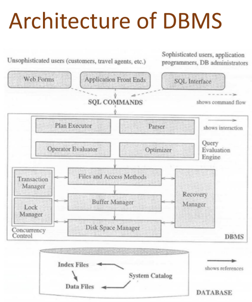
CS3223 Database Systems Implementation

AY23/24 Sem 2, github.com/gerteck

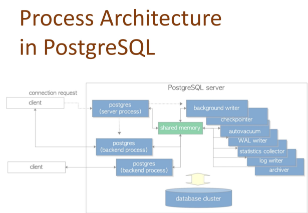
Introduction

Course Details

- Prerequisite Knowledge: 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.



Source: Ramakrishnan & Gehrke's Figure 1.3



Source: H. Suzuki's The Internals of PostgreSQL

- 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

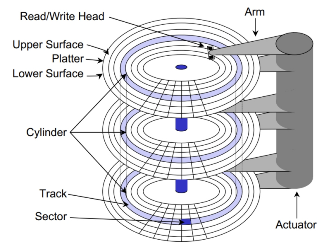
- Relations (Actual tables)
- System catalog (aka data dictionary) storing metadata about relations. (Relation schemas, structure of relations, constraints, triggers. View definitions, Indexes - derived info to speed up access to relations, Statistical information about relations for use by query optimizer.)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

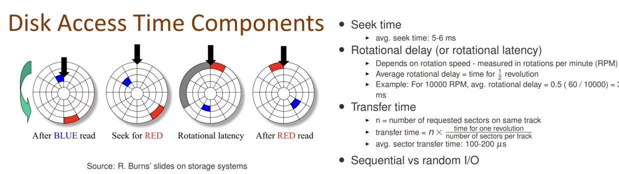
- DBMS stores data on non-volatile disk for persistence.
- DBMS processes data in main memory (RAM).
- Disk access operations (I/O). Read: transfer data from disk to RAM. Write: transfer data from RAM to disk.
- Make use of index to speed up access, so that don't have to retrieve all the data when you 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).**



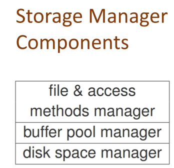
- **Concept of Sequential vs random I/O.**
  - **Sequential:** Both sector on same track.
  - **Random:** Sectors on different track, require seeking (moving arm).
- Given a set of data, we hope to store the data contiguously, on the same track. (Minimize incurring random I/O). If data is too large,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

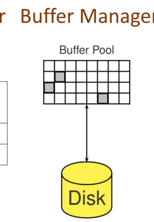
Solid-State Drive: SSD

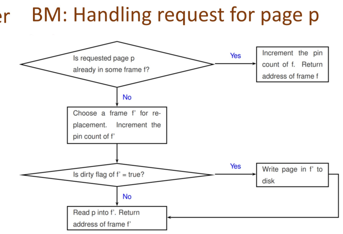
- Build with NAND flash memory without any mechanical moving parts. Lower power consumption.
- **Random I/O**: 100x faster than HDD. (no moving parts)
- **Sequential I/O**: slightly faster than HDD (2x)
- **Disadvantages**: update to a page requires erasure of multiple pages (5ms) before ovewriting page.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

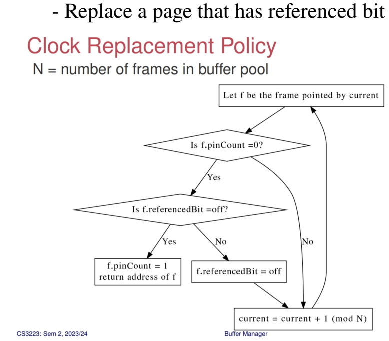


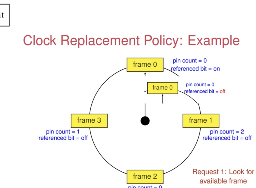




Buffer Manager

- **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.





Files

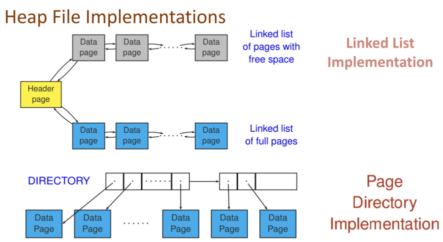
File Abstraction

- 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.

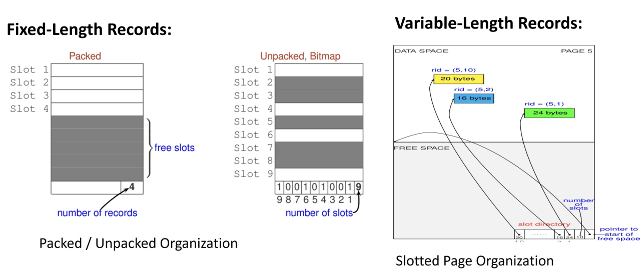
Heap File Implementations

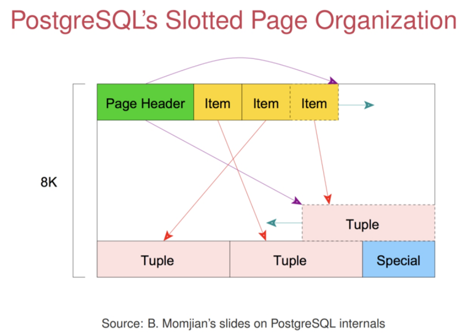


- **Linked list implementation**: Two linked lists, one with pages with free space, other of completely full pages.
- **Page Directory Implementation**: Two leveled implementation. Each big block is a disk block with some metadata. Each disk block has a number of data pages.

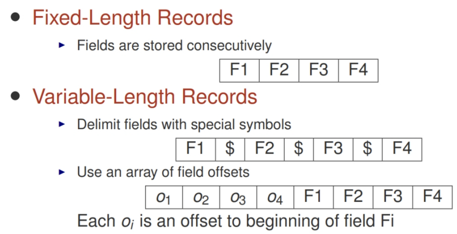
Page Formats:
Records are organized within a page and referenced with the RID.

- **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**.





Record Formats: Organizing fields within a record.

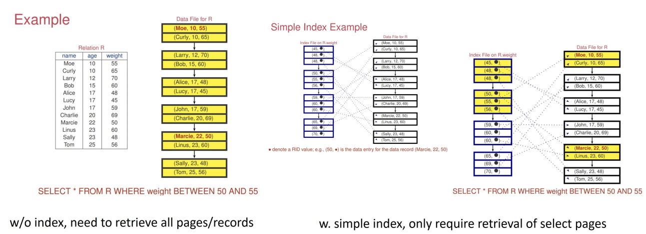


2. Indexing

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**.



