CS3223 Database Systems Implementation

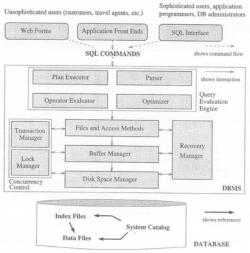
AY23/24 Sem 2, github.com/gerteck

Introduction

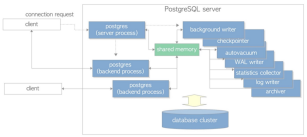
Course Details

- Prerequisite knowledges: CS2040S, CS2102, CS2106 background (helpful).
- Reference Textbook: Raghu & Johannes Database M. Systems, 2002. Encouraged to read ahead based on schedule before the lecture.
- Course covers data structures, algorithms, different components making up database systems.

Architecture of DBMS



Process Architecture in PostgreSQL



Source: H. Suzuki's The Internals of PostgreSQL

Source: Ramakrishnan & Gehrke's Figure 1.3

- OLTP: Online Transaction Processing is a type of data processing that consists of executing a number of transactions occurring concurrently—online banking, shopping, order entry, or sending text messages, for example.
- OLAP: Online Analytical Processing.
- Focusing on centralized database running on a single server.

1. Data Storage

References: R&G Chapt 8. (Storage & Indexing Overview), Chapt 9. (Storing Data: Disks and Files).

A DBMS stores

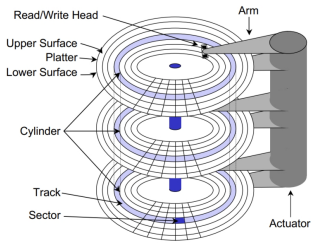
- Relations (Actual tables)
- System catalog (aka data dictionary) storing metadata about relations. (Relation schemas, structure of relations, constraints, triggers. View definitions, Indexes - derived info to speed up access to relations, Statistical information about relations for use by query optimizer.)
- Log files: Information maintained for data recovery.

DBMS Storage

Memory Hierarchy: Primary (registers, RAM), secondary (HDD, SSD), tertiary memory with capacity / cost / access speed / volatility tradeoffs.

- DBMS stores data on non-volatile disk for persistence.
- DBMS processes data in main memory (RAM).
- Disk access operations (I/O). Read: transfer data from disk to RAM. Write: transfer data from RAM to disk.
- Make use of index to speed up access, so that don't have to retrieve all the data when you run a query. Retrieve index and read only the block that contains specified data. Minimize I/O cost.

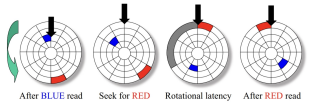
Magnetic Hard-Disk Drive HDD



Source: R. Burns' slides on storage systems

- Cylinder, Track, Sector: Units of the HDD storage system. To read from different tracks, need to move the mechanical HDD arm.
- **Disk Access Time**:
  - command processing time: interpreting access command by disk controller.
  - seek time: moving arms to position disk head on track.
  - rotational delay: waiting for block to rotate under head.
  - transfer time: actually moving data to/from disk surface.
  - **access time**: = seek time + rotational delay + transfer time. (CPT considered negligible).

Disk Access Time Components



- Seek time
  - avg. seek time: 5-6 ms
- Rotational delay (or rotational latency)
  - Depends on rotation speed - measured in rotations per minute (RPM)
  - Average rotational delay = time for $\frac{1}{2}$ revolution
  - Example: For 10000 RPM, avg. rotational delay = 0.5 (60 / 10000) = 3 ms
- Transfer time
  - n = number of requested sectors on same track
  - transfer time = $n \times \frac{\text{time for one revolution}}{\text{number of sectors per track}}$
  - avg. sector transfer time: 100-200 µs
- Sequential vs random I/O

Source: R. Burns' slides on storage systems

- **Concept of Sequential vs random I/O.**
  - **Sequential:** Both sector on same track.
  - **Random:** Sectors on different track, require seeking (moving arm).
- Given a set of data, we hope to store the data contiguously, on the same track. (Minimize incurring random I/O). If data is too large, store on same track, but different surface (aka same cylinder).
- Complexity hidden to OS by disk controller. Shown as a sequence of memory locations.

