CS3223 Database Systems Implementation

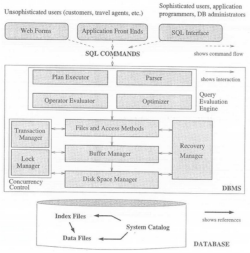
AY23/24 Sem 2, github.com/gerteck

Introduction

Course Details

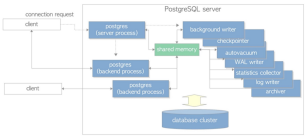
- Prerequisite Knowledge: CS2040S, CS2102, CS2106 background (helpful).
- Reference Textbook: Raghu & Johannes Database M. Systems, 2002. Encouraged to read ahead based on schedule before the lecture.
- Course covers data structures, algorithms, different components making up database systems.

Architecture of DBMS



Source: Ramakrishnan & Gehrke's Figure 1.3

Process Architecture in PostgreSQL



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

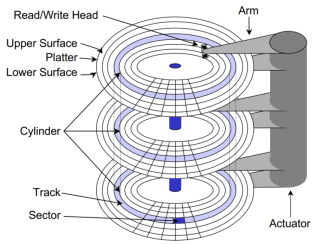
- Relations (Actual tables)
- System catalog (aka data dictionary) storing metadata about relations. (Relation schemas, structure of relations, constraints, triggers. View definitions, Indexes - derived info to speed up access to relations, Statistical information about relations for use by query optimizer.)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

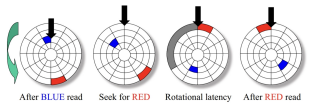
Magnetic Hard-Disk Drive HDD



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).**

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.

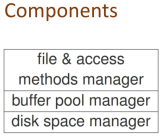
Solid-State Drive: SSD

- 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 ($10^5 - 10^6$)

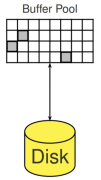
Storage Manager Components

- 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.

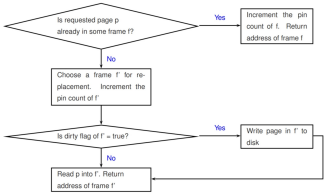
Storage Manager Components



Buffer Manager



BM: Handling request for page p

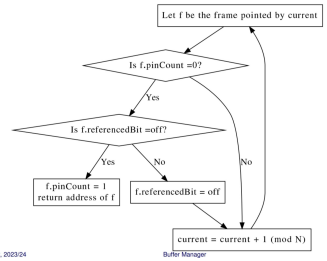


Buffer Manager

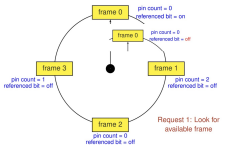
- **Buffer pool**: Main memory allocated for DBMS.
- Buffer pool is partitioned into block-sized pages called **frames**.
- Clients of buffer pool can request for disk page to be fetched into buffer pool, release a disk page in buffer pool.
- A page in the buffer is **dirty** if it has been modified & 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

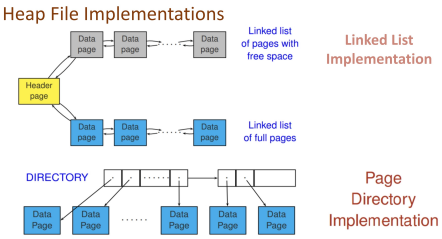
File Abstraction

- 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

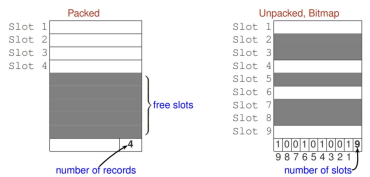


- **Linked list implementation**: Two linked lists, one with pages with free space, other of completely full pages.
- **Page Directory Implementation**: Two leveled implementation. Each big block is a disk block with some metadata. Each disk block has a number of data pages.

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

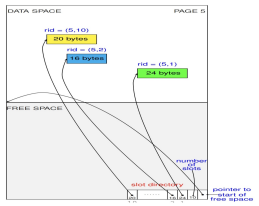
- **RID = (page id, slot number)**
- For **Fixed-Length Records**, Organization can be:
  - **Packed Organization**: Store records in contiguous slots.
  - For packed organization, memory organization is tough and costly when record in slot is deleted, need to move up a record. But as RID serves as a 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:



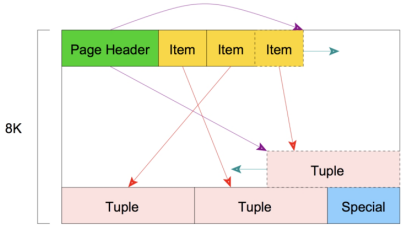
Packed / Unpacked Organization

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

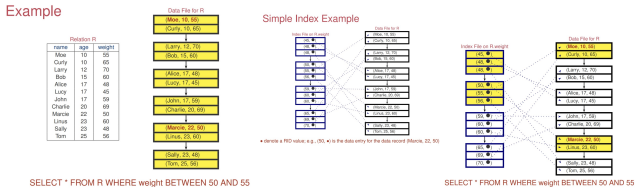
Each o_i is an offset to beginning of field Fi