Index Types

Two main types of indexes

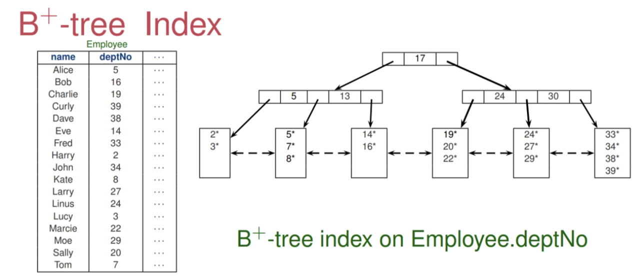
- **Tree-based Index**: Based on sorting of search key values (E.g. ISAM, B^+-tree)
- **Hash-based Index**: Data entries accessed using hashing function (E.g. static/ extendible / linear hashing)
- Considerations when choosing an index:
  - Search Performance
    (Equality search: $k = v$, use hash-based.)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

Tree-based 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 direction.
- **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

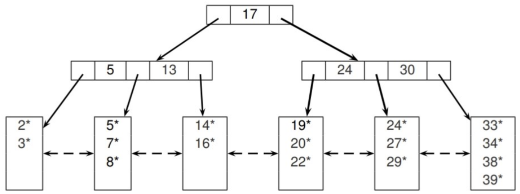


- Each node is either a **leaf node** (bottom-most level) or an internal node.
- Top-most internal node is the **root node** located at **level 0**.
- **Height of Tree** = number of level of internal nodes. (Leaf nodes are at level h where h = height of tree).
- Nodes at same level are **sibling nodes** 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



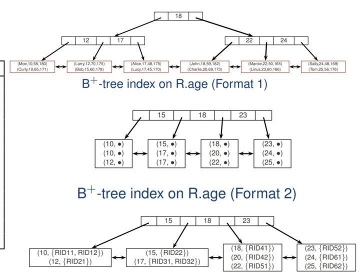
Formats of Data Entries in B-Tree

- **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 index on R.age (Format 1)

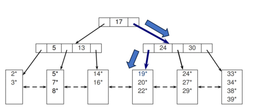
B^+-tree index on R.age (Format 2)

Index on R.age (format 3)

B^+-tree Search Algorithms

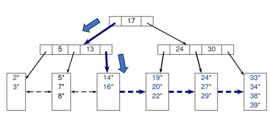
- Search algorithm finds the leaf node a given data entry belongs to.
- We assume no duplicates, no data entries same key value. Note in practice, duplicates arise whenever search key does not contain candidate key, must be dealt with.

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 ($k \geq 15$)



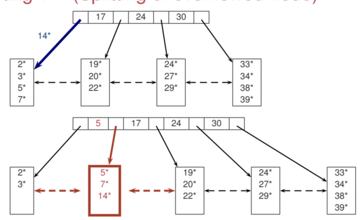
B^+-Tree Insertion

- **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

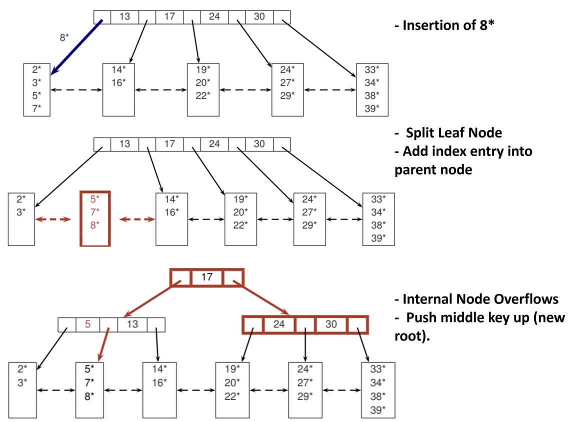
- Split overflowed leaf node by distributing $d + 1$ entries to new leaf node.
- Create a new entry index using smallest key in leaf node.
- Insert new index entry into parent node of overflowed node.

Inserting 14* (Splitting of overflowed node)

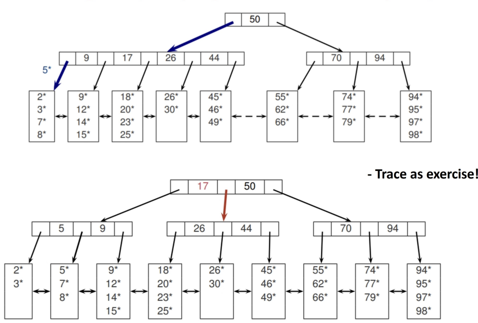


- Sometimes, **node split is propagated upwards** to ancestor internal nodes.
- When splitting an **internal node**, the middle key is pushed to parent node.

Inserting 8* (Propagation of node splits)



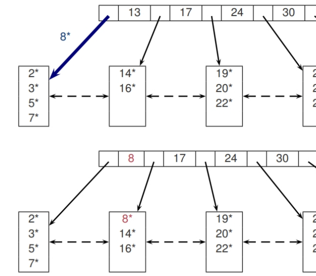
Inserting 5* (Propagation of node splits)



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.

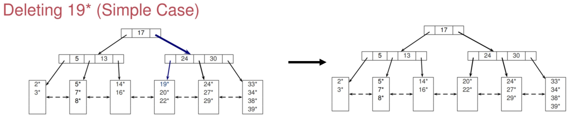
Inserting 8* (Redistribution of data entries)



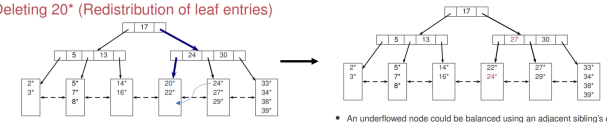
B^+-Tree Deletion

- **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)

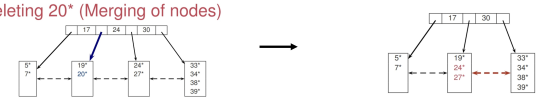


Deleting 20* (Redistribution of leaf entries)



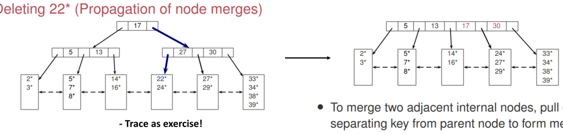
- An underflowed node could be balanced using an adjacent sibling's entry

Deleting 20* (Merging of nodes)



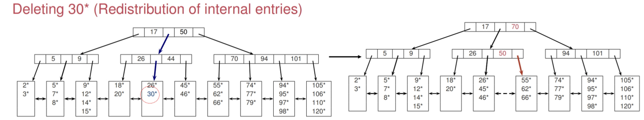
- Node mergers may propagate upwards.