Solid-State Drive: SSD

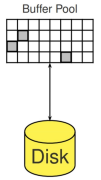
- Build with NAND flash memory without any mechanical moving parts. Lower power consumption.
- **Random I/O**: 100x faster than HDD. (no moving parts)
- **Sequential I/O**: slightly faster than HDD (2x)
- **Disadvantages**: update to a page requires erasure of multiple pages (5ms) before ovewriting page. Limited number of times a page can be erased ($10^5 - 10^6$)

Storage Manager Components

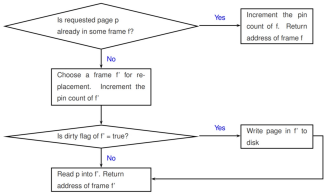
- Data is stored, retrieved in units called **disk blocks (or pages**. (Each block = sequence of one or more contiguous sectors.
- **FIles & access methods layer (aka file layer)** - deals with organization and retrieval of data.
- **Buffer Manager** - controls reading/writing of disk pages.
- **Disk Space Manager** - keeps track of pages used by file layer.

Storage Manager Components

Buffer Manager



BM: Handling request for page p

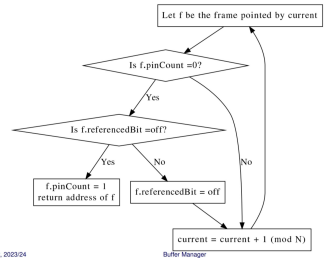


Buffer Manager

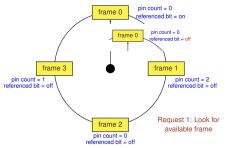
- **Buffer pool**: Main memory allocated for DBMS.
- Buffer pool is partitioned into block-sized pages called **frames**.
- Clients of buffer pool can request for disk page to be fetched into buffer pool, release a disk page in buffer pool.
- A page in the buffer is **dirty** if it has been modified & not updated on disk.
- **Two variables** maintained for each frame in buffer pool:
  - **pin count**: number of clients using page (initialized 0)
  - **dirty flag**: whether page is dirty (initialized false)
- Free list: Keeps track of frames that are free / empty.
- **Pin count**:
  - Incrementing pin count is **pinning** the requested page in its frame.
  - Decrementing is **unpinning** the page.
  - Unpinning a page, dirty flag should be updated to true if page is dirty.
  - A page in buffer can be replaced only when pin count is 0.
  - Before replacing buffer page, needs to be written back to disk if its dirty flag is true.
- Buffer manager coordinates with transaction manager to ensure data correctness and recoverability.
- **Replacement Policies**
  - Replacement policy: Deciding which unpinned page to replace. (some examples:)
  - Random, FIFO, Most Recently Used (MRU), Least Recently Used (LRU): (Use queue of pointers to frames with pin count = 0), most common, makes use of temporal locality.
  - **Clock**: cheaper popular variant of LRU
    - **current** variable: points to some buffer frame.
    - Each frame has a **referenced bit**, turns on when its pin count turns 0.
    - Replace a page that has referenced bit off & pin count = 0.

Clock Replacement Policy

N = number of frames in buffer pool



Clock Replacement Policy: Example



Files

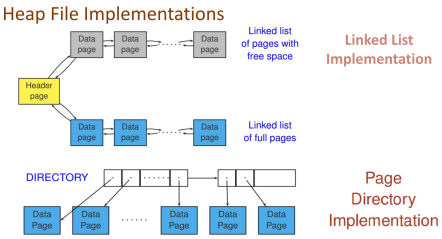
File Abstraction

- Each relation is a file of records.
- Each record has a unique record identifier called RID / TID.
- Common file operations: create/delete file, insert record, delete/get record with given RID, scan all records.

File Organization: Method of arranging data records in a file that is stored on disk.

- **Heap file**: Unordered file
- **Sorted file**: Records order on some search key.
- **Hashed file**: Records located in blocks via a hash function.

Heap File Implementations

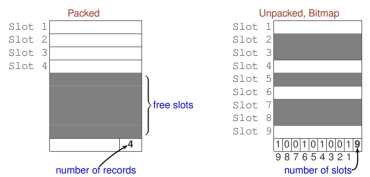


- **Linked list implementation**: Two linked lists, one with pages with free space, other of completely full pages.
- **Page Directory Implementation**: Two leveled implementation. Each big block is a disk block with some metadata. Each disk block has a number of data pages.

Page Formats:
Records are organized within a page and referenced with the RID.