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

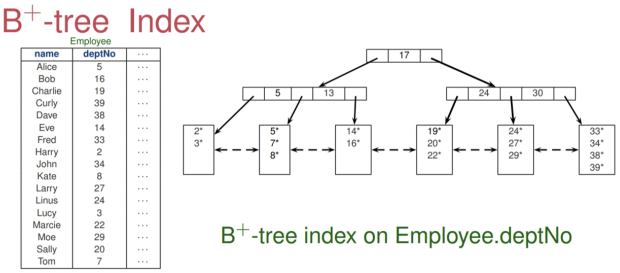
- **Tree-based Index**: Based on sorting of search key values (E.g. ISAM, B^+-tree)
- **Hash-based Index**: Data entries accessed using hashing function (E.g. static/ extendible / linear hashing)
- Considerations when choosing an index:
  - Search Performance
    (Equality search: $k = v$, use hash-based.)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

Tree-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

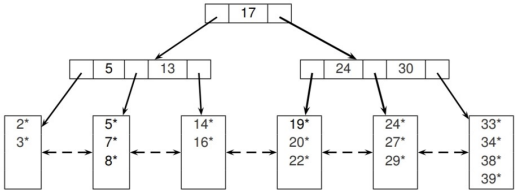


- Each node is either a **leaf node** (bottom-most level) or an internal node.
- Top-most internal node is the **root node** located at **level 0**.
- **Height of Tree** = number of level of internal nodes. (Leaf nodes are at level h where h = height of tree).
- Nodes at same level are **sibling nodes**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



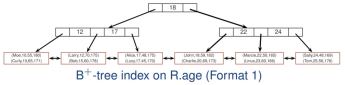
Formats of Data Entries in B-Tree

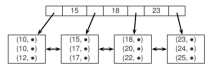
- **Format 1**: k* is actual *data record* (with search key value k)
- **Format 2**: k* is of form *(k, rid)*, where rid is record identifier of record with search key value k.
- **Format 3**: k* is of form *(k, rid-list)*, where rid-list is list of record identifiers of data records with search key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

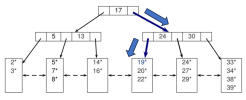




B^+-tree Search Algorithms

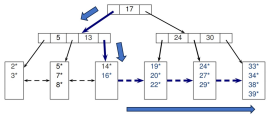
- Search algorithm finds the leaf node a given data entry belongs to.
- We assume no duplicates, no data entries same key value. Note in practice, duplicates arise whenever search key does not contain candidate key, must be dealt with.

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 ($k \geq 15$)



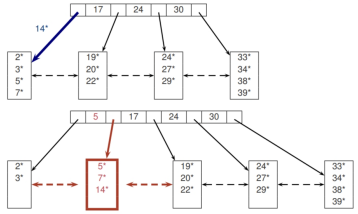
B^+-Tree Insertion

- **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

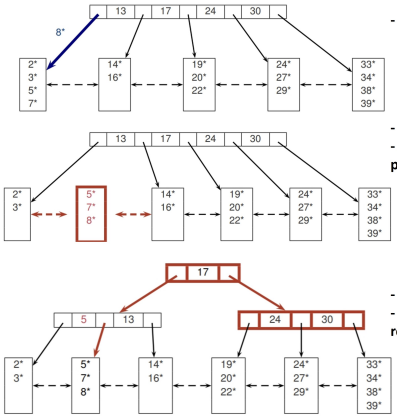
- Split overflowed leaf node by distributing $d + 1$ entries to new leaf node.
- Create a new entry index using smallest key in leaf node.
- Insert new index entry into parent node of overflowed node.

Inserting 14* (Splitting of overflowed node)

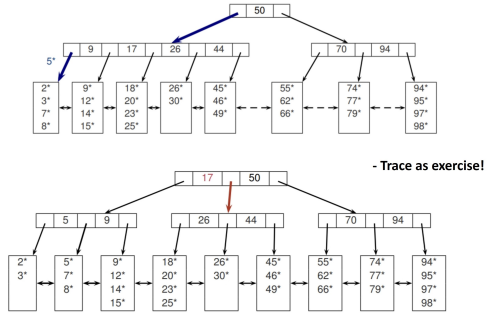


- Sometimes, **node split is propagated upwards** to ancestor internal nodes.
- When splitting an **internal node**, the middle key is pushed to parent node.