Deleting 22* (Propagation of node merges)



- To merge two adjacent internal nodes, pull down the separating key from parent node to form merged node

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 propagate 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 due 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).

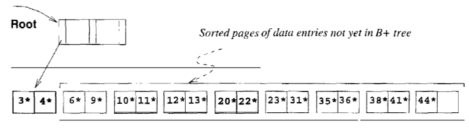


Figure 10.22 Initial Step in B+ Tree Bulk-Loading

  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.

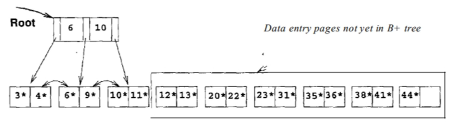


Figure 10.23 Root Page Fills up in B+ Tree Bulk-Loading

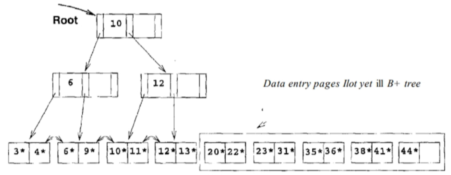


Figure 10.24 Page Split during B+ Tree Bulk-Loading

  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.

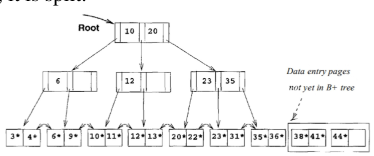
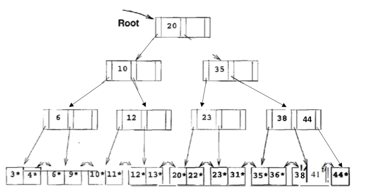


Figure 10.25 Before Adding Entry for Leaf Page Containing 38*

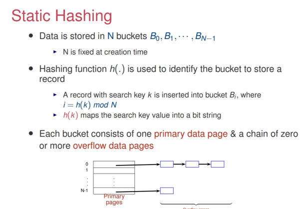


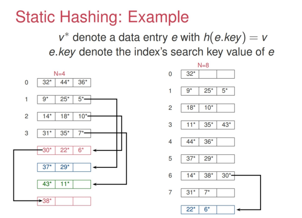
3. Hash-based Indexing

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

Static Hashing

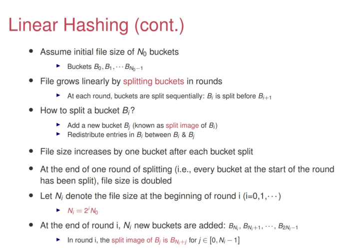
- **Data stored in N buckets**, fixed at creation time. Bucket consists of **one primary data page & chain** of zero or more overflow data pages.
- $v*$ represents data entry e with h(e.key) = v, not search entry with RID.
- **Problem with static hashing**: As data grows, longer overflow chain, efficiency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

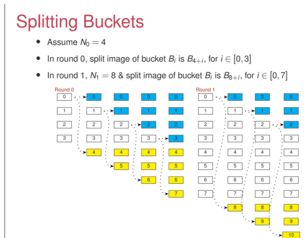




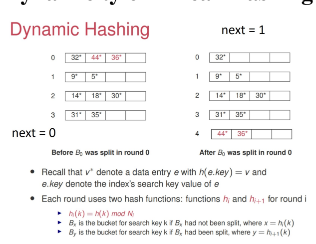
Dynamic Hashing: Linear Hashing

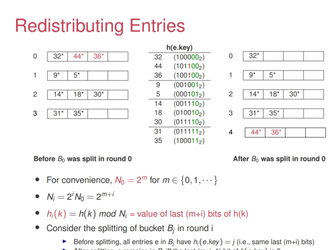
- Hash file grows / shrinks linearly, systematic **splitting of buckets**.
- Overflow pages needed as overflowed bucket may not be split immediately.
- Hashing function changes dynamically and at given instant, **at most two (successive) hashing functions** used by the scheme during search.
- Each bucket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





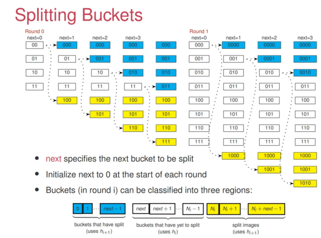
Dynamicity of Linear Hashing

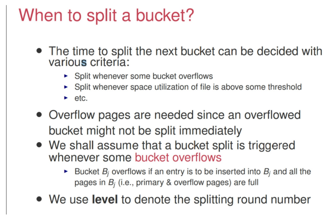




Linear (Hashing) Splitting of Buckets (Insertion)

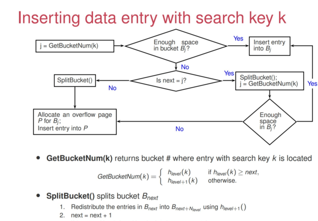
- Number in buckets represent the rightmost bit values.
- **Split Criteria**: variable, could be when some bucket overflow, space utilization of file above some threshold etc.
- **Level**: We use level to denote splitting round number (use with **next**).

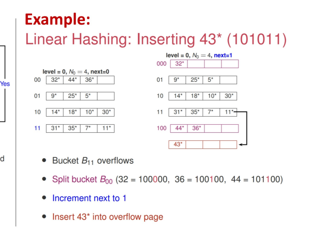


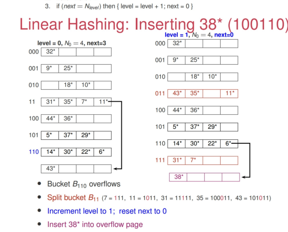


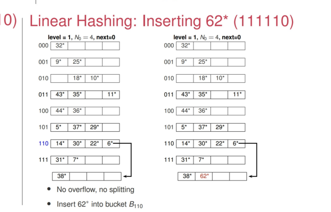
Examples: Linear Hashing Search & Insert

- Even though overflowed bucket split, not necessary 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.