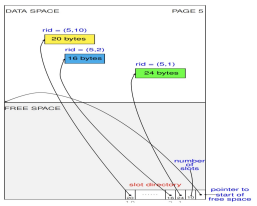
- **RID = (page id, slot number)**
- For **Fixed-Length Records**, Organization can be:
  - **Packed Organization**: Store records in contiguous slots.
  - For packed organization, memory organization is tough and costly when record in slot is deleted, need to move up a record. But as RID serves as a reference, but need to propagate change in RID.
  - **Unpacked Organization**: Uses bit array to maintain free slots.
  - For unpacked organization, more bookkeeping needed (use bitmap, 1 & 0 to check if occupied) to store records.
- For **Variable-Length Records**: We could assume some maximum size, then use packed organization. But wasteful. Instead, we can use **Slotted Page Organization**.

Fixed-Length Records:



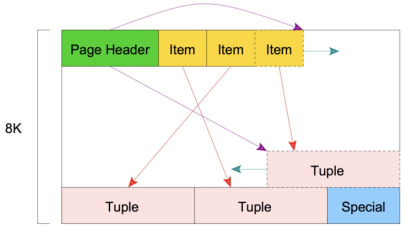
Packed / Unpacked Organization

Variable-Length Records:



Slotted Page Organization

PostgreSQL's Slotted Page Organization



Source: B. Momjian's slides on PostgreSQL internals

Record Formats: Organizing fields within a record.

- Fixed-Length Records
  - Fields are stored consecutively

F1	F2	F3	F4

- Variable-Length Records
  - Delimit fields with special symbols

F1	$	F2	$	F3	$	F4

  - Use an array of field offsets

o_1	o_2	o_3	o_4	F1	F2	F3	F4

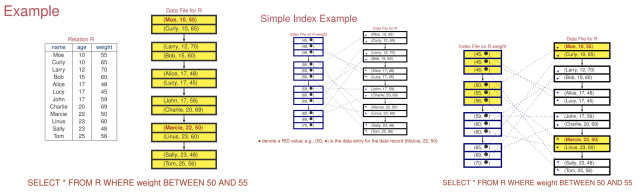
Each o_i is an offset to beginning of field Fi

2. Indexing

Need some auxiliary data structure to make efficient queries.

Index

- An **index** is a data structure to speed up retrieval of data records based on some search key.
- A **search key** is a sequence of k data attributes, $k \geq 1$. (A search key is aka *composite search key* if $k > 1$, e.g. (state, city).)
- An index is a **unique index** if search key is a candidate key, otherwise it is **non-unique index**.
- An index is stored as a file, records in index file referred to as **data entries**.



w/o index, need to retrieve all pages/records

w. simple index, only require retrieval of select pages

Index Types

Two main types of indexes

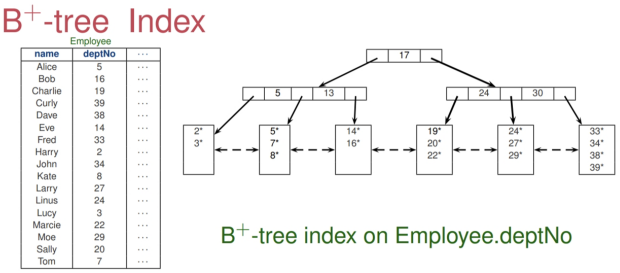
- **Tree-based Index**: Based on sorting of search key values (E.g. ISAM, B^+-tree)
- **Hash-based Index**: Data entries accessed using hashing function (E.g. static/ extendible / linear hashing)
- Considerations when choosing an index:
  - Search Performance
    (Equality search: $k = v$, use hash-based.) (Range search, use tree)
  - Storage overhead
  - Update performance

Tree-basd Indexing: B^+-Tree

B^+ tree is a dynamic structure that adjusts to changes in the file gracefully, most widely used index structure as it adjusts well to changes and supports both equality and range queries.

- **Balanced tree**: Operations (insert, delete) on tree keep it balanced.
- **Internal nodes** direct the search.
- **Leaf nodes** contain the data entries. Leaf pages linked using page pointers for easy traversal of sequence of leaf pages in either direciton.
- **Value** d is parameter of B^+-tree, called order of the tree, is a measure of capacity of a tree node. Each node contains m entries, where $d \leq m \leq 2d$, except root node, where $1 \leq m \leq 2d$