Inserting 8* (Propagation of node splits)



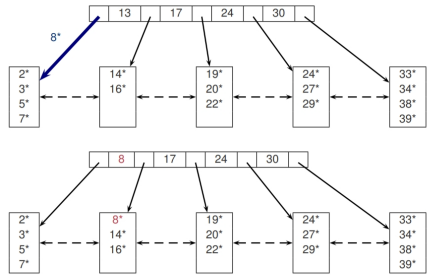
Inserting 5* (Propagation of node splits)



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.

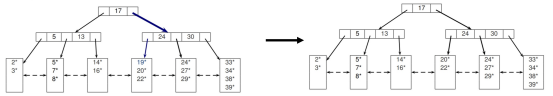
Inserting 8* (Redistribution of data entries)



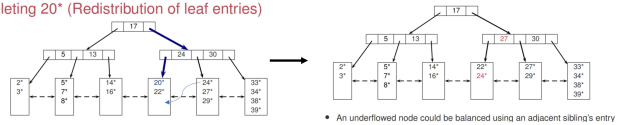
B^+-Tree Deletion

- **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)



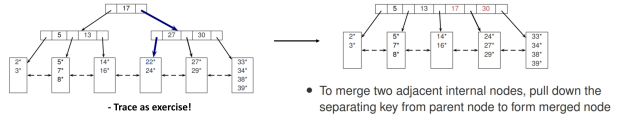
Deleting 20* (Redistribution of leaf entries)



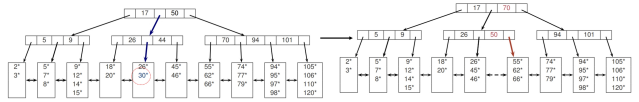
Deleting 20* (Merging of nodes)

- 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 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).

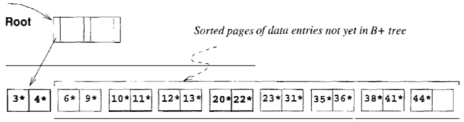


Figure 10.22 Initial Step in B+ Tree Bulk-Loading

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.

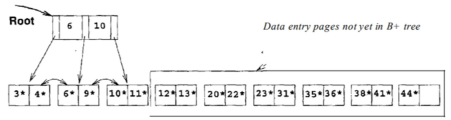


Figure 10.23 Root Page Fills up in B+ Tree Bulk-Loading

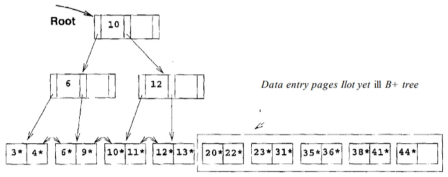


Figure 10.24 Page Split during B+ Tree Bulk-Loading

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.

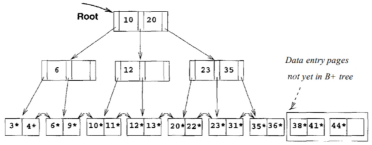
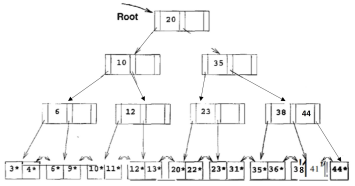


Figure 10.25 Before Adding Entry for Leaf Page Containing 38*



3. Hash-based Indexing

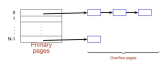
- Used for **equality queries**, not for range queries.
- **Hashing techniques**: **Static hashing, dynamic hashing** (linear hashing, extendible hashing, etc).

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.

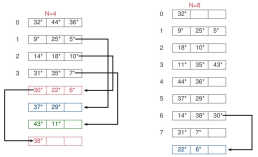
Static Hashing

- 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.)

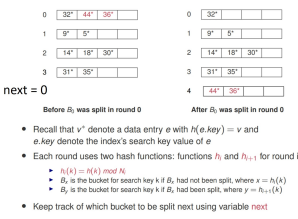
- Assume initial file size of N_0 buckets
  - Buckets $B_0, B_1, \cdots B_{N_0-1}$
- File grows linearly by splitting buckets in rounds
  - At each round, buckets are split sequentially: B_i is split before B_{i+1}
- How to split a bucket B_i?
  - Add a new bucket B_j (known as split image of B_i)
  - 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_i is B_{8+i}, for $i \in [0,7]$