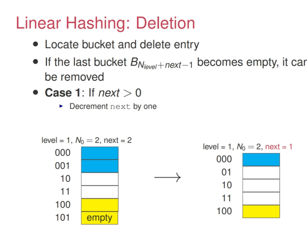


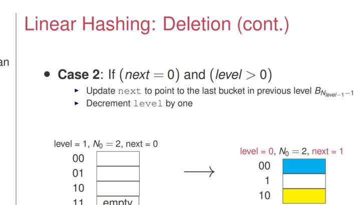




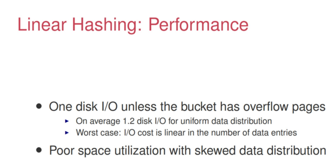
Linear Hashing Deletion

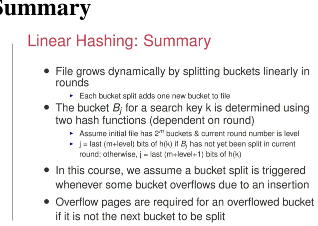
- Opposite of insertion.
- **Two cases**: 1. (Next ¿ 0), 2. (Next = 0, and level > 0.)





Linear Hashing Performance, Summary





Dynamic Hashing: Extendible Hashing

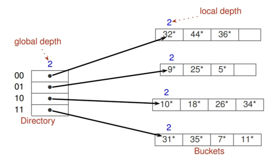
- **Similar to Linear Hashing**: we want **bucket number to grow dynamically**, and use some **number of least significant bits** of h(k) to determine bucket address for search key k.
- **Difference**: Add new bucket (as split image) when existing bucket overflows, No overflow pages (except when number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

Extendible Hashing

- **Extendible hashing**: dynamically updatable 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

- Uses a directory of pointers to buckets
  - Directory expands dynamically as buckets overflow
- Directory has 2^d entries
  - d is called global depth of the hashed file
  - Each directory entry has a unique d-bit address $b_d b_{d-1} \cdots b_2 b_1$
  - Two directory entries are said to correspond if their addresses differ only in the d^{th} bit (i.e., b_d); such entries are called corresponding entries
  - Each bucket maintains a local depth (denoted by $\ell \in [0, d]$)
    - All entries in a bucket with local depth ℓ have the same last ℓ bits in h(.)



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

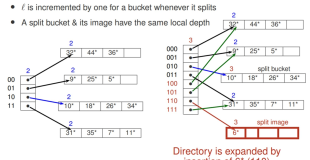
Extendible Hashing

Handling Bucket Overflow (Case 1)

- When a bucket overflows, it is split
  - Allocate a new bucket called its split image
  - Redistribute entries (including new entry) between split bucket & its split image
- **Case 1**: Split bucket's local depth is equal to global depth
- When the directory is doubled,
  - Each new directory entry (except for the entry for the split image) points to the same bucket as its corresponding entry
- Number of directory entries pointing to a bucket = $2^{d-\ell}$

Example: 6* (110)

- Initially, all buckets have local depth $\ell = d$
- A bucket split causes directory to double only if bucket's $\ell = d$
  - d is incremented by one whenever the directory is doubled
- ℓ is incremented by one for a bucket whenever it splits
- A split bucket & its image have the same local depth

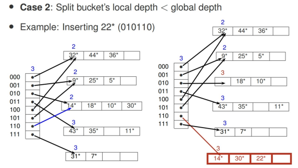


Case 2: Split bucket local depth < global depth.

Extendible Hashing

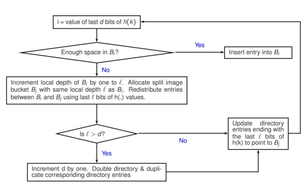
Handling Bucket Overflow (Case 2)

- **Case 2**: Split bucket's local depth < global depth
- Example: Inserting 22* (010110)



Both cases:

Inserting data entry with search key k



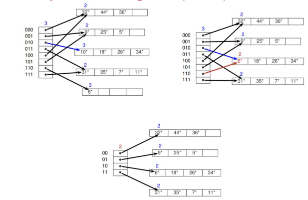
Extendible Hashing Deletion

- 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.

Extendible Hashing: Deletion

- Locate bucket B_i containing entry & delete entry
- If B_i becomes empty, B_i can be merged with the bucket B_j where both buckets have the same local depth ℓ and i & j differs only in the ℓ^{th} bit
  - B_i is deallocated
  - B_j's local depth is decremented by one
  - Directory entries that point to B_i are updated to point to B_j
- More generally, B_i & B_j (with same local depth ℓ and i & j differs only in ℓ^{th} bit) can be merged if their entries can fit within a bucket
- If each pair of corresponding entries point to the same bucket, directory can be halved
  - d is decremented by one

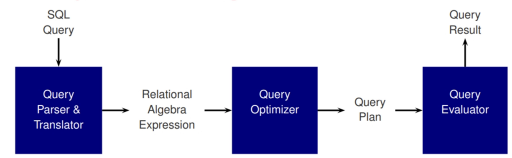
Example: Deleting 10* (1010)



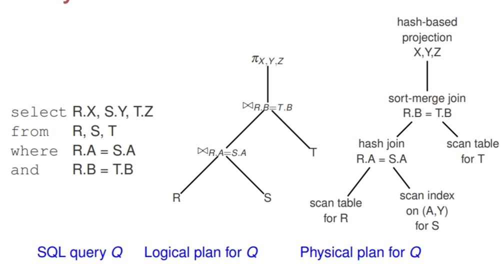
4. Query Evaluation: Sorting & Selection

DBMS describes data it manages using tables, indexes (metadata), which is stored in special tables (system catalogs). This data used to find best way to evaluate a query.

Query Processing



Query Plans



- SQL queries translate into extended form of relational algebra
- Query evaluation plans represented as tree of relational operators, with labels identifying algorithm to use at each node.
- R. operators building blocks for evaluating queries, implementation of operators optimized for good perf.

External Sorting

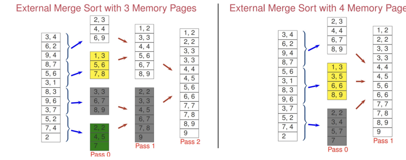
Sorting collection of records on some (search) key is useful and required in variety of situations, including

- Some sorted table of results: `SELECT * FROM student ORDER BY age`.
- Bulk loading a B^+-tree index
- Implementation of other relational algebra operators (e.g. projection, join), which require some sorting step.