B^+-tree Index

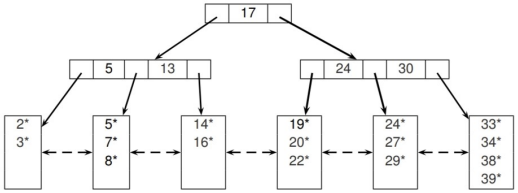


- Each node is either a **leaf node** (bottom-most level) or an internal node.
- Top-most internal node is the **root node** located at **level 0**.
- **Height of Tree** = number of level of internal nodes. (Leaf nodes are at level h where h = height of tree.
- Nodes at same level are **sibling nodes** if they have the same parent node.
- **Leaf Nodes**:
  - Leaf nodes store sorted data entries.
  - $k*$ denote data entry of form (k, RID), where k = search key value of corresponding data record, RID = RID of data record.
  - Lead nodes are doubly-linked to adjacent nodes.
- **Internal Nodes**:
  - Internal nodes store index entries of the form (p: pointer, k: separator) $(p_0, k_1, p_1, k_2, p_2, ..., p_n)$
  - $k_1 < k_2 < ... < K_n$
  - Each (k_i, p_i) is an **index entry**, k_i serves as **separator** between node contents pointed to by p_{i-1} & p_i
  - p_i = disk page address (root node of an index subtree T_i)

B^+-tree Index Properties

Properties of B^+-tree Index

- Dynamic index structure; adapts to data updates gracefully
- Height-balanced index structure
- Order of index tree, $d \in Z^+$
  1. Controls space utilization of index nodes
  2. Each non-root node contains m entries, where $m \in [d, 2d]$
  3. The root node contains m entries, where $m \in [1, 2d]$
- **Example:** B^+-tree with order = 2



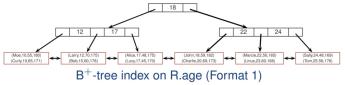
Formats of Data Entries in B-Tree

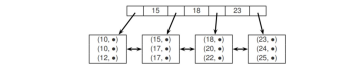
- **Format 1**: k* is actual *data record* (with search key value k)
- **Format 2**: k* is of form *(k, rid)*, where rid is record identifier of record with search key value k.
- **Format 3**: k* is of form *(k, rid-list)*, where rid-list is list of record identifiers of data records with search key value k.
- Note, examples assume Format 2.

Formats of Data Entries: Example

Relation R

name	age	weight	height
Moe	10	55	180
Curly	10	65	171
Larry	12	70	175
Bob	15	60	178
Alice	17	48	175
Lucy	17	45	170
John	18	59	182
Charlie	20	69	173
Marcie	22	50	165
Linus	23	60	166
Sally	24	48	169
Tom	25	56	176

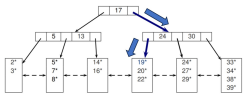




B^+-tree Search Algorithms

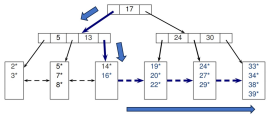
- Search algorithm finds the leaf node a given data entry belongs to.
- We assume no duplicates, no data entries same key value. Note in practice, duplicates arise whenever search key does not contain candidate key, must be dealt with.

Equality Search ($k = 19$)



- At each internal node N, find the largest key k_i in N s.t. $k \geq k_i$
  - If k_i exists, then search subtree at p_i
  - Otherwise, search subtree at p_0

Range Search ($k \geq 15$)



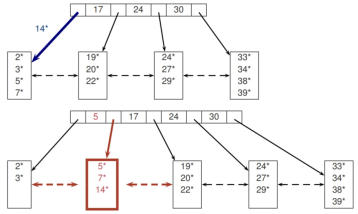
B^+-Tree Insertion

- **Algorithm for insertion** takes an entry, finds the leaf node where it belongs, and inserts it there.
- Occasionally, a **node is full and must be split**. (More than $2d$ entries) When node is split, entry pointing to the node created by the split must be inserted into the parent.
- If the (old) root is split, a new root node is created and height of tree increases by 1.

Splitting of overflowed node

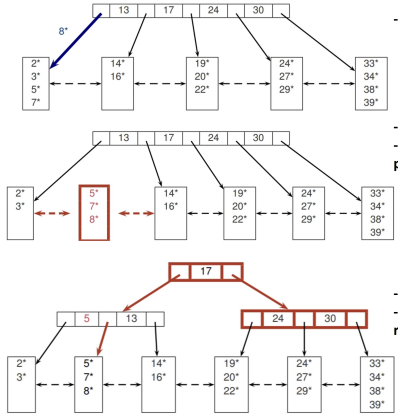
- Split overflowed leaf node by distributing $d + 1$ entries to new leaf node.
- Create a new entry index using smallest key in leaf node.
- Insert new index entry into parent node of overflowed node.