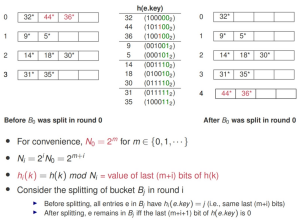
Dynamicity of Linear Hashing

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

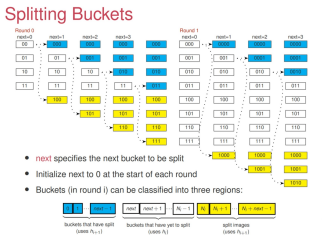


- For convenience, $N_0 = 2^m$ for $m \in \{0, 1, \cdots\}$
- $N_i = 2^i N_0 = 2^{m+i}$
- $h_i(k) = h(k) \bmod N_i$ = value of last (m+i) bits of h(k)
- Consider the splitting of bucket B_j in round i
  - Before splitting, all entries e in B_j have $h_i(e.key) = j$ (i.e., same last (m+i) bits)
  - After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



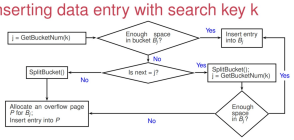
When to split a bucket?

- 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 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.

Inserting data entry with search key k



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

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

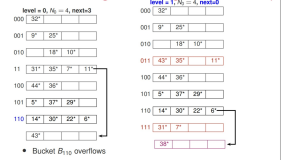
- **SplitBucket()** splits bucket B_{next}
  1. Redistribute the entries in B_{next} into B_{next} & $B_{next+N_{level}}$ using $h_{level+1}()$
  2. next = next + 1
  3. If (next = N_{level}) then { level = level + 1; next = 0 }

Example:

Linear Hashing: Inserting 43* (101011)

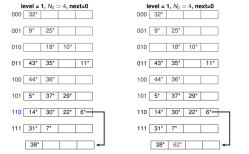
- Bucket B_{11} overflows
- Split bucket B_{00} (32 = 100000, 36 = 100100, 44 = 101100)
- Increment next to 1
- Insert 43* into overflow page

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)



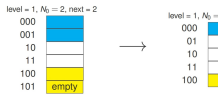
- No overflow, no splitting
- Insert 62* into bucket B_{110}

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

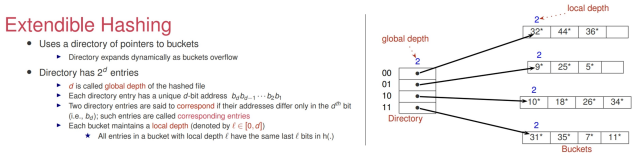
- File grows dynamically by splitting buckets linearly in rounds
  - Each bucket split adds one new bucket to file
- The bucket B_j for a search key k is determined using two hash functions (dependent on round)
  - Assume initial file has 2^m buckets & current round number is level
  - j = last (m+level) 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 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**.



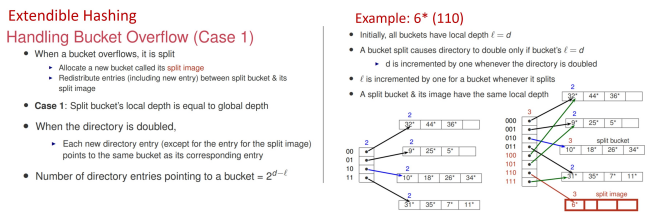
Extendible Hashing Performance

- **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.

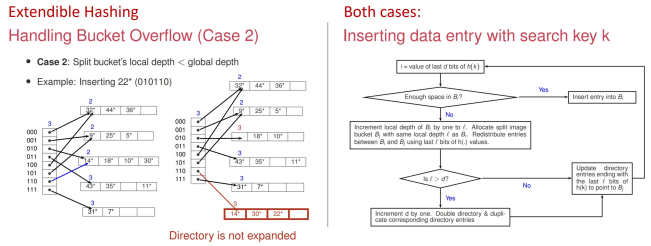
Extendible Hashing: Handling Bucket Overflow

- 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

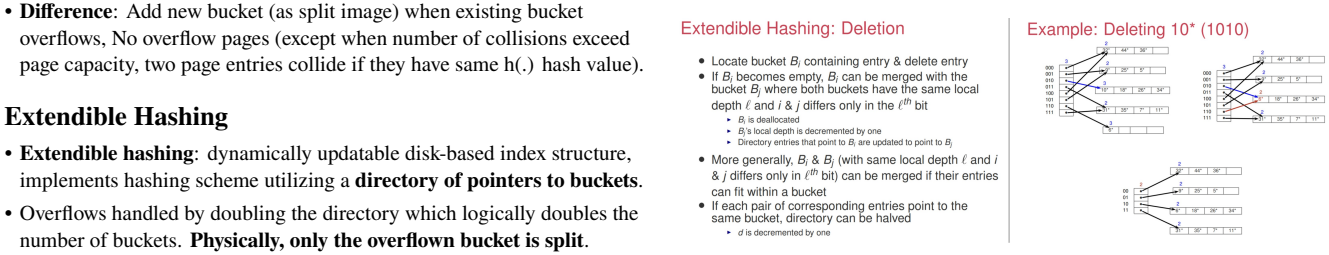


Case 2: Split bucket local depth < global depth.