When data to be sorted too large to fit into available main memory. Need some **external sorting algorithm**. Algos seek to minimize cost of disk accesses.

External Merge Sort

- **Main Idea**: Pass 0: Creating initial sorted runs (each of X memory pages), then continue during merging passes till you get final sorted pass.
- Sort entire fire by breaking into smaller subfiles, sorting subfiles and merging using minimal amount of main memory at given time.
- **Each sorted subfile is referred to as a run**.
- Sorting 11-page data R using 3 vs 4 memory pages:



Number of Passes of External Merge Sort, $\lceil \log_{B-1}(\lceil N/B \rceil) \rceil + 1$

Num. of data pages, N	Num. of buffer pages, B: 3	5	9	17	129	257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

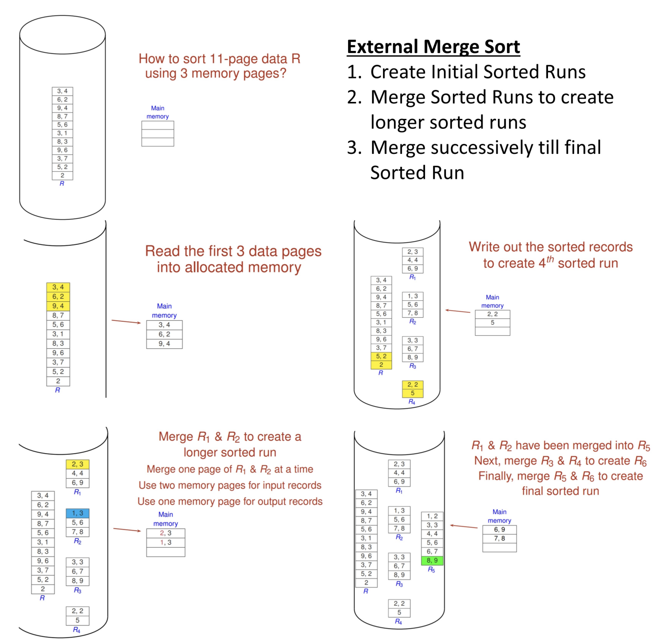
External Merge Sort Analysis

- **Note**: We consider only I/O costs, which approx by counting no. of pages read/written as per cost model. (Simple cost model to convey main idea).

External Merge Sort

- File size of N pages
- Uses B number of buffer pages
- Pass 0: Creation of sorted runs
  - Read in and sort B pages at a time
  - Number of sorted runs created = $\lceil N/B \rceil$
  - Size of each sorted run = B pages (except possibly for last run)
- Pass i, $i \geq 1$: Merging of sorted runs
  - Use $B - 1$ buffer pages for input & one buffer page for output
  - Performs (B-1)-way merge
- Analysis:
  - N_0 = number of sorted runs created in pass 0 = $\lceil N/B \rceil$
  - Total number of passes = $\lceil \log_{B-1}(N_0) \rceil + 1$
  - Total number of I/O = $2N(\lceil \log_{B-1}(N_0) \rceil + 1)$
    - Each pass reads N pages & writes N pages

External Merge Sort Steps



External Sort Optimization: I/O Cost vs. No. of I/Os

- **Cost Metric**: No. of page I/Os.
- Only an approx of true I/O costs, ignores effect of **block(ed)** I/O, **single request to read/write several consecutive pages can be cheaper** than read/write same number of pages through independent I/O requests.
- **Others**: Consider CPU costs as well, can use *double buffering* to keep CPU busy while I/O op. in progress.

External Sort, Block(ed) Page I/O Optimization

Non-Block(ed) Page I/O

- **Consider only No. of page I/O as metric**: Minimize no. of passes in sorting, as each page in file read and written in each pass.
- This means we maximise fan-in **during merging** (aka, how many runs merged per pass), allocate just one buffer pool page per run, and one buffer page for output of merge.
- Hence, (B-1)-way merge per run, minimizing number of passes in sorting algorithm.

Block(ed) Page I/O

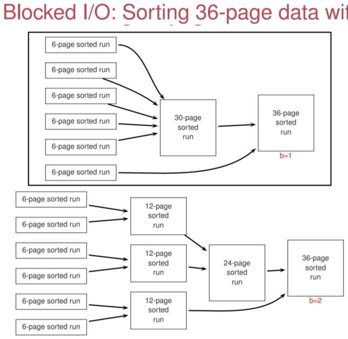
- **Optimization**: Access in blocks may reduce average cost to read/write single page, consideration to read/write in units of more than one page, or R/W in units of a **buffer block**.
- **Buffer block**, of b **pages**. B **is total number of buffer pages** for sorting.

Optimization with Blocked I/O

- Read and write in units of buffer blocks of b pages
- Given an allocation of B buffer pages for sorting,
  - Allocate one block (b pages) for output
  - Remainder space can accommodate $\lfloor \frac{B-b}{b} \rfloor$ blocks for input
  - Thus, can merge at most $\lfloor \frac{B-b}{b} \rfloor$ sorted runs in each merge pass
- Analysis:
  - N = number of pages in file to be sorted
  - B = number of available buffer pages
  - b = number of pages of each buffer block
  - N_0 = number of initial sorted runs = $\lceil N/B \rceil$
  - F = number of runs that can be merged at each merge pass $= \lfloor \frac{B}{b} \rfloor - 1$
  - Number of passes = $\lceil \log_F(N_0) \rceil + 1$

- Set aside one buffer block for output of merge. $(B-b)$. One buffer block per input run ($\frac{B-b}{b}$).
- **Means can merge** at most floor($\frac{B-b}{b}$) sorted runs in each merge pass.

Blocked I/O: Sorting 36-page data with B=6



Non-Block(ed) I/O
b = pages per buffer block = 1.

- Minimize number of Passes.
- Minimize number of individual page I/O.

Block(ed) I/O
b = pages per buffer block = 2.

- Increases number of passes.
- May lower average cost of R/W per individual I/O
- Increase indiv. No. of I/O, but decrease total cost of I/O.

- **Overall**:
  - Larger buffer blocks, (lower average page I/O cost).
  - But num. passes increase, num page I/O increase.
- **Decrease per-page I/O cost tradeoffs Increase No. of page I/Os.**

Sorting using B^+-trees

- When table to be sorted has existing B^+-tree index on sorting attributes.
  - Format 1: Sequentially scan leaf pages of B^+-tree.
  - Format 2/3: Sequentially scan leaf pages of B^+-tree. For each leaf page visited, retrieve data records using RIDs.