Inserting 14* (Splitting of overflowed node)

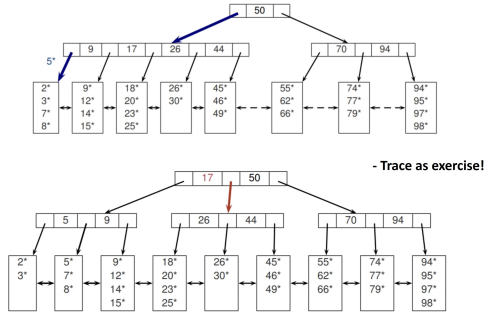


- Sometimes, **node split is propagated upwards** to ancestor internal nodes.
- When splitting an **internal node**, the middle key is pushed to parent node.

Inserting 8* (Propagation of node splits)



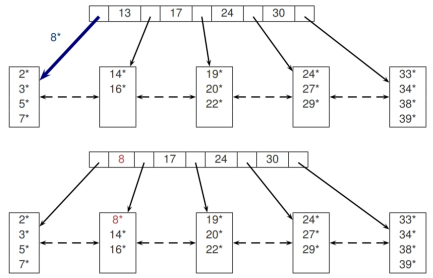
Inserting 5* (Propagation of node splits)



Redistributing of data entries in Overflow

- A node split can sometimes be avoided by distributing entries from overflowed node to a non-full adjacent sibling node.

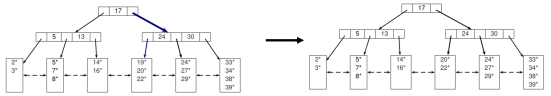
Inserting 8* (Redistribution of data entries)



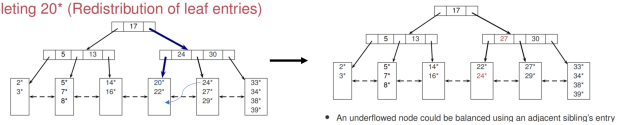
B^+-Tree Deletion

- **Algorithm for deletion** takes an entry, finds leaf node it belongs to, and deletes it.
- **Underflowed node**: When node is at minimum occupancy before deletion, and goes below threshold, we must either **redistribute entries from adjacent sibling**, or **merge node with sibling** to maintain minimum occupancy.
- **Merging**: Underflowed node needs to be merged if each of **adjacent** sibling nodes has exactly d entries.

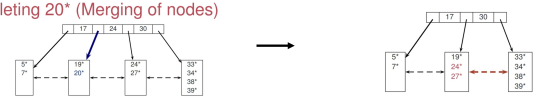
Deleting 19* (Simple Case)



Deleting 20* (Redistribution of leaf entries)

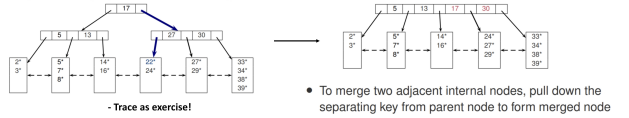


Deleting 20* (Merging of nodes)

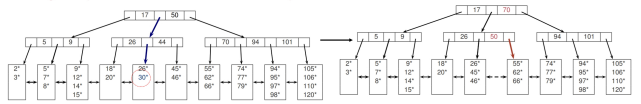


- Node mergers may propagate upwards.

Deleting 22* (Propagation of node merges)



Deleting 30* (Redistribution of internal entries)



- Chances are high that redistribution is possible if node has two siblings, and unlike merging, redistribution is guaranteed to propage no further than parent node. Also, pages have more space, reducing likelihood of split on subsequent insertions.

B^+-Tree Bulk Loading

- Entries added to a B^+ in two ways.

1. Have existing collection of data records with B^+ tree index on it. When record added to collection, corresponding entry added to B^+ tree. (Insert, Delete individually)
2. Have collection of data records we want to create new B^+ tree index on some key field(s). Start with an empty tree. Inserting one by one expensive dur to overhead, systems provide **bulk loading** utility.

- **Bulk Loading**:

1. Sort data entries $k*$ to be inserted into B^+ tree according to search key k. (Here, $d = 1$).
2. Allocate empty page to serve as root. Insert a pointer to first page of (sorted entries into it).

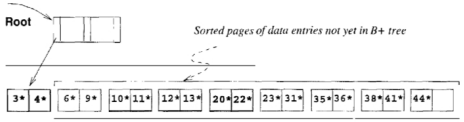


Figure 10.22 Initial Step in B+ Tree Bulk-Loading

3. Add one entry to root page for each page of sorted data entries. Proceed until root page is full. Here, we must split root and create a new root page.