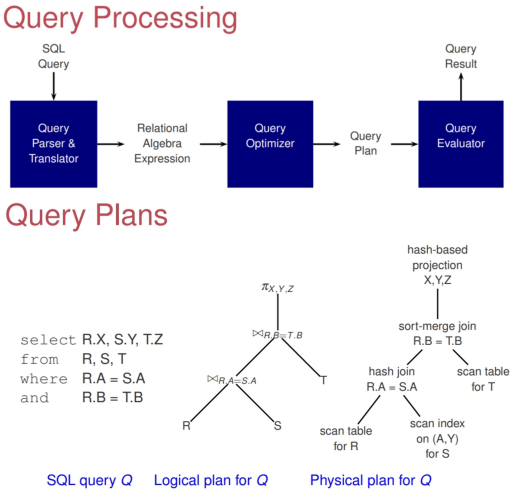
Extendible Hashing Deletion

- 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.



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.



- 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

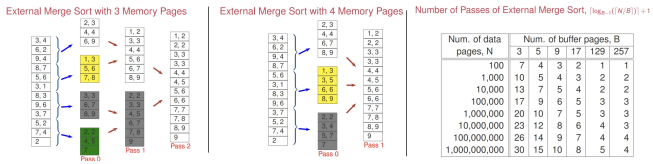
Sorting collection of records on some (search) key is useful and required in variety of situations, including

- 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

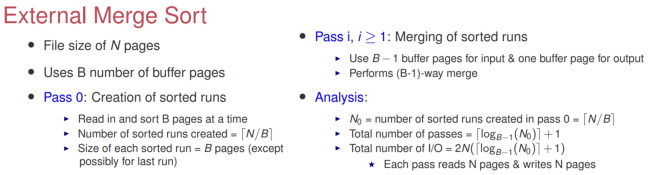
- **Main Idea**: Pass 0: Creating initial sorted runs (each of X memory pages), then continue during merging passes till you get final sorted pass.
- Sort entire fire by breaking into smaller subfiles, sorting subfiles and merging using minimal amount of main memory at given time.
- **Each sorted subfile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



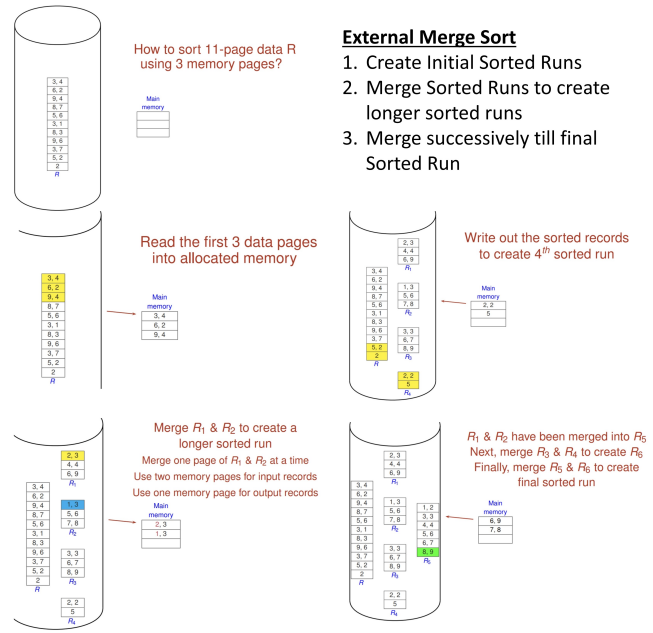
Num. of data pages, N	Num. of buffer pages, B					
	3	5	9	17	129	257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

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 Steps



External Merge Sort

1. Create Initial Sorted Runs
2. Merge Sorted Runs to create longer sorted runs
3. Merge successively till final Sorted Run

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

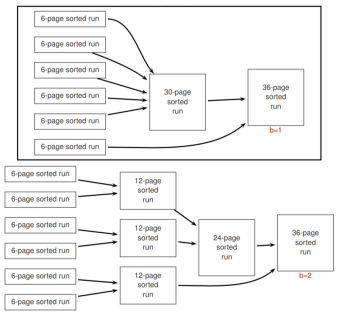
- **Optimization**: Access in blocks may reduce average cost to read/write single page, consideration to read/write in units of more than one page, or R/W in units of a **buffer block**.
- **Buffer block**, of b **pages**. B **is total number of buffer pages** for sorting.

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

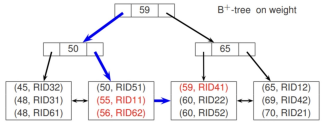
- **Index Matching**: Index **matches** selection predicate if index can be used to evaluate it. Consider Hash index, and B+ Tree index.