Selection $\sigma_p(R)$

Select rows from Relation R that satisfy selection predicate p

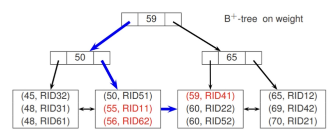
- **Index Matching**: Index **matches** selection predicate if index can be used to evaluate it. Consider Hash index, and B+ Tree index.

Access Path

- **Access path** refers to the given way of accessing data records/entries.
  - **Table scan**: scan all data pages.
  - **Index scan**: scan all index pages.
  - **Index Intersection**: Combine results from multiple index scans (e.g. intersect, union).
- Index scan/intersect follow by **RID lookups** to retrieve data records.

Selectivity of Access Path

- **Selectivity of Access Path:** Number of index & data pages retrieved to access data record/entries
- **Most Selective Access Path:** (Smallest selectivity), retrieves fewest pages.



Q_1: **select** weight **from** R **where** weight **between** 51 **and** 59
Q_2: **select** * **from** R **where** weight **between** 51 **and** 59

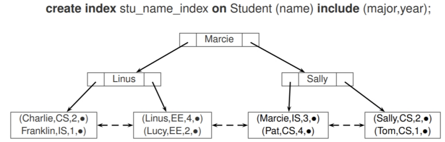
- **Most Selective Access Path minimizes cost of data retrieval.**

Covering Index

- An index I is a covering index for a query Q if all the attributes referenced in Q are part of the key or include column(s) of I
  - Q can be evaluated using I without any RID lookup
  - Such an evaluation plan is known as index-only plan
- **Example**:
  - Consider query Q:
    select major **from** Student **where** name = 'Bob'
  - An index on (name, major) is a covering index for Q
  - An index on (name) is not a covering index for Q
  - An index on (name) with include columns (address,major) is a covering index for Q

B^+-Tree: Include Columns (In Index)

- Relation: Student(sid,name,major,year,address).
- B^+-tree index on Student:
  create index stu_name_index **on** Student (name) **include** (major,year);

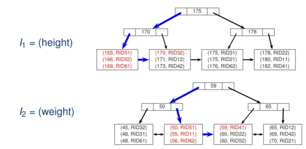


Query: **select** major **from** Student **where** name = 'Lucy';

B^+-Tree: Selection Evaluation

B^+-tree : Index Intersection

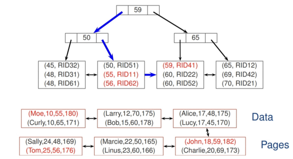
select height, weight **from** Student
where height **between** 164 **and** 170
and weight **between** 50 **and** 59



B^+-tree : Index Scan + RID Lookups

select name
from Student
where weight **between** 55 **and** 59

select weight
from Student
where weight **between** 55 **and** 59
and age $\geq$ 20



CNF Predicates

- **Selectivity of an access path** depends on primary conjuncts in selection condition (w.r.t. index involved.)
- Each conjunct acts as filter on table, fraction of tuples satisfying given conjuct called reduction factor.
  - A term is of the form $R.A$ op c or $R.A_i$ op $R.A_j$
  - A conjunct consists of one or more terms connected by $\vee$
  - A conjunct that contains $\vee$ is said to be disjunctive (or contains a disjunction)
  - A conjunctive normal form (CNF) predicate consists of one or more conjuncts connected by $\wedge$

$$\overbrace{(\underbrace{\text{rating} \geq 8}_{\text{term/conjunct}} \vee \underbrace{\text{director} = \text{“Coen”}}_{\text{term/conjunct}})}^{\text{disjunctive conjunct}} \wedge \underbrace{(\text{year} > 2003)}_{\text{term/conjunct}} \wedge \underbrace{(\text{language} = \text{“English”})}_{\text{term/conjunct}}$$

- **Non-equality Comparison Operators**: $<, \leq, >, \geq, <>$, between, in.

Tree Index matching CNF Selection

- Determines if Index is useful / appropriate, if index can be used to retrieve just the tuples that satisfy the condition.

B^+-tree : Matching predicates

- B^+-tree index $I = (K_1, K_2, \cdots, K_n)$
- Non-disjunctive CNF predicate p
- I matches p if p is of the form:

$$\underbrace{(K_1 = c_1) \wedge \cdots \wedge (K_{i-1} = c_{i-1})}_{\text{zero or more equality predicates}} \wedge (K_i\ op_i\ c_i),\ i \in [1, n]$$

where
1. $(K_1, \cdots, K_i)$ is a prefix of the key of I, and
2. there is at most one non-equality comparison operator which must be on the last attribute of the prefix (i.e., K_i)

- Note: this definition is stronger than R&G's definition

B^+-tree : Matching predicates (cont.)

Example: Which predicates does $I = (age, weight, height)$ match?
Order Matters in search key

1. age $\geq$ 20 Yes, match
2. weight = 80 No
3. (age = 20) $\wedge$ (weight = 70) Yes, match
4. (age = 20) $\wedge$ (weight < 70) Yes, match
5. (age > 20) $\wedge$ (weight = 70) No
6. (age = 20) $\wedge$ (height = 170) Yes, match
7. (height > 180) $\wedge$ (weight = 65) $\wedge$ (age = 20) Yes, match
8. (age = 20) $\wedge$ (weight $\leq$ 65) $\wedge$ (height = 180) No

Hash Index matching CNF Selection

Hash Index: Matching predicates

Determines if entry lies in the bucket.

- Hash index $I = (K_1, K_2, \cdots, K_n)$
- Non-disjunctive CNF predicate p
- I matches p if p is of the form:

$$(K_1 = c_1) \wedge (K_2 = c_2) \wedge \cdots \wedge (K_n = c_n)$$

Hash Index: Matching predicates (cont.)

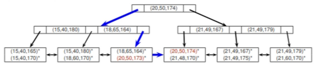
Example: Which predicates does $I = (age, weight, height)$ match?