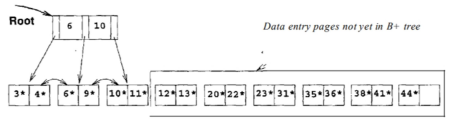


Figure 10.23 Root Page Fills up in B+ Tree Bulk-Loading

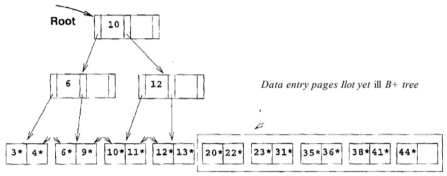


Figure 10.24 Page Split during B+ Tree Bulk-Loading

4. To continue, entries for leaf pages **always inserted into right-most index page just above the leaf level**. When right-most page above leaf level fills up, it is split.

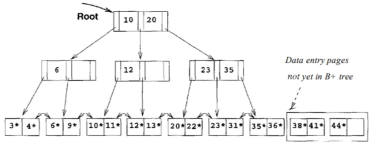
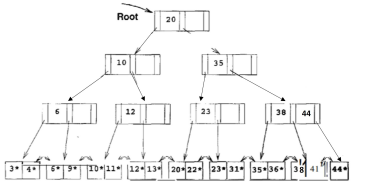


Figure 10.25 Before Adding Entry for Leaf Page Containing 38*



3. Hash-based Indexing

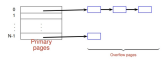
- Used for **equality queries**, not for range queries.
- **Hashing techniques**: **Static hashing, dynamic hashing** (linear hashing, extendible hashing, etc).

Static Hashing

- **Data stored in N buckets**, fixed at creation time. Bucket consists of **one primary data page & chain** of zero or more overflow data pages.
- $v*$ represents data entry e with h(e.key) = v, not search entry with RID.
- **Problem with static hashing**: As data grows, longer overflow chain, efficiency drops. Need to periodically rehash and increase no. of buckets.

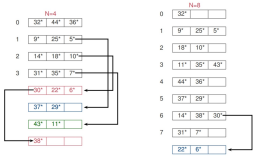
Static Hashing

- Data is stored in N buckets $B_0, B_1, \cdots, B_{N-1}$
  - N is fixed at creation time
- Hashing function $h(.)$ is used to identify the bucket to store a record
  - A record with search key k is inserted into bucket B_i, where $i = h(k) \bmod N$
  - $h(k)$ maps the search key value into a bit string
- Each bucket consists of one primary data page & a chain of zero or more overflow data pages



Static Hashing: Example

v^* denote a data entry e with $h(e.key) = v$
$e.key$ denote the index's search key value of e



Dynamic Hashing: Linear Hashing

- Hash file grows / shrinks linearly, systematic **splitting of buckets**.
- Overflow pages needed as overflowed bucket may not be split immediately.
- Hashing function changes dynamically and at given instant, **at most two (successive) hashing functions** used by the scheme during search.
- Each bucket has primary data page & chain of zero+ overflow pages.
- Insert in bucket B_i overflows if all pages in B_i (primary + overflow) full.

Linear Hashing (cont.)

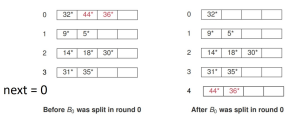
- Assume initial file size of N_0 buckets
  - Buckets $B_0, B_1, \cdots B_{N_0-1}$
- File grows linearly by splitting buckets in rounds
  - At each round, buckets are split sequentially: B_i is split before B_{i+1}
- How to split a bucket B_i?
  - Add a new bucket B_j (known as split image of B_i)
  - Redistribute entries in B_i between B_i & B_j
- File size increases by one bucket after each bucket split
- At the end of one round of splitting (i.e., every bucket at the start of the round has been split), file size is doubled
- Let N_i denote the file size at the beginning of round i (i=0,1,$\cdots$)
  - $N_i = 2^i N_0$
- At the end of round i, N_i new buckets are added: $B_{N_i}, B_{N_i+1}, \cdots, B_{2N_i-1}$
  - In round i, the split image of B_j is B_{N_i+j} for $j \in [0, N_i - 1]$

Splitting Buckets

- Assume $N_0 = 4$
- In round 0, split image of bucket B_i is B_{4+i}, for $i \in [0,3]$
- In round 1, $N_1 = 8$ & split image of bucket B_j is B_{8+j}, for $i \in [0,7]$