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

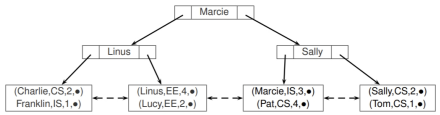
- **Most Selective Access Path minimizes cost of data retrieval.**

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)

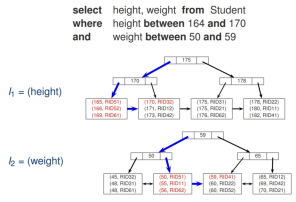
- Relation: Student(sid,name,major,year,address).
- B^+-tree index on Student:
  create index stu_name_index **on** Student (name) **include** (major,year);



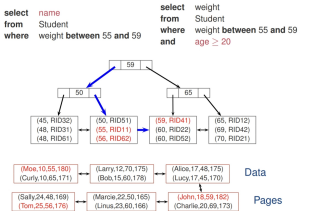
Query: **select** major **from** Student **where** name = 'Lucy';

B^+-Tree: Selection Evaluation

B+-tree : Index Intersection



B+-tree : Index Scan + RID Lookups



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 $\wedge$

disjunctive conjunct

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

- **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 ≥ 20 — Yes, match
2. weight = 80 — No
3. (age = 20) ∧ (weight = 70) — Yes, match
4. (age = 20) ∧ (weight < 70) — Yes, match
5. (age > 20) ∧ (weight = 70) — No
6. (age = 20) ∧ (height = 170) — Yes, match
7. (height > 180) ∧ (weight = 65) ∧ (age = 20) — Yes, match
8. (age = 20) ∧ (weight ≤ 65) ∧ (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:

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

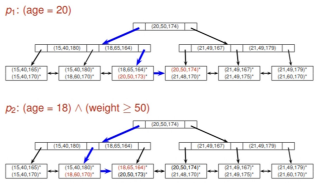
Hash Index: Matching predicates (cont.)

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

1. age ≥ 20
2. weight = 80
3. (age = 20) ∧ (weight = 70)
4. (age = 20) ∧ (weight < 70)
5. (age > 20) ∧ (weight = 70)
6. (age = 20) ∧ (height = 170)
7. (height = 180) ∧ (weight = 65) ∧ (age = 20)
8. (age = 20) ∧ (weight ≤ 65) ∧ (height = 180)

Examples of Index matching CNF Selection

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



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

CS564 (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

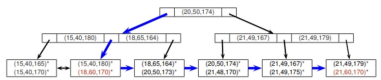
$$Cost_{rid} = \begin{cases} 0 & \text{if I is covering or format-1 index,} \\ \|\|\sigma_{p_c}(R)\|\| & \text{otherwise.} \end{cases}$$

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 ≥ 18) ∧ (weight = 60) ∧ (level = 3)
  - p' = (age ≥ 18)
  - p_c = (age ≥ 18) ∧ (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 $\lceil \frac{\|\|\sigma_{p'}(R)\|\|}{b_i} \rceil$
  -

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

Evaluating Non-Disjunctive / Disjunctive Conjuncts

Evaluating Non-Disjunctive Conjuncts

- Consider the query $\sigma_p(R)$, where p = (age = 21) ∧ (weight ≥ 70) ∧ (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

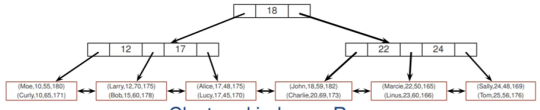
- Suppose the available unclustered indexes are
  - a hash index H_{age} on (age), and
  - a B$^+$-tree index T_{weight} on (weight)
- What are the possible strategies to evaluate the following predicates?
- p_1 = (age = 21) ∨ (height = 180)
- p_2 = ((age = 21) ∨ (height = 180)) ∧ (weight ≥ 70)
- p_3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

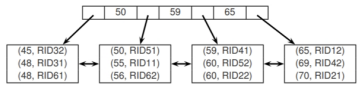
Clustered vs Unclustered Index: Example



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

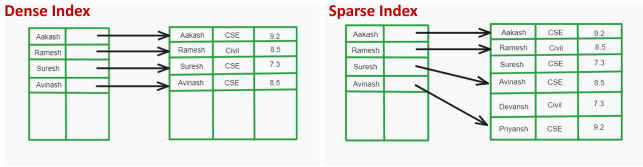


Unclustered index on R.weight

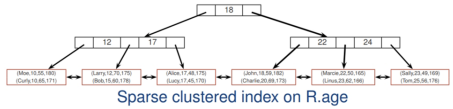
(RIDij = slot j on data page i)

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).