1. age $\geq$ 20
2. weight = 80
3. (age = 20) $\wedge$ (weight = 70)
4. (age = 20) $\wedge$ (weight < 70)
5. (age > 20) $\wedge$ (weight = 70)
6. (age = 20) $\wedge$ (height = 170)
7. (height = 180) $\wedge$ (weight = 65) $\wedge$ (age = 20)
8. (age = 20) $\wedge$ (weight $\leq$ 65) $\wedge$ (height = 180)

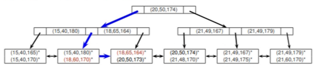
Examples of Index matching CNF Selection

Example: B^+-tree on (age,weight,height)

p_1: (age = 20)



p_2: (age = 18) $\wedge$ (weight $\geq$ 50)



EXAMPLE

- A relation R(a,b,c,d)
- Does the index match the predicate?

Predicate	B+ tree on (a,b,c)	Hash index on (a,b,c)
a=5 and b=3	yes	no
a>5 and b<4	yes	no
b=3	no	no
a=5 and c>10	no	no
a=5 and b=3 and c=1	yes	yes
a=5 and b=3 and c=1 and d >6	yes	yes

a=5 and b=3 and c=1 are primary conjuncts here

CS 564 [Fall 2015] - Paris Koutris

Primary and Covered Conjuncts

- **Primary Conjuncts**: Subset of conjuncts in selection predicate p that index I **matches**.
- In general, only subset of conjuncts of predicate matches index.
- **Covered Conjunct**: Conjunct C in predicate p covered if all attributes in C appear in the key, or *include column(s)* of index I.
- Primary conjuncts subset of covered conjuncts.

Cost of Evaluation of Selection Predicate p

Notation	Meaning
r	relational algebra expression
$\|\|r\|\|$	number of tuples in output of r
$\|r\|$	number of pages in output of r
b_d	number of data records that can fit on a page
b_i	number of data entries that can fit on a page
F	average fanout of B^+-tree index (i.e., number of pointers to child nodes)
h	height of B^+-tree index (i.e., number of levels of internal nodes) $h = \lceil \log_F(\lceil \frac{\|\|R\|\|}{b_i} \rceil) \rceil$ if format-2 index on table R
B	number of available buffer pages

Cost of B^+-tree Index Evaluation of p

Let p' = primary conjuncts of p, p_c = covered conjuncts of p

1. Navigate internal nodes to locate first leaf page

$$Cost_{internal} = \begin{cases} \lceil \log_F(\lceil \frac{||R||}{b_d} \rceil) \rceil & \text{if } I \text{ is a format-1 index,} \\ \lceil \log_F(\lceil \frac{||R||}{b_i} \rceil) \rceil & \text{otherwise.} \end{cases}$$

2. Scan leaf pages to access all qualifying data entries

$$Cost_{leaf} = \begin{cases} \lceil \frac{||\sigma_{p'}(R)||}{b_d} \rceil & \text{if } I \text{ is a format-1 index,} \\ \lceil \frac{||\sigma_{p'}(R)||}{b_i} \rceil & \text{otherwise.} \end{cases}$$

3. Retrieve qualified data records via RID lookups

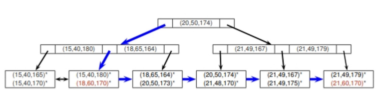
$$Cost_{rid} = \begin{cases} 0 & \text{if I is covering or format-1 index,} \\ ||\sigma_{p_c}(R)|| & \text{otherwise.} \end{cases}$$

Cost of RID lookups could be reduced by first sorting the RIDs

$$\lceil \frac{||\sigma_{p_c}(R)||}{b_d} \rceil \leq Cost_{rid} \leq \min\{||\sigma_{p_c}(R)||, |R|\}$$

Example

- B^+-tree index I = (age,weight,height), Format 2
- Query: **select** * **from** R **where** p
  - p: (age $\geq$ 18) $\wedge$ (weight = 60) $\wedge$ (level = 3)
  - p' = (age $\geq$ 18)
  - p_c = (age $\geq$ 18) $\wedge$ (weight = 60)
- $||R|| = 12$, $||\sigma_{p'}(R)|| = 9$, $||\sigma_{p_c}(R)|| = 2$
- $b_d = b_i = 2$, Height of $I = 2$
- Evaluation cost of p using $I = 2 + \lceil \frac{9}{2} \rceil + 2 = 9$



Cost of Hash Index Evaluation of p

- Let p' = primary conjuncts of p
- For format-1 index
  - Cost to retrieve data records: at least $\lceil \frac{||\sigma_{p'}(R)||}{b_d} \rceil$
- For format-2 index
  - Cost to retrieve data entries: at least $\lceil \frac{||\sigma_{p'}(R)||}{b_i} \rceil$
  -

$$\text{Cost to retrieve data records} = \begin{cases} 0 & \text{if I is a covering index,} \\ \cancel{||\sigma_p(R)||}\ ||\sigma_{p_c}(R)|| & \text{otherwise.} \end{cases}$$

Evaluating Non-Disjunctive / Disjunctive Conjuncts

Evaluating Non-Disjunctive Conjuncts

- Consider the query $\sigma_p(R)$, where
  p = (age = 21) $\wedge$ (weight $\geq$ 70) $\wedge$ (height = 180)
- Suppose the available unclustered indexes on R are
  - a hash index H_{age} on (age), and
  - a B^+-tree index T_{weight} on (weight)
- What are the possible strategies to evaluate p?

Evaluating Disjunctive Conjuncts

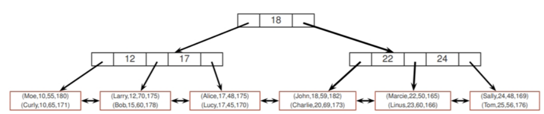
- Suppose the available unclustered indexes are
  - a hash index H_{age} on (age), and
  - a B^+-tree index T_{weight} on (weight)
- What are the possible strategies to evaluate the following predicates?
- p_1 = (age = 21) $\vee$ (height = 180)
- p_2 = ((age = 21) $\vee$ (height = 180)) $\wedge$ (weight $\geq$ 70)
- p_3 = (age = 21) $\vee$ (weight $\geq$ 70)

- **Possible strategies to evaluate Disjunctive / Non-Disjunctive predicates**
- File scan, Use both (fetch RID, take Union), Use B+ tree etc.