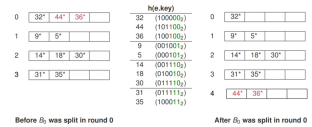
Dynamicity of Linear Hashing

Dynamic Hashing



- Recall that v^* denote a data entry e with $h(e.key) = v$ and $e.key$ denote the index's search key value of e
- Each round uses two hash functions: functions h_i and h_{i+1} for round i
  - $h_i(k) = h(k) \bmod N_i$
  - B_x is the bucket for search key k if B_x had not been split, where $x = h_i(k)$
  - B_y is the bucket for search key k if B_x had been split, where $y = h_{i+1}(k)$
- Keep track of which bucket to be split next using variable next

Redistributing Entries

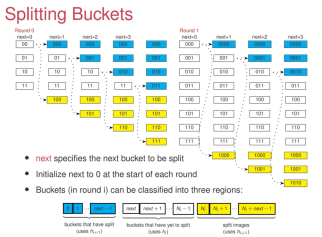


- For convenience, $N_0 = 2^m$ for $m \in \{0, 1, \cdots\}$
- $N_i = 2^i N_0 = 2^{m+i}$
- $h_i(k) = h(k) \bmod N_i$ = value of last (m+i) bits of h(k)
- Consider the splitting of bucket B_j in round i
  - Before splitting, all entries e in B_j have $h_i(e.key) = j$ (i.e., same last (m+i) bits)
  - After splitting, e remains in B_j iff the last (m+i+1) bit of $h(e.key)$ is 0

Linear (Hashing) Splitting of Buckets (Insertion)

- Number in buckets represent the rightmost bit values.
- **Split Criteria**: variable, could be when some bucket overflow, space utilization of file above some threshold etc.
- **Level**: We use level to denote splitting round number (use with **next**).

Splitting Buckets



- next specifies the next bucket to be split
- Initialize next to 0 at the start of each round
- Buckets (in round i) can be classified into three regions:

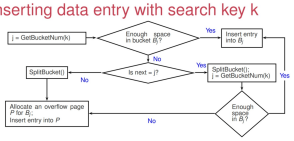
When to split a bucket?

- The time to split the next bucket can be decided with various criteria:
  - Split whenever some bucket overflows
  - Split whenever space utilization of file is above some threshold
  - etc.
- Overflow pages are needed since an overflowed bucket might not be split immediately
- We shall assume that a bucket split is triggered whenever some bucket overflows
  - Bucket B_j overflows if an entry is to be inserted into B_j and all the pages in B_j (i.e., primary & overflow pages) are full
- We use **level** to denote the splitting round number

Examples: Linear Hashing Search & Insert

- Even though overflowed bucket split, not neccessary mean enough space if bit value not right. Still require overflow page.
- Using **level and next**, we determine how many additional bits to consider in **second hashing function**.
- **Second hashing function**: Is simply looking at the m rightmost bits of the hash.

Inserting data entry with search key k



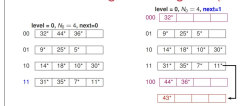
- **GetBucketNum(k)** returns bucket # where entry with search key k is located

$$GetBucketNum(k) = \begin{cases} h_{next}(k) & \text{if } h_{new}(k) \geq next, \\ h_{next+1}(k) & \text{otherwise.} \end{cases}$$

- **SplitBucket()** splits bucket B_{next}
  1. Redistribute the entries in B_{next} into B_{next} & $B_{next+N_{next}}$ using $h_{level+1}()$
  2. next = next + 1
  3. If (next = N_{next}) then { level = level + 1; next = 0 }

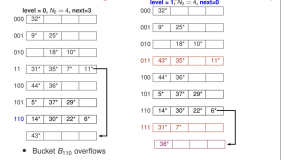
Example:

Linear Hashing: Inserting 43* (101011)



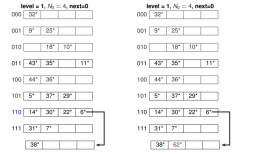
- Bucket B_{11} overflows
- Split bucket B_{00} (32 = 100000, 36 = 100100, 44 = 101100)
- Increment next to 1
- Insert 43* into overflow page

Linear Hashing: Inserting 38* (100110)



- Bucket B_{10} overflows
- Split bucket B_{11} (7 = 111, 11 = 1011, 31 = 11111, 35 = 100011, 43 = 101011)
- Increment level to 1; reset next to 0
- Insert 38* into overflow page

Linear Hashing: Inserting 62* (111110)



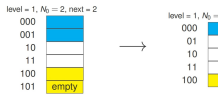
- No overflow, no splitting
- Insert 62* into bucket B_{110}

Linear Hashing Deletion

- Opposite of insertion.
- **Two cases**: 1. (Next ¿ 0), 2. (Next = 0, and level $>$ 0.)