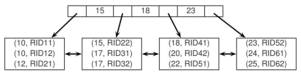
Dense vs Sparse Index: Example



Sparse clustered index on R.age

Relation R

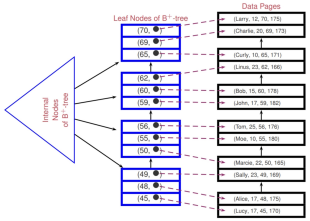
name	age	weight	height
Moe	10	55	180
Curly	10	65	171
Larry	12	70	175
Bob	15	60	178
Alice	17	48	175
Lucy	17	45	170
John	18	59	182
Charlie	20	69	173
Marcie	22	50	165
Linus	23	62	166
Sally	24	49	169
Tom	25	56	176



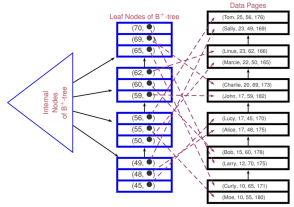
Dense clustered index on R.age

(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.

Implementing LRU Buffer Replacement Policy

- 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.

Enhanced LRU (ELRU) Buffer Replacement Policy

- A disadvantage of LRU is that a page that is accessed only once could evict a frequently accessed page from the buffer.
- To address this limitation, ELRU, which is a variant of LRU, keeps track of additional page access information to manage the eviction of buffer pages in two separate groups.
- In contrast to LRU which uses only the last access time of buffer pages, ELRU maintains the last two access times of buffer pages.

Specifications of ELRU

- Specifically, let $B = B_1 \cup B_2$ denote the set of unpinned buffer pages that can be selected for replacement.
- B_1 is the set of buffer pages that have been accessed only once.
- B_2 is the set of buffer pages that have been accessed at least twice.
- The ELRU policy selects a replacement page from B for replacement as follows. If B_1 is non-empty, ELRU selects the least recently accessed page in B_1 as the replacement page; i.e., ELRU applies the conventional LRU policy to select the replacement page from B_1. Otherwise, if B_1 is empty, ELRU selects the page in B_2 with the smallest second-last access time as the replacement page.

PostgreSQL: Normal Buffer Replacement Strategy (Clock-Sweep)

- **PostgreSQL standard Buffer Replacement Strategy**
- There is a "free list" of buffers that are prime candidates for replacement. In particular, buffers that are completely free (contain no valid page) are always in this list. We could also throw buffers into this list if we consider their pages unlikely to be needed soon; however, the current algorithm never does that. The list is singly-linked using fields in the buffer headers; we maintain head and tail pointers in global variables. (Note: although the list links are in the buffer headers, they are considered to be protected by the buffer_strategy_lock, not the buffer-header spinlocks.)
- To choose a victim buffer to recycle when there are no free buffers available, we use a simple clock-sweep algorithm, which avoids the need to take system-wide locks during common operations. It works like this:
- Each buffer header contains a usage counter, which is incremented (up to a small limit value) whenever the buffer is pinned. (This requires only the buffer header spinlock, which would have to be taken anyway to increment the buffer reference count, so it's nearly free.)
- The "clock hand" is a buffer index, nextVictimBuffer, that moves circularly through all the available buffers. nextVictimBuffer is protected by the buffer_strategy_lock.

Clock-Sweep Buffer Replacement Algorithm

- The **algorithm** for a process that needs to obtain a victim buffer is:

1. Obtain buffer_strategy_lock.
2. If buffer free list is nonempty, remove its head buffer. Release buffer_strategy_lock. If the buffer is pinned or has a nonzero usage count, it cannot be used; ignore it go back to step 1. Otherwise, pin the buffer, and return it.
3. Otherwise, the buffer free list is empty. Select the buffer pointed to by nextVictimBuffer, and circularly advance nextVictimBuffer for next time. Release buffer_strategy_lock.
4. If the selected buffer is pinned or has a nonzero usage count, it cannot be used. Decrement its usage count (if nonzero), reacquire buffer_strategy_lock, and return to step 3 to examine the next buffer.
5. Pin the selected buffer, and return.

- (Note that if the selected buffer is dirty, we will have to write it out before we can recycle it; if someone else pins the buffer meanwhile we will have to give up and try another buffer. This however is not a concern of the basic select-a-victim-buffer algorithm.)

5. Query Evaluation: Projection & Join

Projection: $\pi_{A1,...,Am}(R)$

- $\pi_L(R)$ projects columns given by list L from relation R.
- $\pi^*_L(R)$ same as $\pi_L(R)$ but preserves duplicates.
- Example: `select distinct age from R`

Projection Operation $\pi_{A1,...,Am}(R)$

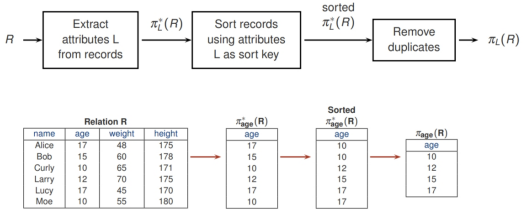
- **Projection involves two tasks**:
  1. Remove unwanted attributes. (from tuples)
  2. Eliminate any duplicate tuples produced.
- **Two approaches to Project**:
  1. Projection based on **sorting**.
  2. Projection based on **hashing**.