Clustered vs. Unclustered Index (B+Tree)

- **Clustered Index**: Order of its data entries is the same or 'close to' order of the data records (in pages).
- Layman Terms: If clustered, if we do a file scan, records will be in order with respect to the attribute.
- An index using Format 1 for data entries is a clustered index.
- Logically, at most one clustered index for each relation.
- **Implication**: Tutorial 3 Q4: When doing index scan with RID lookup, for clustered index, RID page I/O incurred will be the number of leaf pages.
- **Unclustered Index**: Order of data entries not same as actual order of data records. To retrieve each tuple / entry, need to do separate RID lookup / page retrieval.
- **Implication**: Tutorial 3 Q4: When doing index scan with RID lookup, for unclustered index, RID page I/O incurred will be number of pages of tuples (> no. of leaf pages.

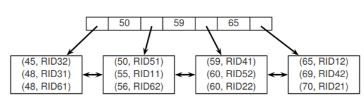
Clustered vs Unclustered Index: Example



Clustered index on R.age

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



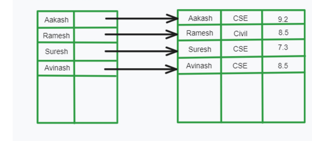
Unclustered index on R.weight

(RIDij = slot j on data page i)

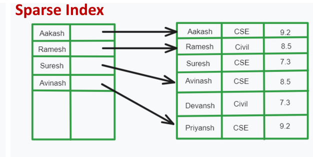
Dense vs. Sparse Index (B+Tree)

- **Dense index**: there is an index record for every search key value in the data.
- The total number of records in the index table and main table are the same.
- Gives quick access to records, effective for range searches as each key value has an entry, but takes more storage, and insertion and deletion higher overhead.
- For *unclustered index*, must be dense.
- **Sparse index**: Some search key value has no have index record. (Main table index points records in specific gap (range of space where index resides in).
- Uses less storage space, lessen effect of insert/delete on index maintenance operations. Time to locate data in index table more, and sparse index records need to be clustered.
- For sparse index, records need to be clustered (in order).

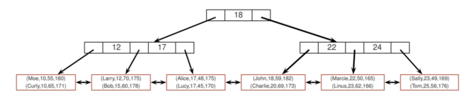
Dense Index



Sparse Index



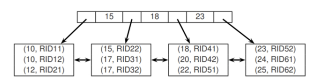
Dense vs Sparse Index: Example



Sparse clustered index on R.age

Relation R

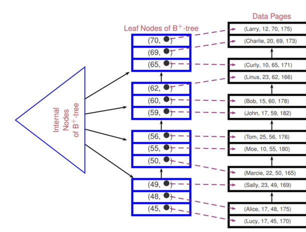
name	age	weight	height
Moe	10	55	180
Curly	10	65	171
Larry	12	70	175
Bob	15	60	178
Alice	17	48	175
Lucy	17	45	170
John	18	59	182
Charlie	20	69	173
Marcie	22	50	165
Linus	23	60	166
Sally	24	48	169
Tom	25	56	176



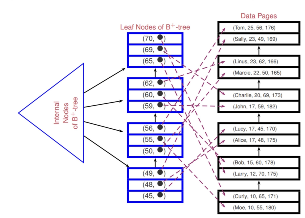
Dense clustered index on R.age

(RIDij = slot j on data page i)

Clustered & Dense B^+-tree on R.weight



Unclustered & Dense B^+-tree on R.weight



PostgreSQL: Buffer Replacement Policy

PostgreSQL: Open source relational database management system.

- **Port Number**: By default, server listens on port number 5432 for client connections.
- PostgreSQL uses variant of Clock policy as default buffer replacement policy.

Overview of PostgreSQL

Shared-memory Data Structures

- As multiple backend server processes may access database at same time, access to shared-memory structures (buffer pool) controlled to ensure consistent access and updates. Use **locks** (spin locks, light-weight locks) to control access.
- **Locking Protocol**: Before accessing shared-memory structure, process acquire lock, upon completion of access, process release lock.

Buffer Manager

- Two main types of Buffers used: Shared Buffer, Local Buffer.
- **Shared Buffer**: Used for holding page from globally accesible relation.
- **Local Buffer**: Usef for holding page from temporary relation locally accessible to specific process.

Management of Shared Buffers

- Initially, all shared buffers maintained in free list.
- **Free list**: Buffer in free list if contents invalid. When new buffer needed, check if buffer available in free list, returned to satisfy buffer request.
- If no available buffer, use **buffer replacement policy** to select victim buffer for eviction to make room for new request. (Use variant of Clock algorithm)
- **Record Deleted/Modified**: Not immediately removed/changed. Multiple versions of record maintained to support **multiversion concurrency control**.
- **Vacuuming Process**: Periodically, vacuuming process runs to remove obsolete versions of records that can be safely deleted from relations. If entire page of records removed, buffer holding page becomes invalid, returned to free list.
- *bgwriter*: background writer process that writes out dirty shared buffers to partly help speed up buffer replacement.

PostgreSQL Buffer Pool

- **Buffer Pool**: Implemented as array of disk blocks, index to each array entry referred to a `buffer_id`, each disk block location identified by buffer tag.
- Pin count for each buffer frame known as reference count `refcount`.
- Each buffer frame associated with a buffer descriptor, stores metadata about contents.
- Given a buffer tag, hash-based buffer table is used to efficiently locate buffer id of buffer frame that stores the disk block (corresponding to given tag) if disk block resident in buffer pool.
- PostgreSQL's buffer manager implementation uses various spin locks and light-weight locks to control access to its shared-memory data structures. For example, the metadata for the Clock replacement policy is stored in a data structure pointed to by a global variable named StrategyControl which consists of a spin lock named `buffer_strategy_lock`. This spin lock is used to control access to various data structures (e.g., free list) by using the functions SpinLockAcquire and SpinLockRelease, respectively, to acquire and release the spin lock `buffer_strategy_lock`.

Implementing LRU Strategy

- Simple approach to implement LRU Policy: **Stack LRU** Method.
- Use doubly Linked List to link up buffer pages. Page closer to the front is more recently used, than page closer to tail of list.
- Whenever buffer page referenced ("Used"), moved to front of list.
- When replacement page sought from list, **unpinned** buffer page closest to tail (LRU) selected for eviction.
- **Whenever buffer accessed, position needs to be adjusted**:

  1. If accessed page in buffer pool already, containing buffer needs to be moved to top of stack.
  2. If accessed page not in buffer pool, **free buffer available** to hold page, selected buffer from free list needs to be inserted onto top of stack.
  3. If accessed page not in buffer pool, **free list empty**, selected victim buffer moved from current stack position to top of the stack.
  4. If buffer in buffer pool **returned to the free list**, buffer removed from stack.