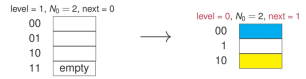
Linear Hashing: Deletion

- Locate bucket and delete entry
- If the last bucket $B_{N_{level}+next-1}$ becomes empty, it can be removed
- **Case 1**: If *next* > 0
  - Decrement `next` by one



Linear Hashing: Deletion (cont.)

- **Case 2**: If (*next* $= 0$) and (*level* > 0)
  - Update `next` to point to the last bucket in previous level $B_{N_{level-1}-1}$
  - Decrement `level` by one



Linear Hashing Performance, Summary

Linear Hashing: Performance

- One disk I/O unless the bucket has overflow pages
  - On average 1.2 disk I/O for uniform data distribution
  - Worst case: I/O cost is linear in the number of data entries
- Poor space utilization with skewed data distribution

Linear Hashing: Summary

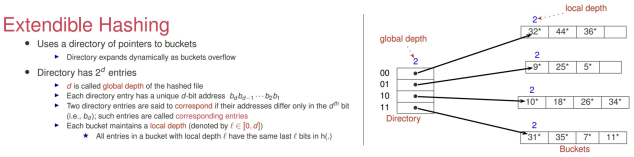
- File grows dynamically by splitting buckets linearly in rounds
  - Each bucket split adds one new bucket to file
- The bucket B_j for a search key k is determined using two hash functions (dependent on round)
  - Assume initial file has 2^m buckets & current round number is level
  - j = last (m+level) bits of h(k) if B_j has not yet been split in current round; otherwise, j = last (m+level+1) bits of h(k)
- In this course, we assume a bucket split is triggered whenever some bucket overflows due to an insertion
- Overflow pages are required for an overflowed bucket if it is not the next bucket to be split

Dynamic Hashing: Extendible Hashing

- **Similar to Linear Hashing**: we want **bucket number to grow dynamically**, and use some **number of least significant bits** of h(k) to determine bucket address for search key k.
- **Difference**: Add new bucket (as split image) when existing bucket overflows, No overflow pages (except when number of collisions exceed page capacity, two page entries collide if they have same h(.) hash value).

Extendible Hashing

- **Extendible hashing**: dynamically updateable disk-based index structure, implements hashing scheme utilizing a **directory of pointers to buckets**.
- Overflows handled by doubling the directory which logically doubles the number of buckets. **Physically, only the overflown bucket is split**.



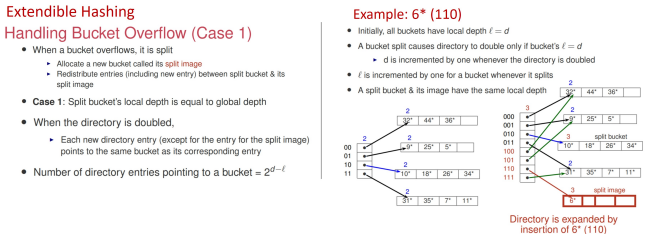
Extendible Hashing Performance

- **Performance**: At most 2 disk I/O for equality selection, at most 1 I/O if directory fits in main memory.
- **Handling collision**: Two data entries **collide** if same hashed value, overflow pages need when number of collisions exceed page capacity.
- Compared with B+-tree index exact match queries (log number of I/Os), E. Hashing better expected query cost O(1) I/O.

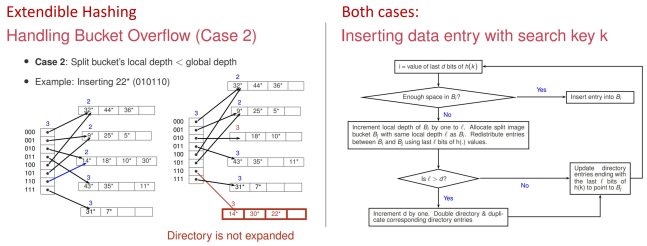
Extendible Hashing: Handling Bucket Overflow

- Main idea: Determine if there is empty directory entry to point to new bucket. **2 Cases**: decision to split, or use empty directory entry.

Case 1: Split bucket local depth = global depth



Case 2: Split bucket local depth $<$ global depth.



Extendible Hashing Deletion

- To delete entry, simply locate Bucket and delete.
- **Merging**: Mergable if entries can fit within a bucket, and same local depth, j differs on in l^{th} bit.