies the condition

What is a good indexing strategy?

\begin{verbatim}
SELECT E.dno
FROM Emp E
WHERE E.hobby='Stamps'
\end{verbatim}

\begin{verbatim}
SELECT E.dno
FROM Emp E
WHERE E.eid=50
\end{verbatim}
Indexes with 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 \(<\text{sal}, \text{age}>\) index:
  - age=20 and sal =75

- **Range query**: Some field value is not a constant. E.g.:
  - sal > 10
  - \(<\text{age}, \text{sal}>\) does not help
  - has to be a prefix

Examples of composite key indexes using lexicographic order.

Data records sorted by name

Data entries in index sorted by \(<\text{sal}, \text{age}>\)

Data entries sorted by \(<\text{sal}>\)
Composite Search Keys

- To retrieve Emp records with \( age=30 \text{ AND } sal=4000 \), an index on \(<age,sal>\) would be better than an index on \( age \) or an index on \( sal \)
  - first find \( age = 30 \), among them search \( sal = 4000 \)

- If condition is: \( 20<age<30 \text{ AND } 3000<sal<5000 \):
  - Clustered tree index on \(<age,sal>\) or \(<sal,age>\) is best.

- If condition is: \( age=30 \text{ AND } 3000<sal<5000 \):
  - Clustered \(<age,sal>\) index much better than \(<sal,age>\) index
  - more index entries are retrieved for the latter

- Composite indexes are larger, updated more often
Index-Only Plans

- A number of queries can be answered without retrieving any tuples from one or more of the relations involved if a suitable index is available.

\[
\text{SELECT E.dno, COUNT(*) FROM Emp E GROUP BY E.dno}
\]

\[
\text{SELECT E.dno, MIN(E.sal) FROM Emp E GROUP BY E.dno}
\]

\[
\text{<E.dno,E.sal>}
\]

\[
\text{Tree index!}
\]

\[
\text{SELECT AVG(E.sal) FROM Emp E WHERE E.age=25 AND E.sal BETWEEN 3000 AND 5000}
\]

\[
\text{<E.age,E.sal>}
\]

\[
\text{Tree index!}
\]