Sort-based Approach

Simple Sort-based Approach

- Treat each step as a black box, push tuples through pipeline to sort.

Consider $\pi_L(R)$ where L denote some sequence of attributes of R



Relation R

name	age	weight	height
Alice	17	48	175
Bob	15	60	178
Curly	10	65	171
Larry	12	70	175
Lucy	17	45	170
Moe	10	55	180

$\pi^*_{age}(R)$

age
17
15
10
12
17
17

Sorted $\pi^*_{age}(R)$

age
10
10
12
15
17
17

$\pi_{age}(R)$

age
10
12
15
17

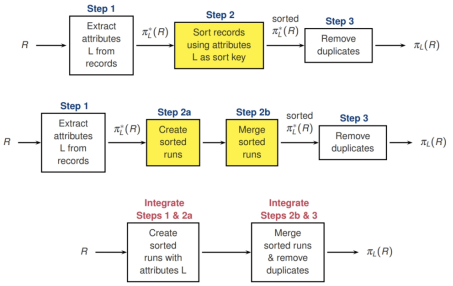
Cost Analysis

- **Step 1**:
  - Cost to scan records = $|R|$
  - Cost to output temporary result = $|\pi^*_L(R)|$
- **Step 2**:
  - Cost to sort records = $2|\pi^*_L(R)|\ (\log_m(N_0)+1)$
  - N_0 = number of initial sorted runs, m = merge factor
- **Step 3**:
  - Cost to scan records = $|\pi^*_L(R)|$

Optimized Sort-based Approach

- By opening black box and examining the sort algorithm (external merge sort), we can optimize the sort-based approach.

Optimized Sort-based Approach



Hash-based Approach

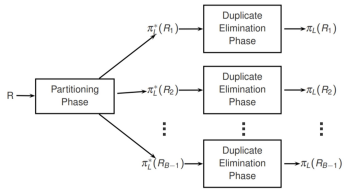
- For , we build a **main-memory hash table** T to detect and remove duplicates.
- Cost = $|R|$ if T fits in main memory.
- **Two Phases**: Partitioning phase and Duplicate Elimination phase.

1. Partitioning Phase	2. Duplicate Elimination Phase
Partition R into R_1, R_2, ..., R_{B-1} • Hash on $\pi_L(t)$ for each tuple t ∈ R • R = R_1 U R_2 U ... U R_{B-1} • $\pi_L^*(R_i) \cap \pi_L^*(R_j) = \emptyset$ • Basically, each hash table does not overlap, and all tables make up R.	Eliminates duplicates for each partition $\pi_L^*(R_i)$. • For each tuple t, hash t into bucket B_j with diff. hash function, insert if not already in. • Output all tuples in all buckets once done. • Partition with * means contain duplicate. • Partition with no * means no duplicate. • $\pi_L(R)$ = duplicate-free union of $\pi_L(R_1)$, ... , $\pi_L(R_{B-1})$

Overview of Hash-based Approach

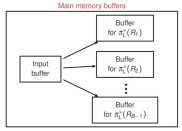
- Build a main-memory hash table T to detect & remove duplicates

```
01. initialize an empty hash table T
02. for each tuple t in R do
03.     apply hash function h on πL(t)
04.     let t be hashed to bucket Bi in T
05.     if (πL(t) is not in Bi) then
06.         insert πL(t) into Bi
07. output all entries in T
```



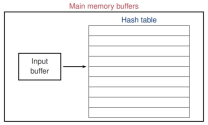
Partitioning Phase

- Use one buffer for input & $(B-1)$ buffers for output
- Read R one page at a time into input buffer
- For each tuple t in input buffer,
  - project out unwanted attributes from t to form t'
  - apply a hash function h on t' to distribute t' into one output buffer
  - flush output buffer to disk whenever buffer is full



Duplicate Elimination Phase

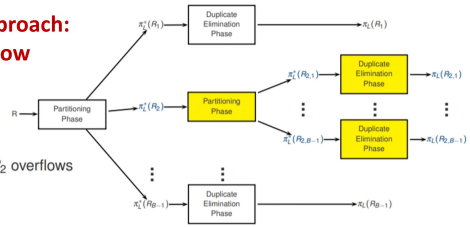
- For each partition R_i,
  - Initialize an in-memory hash table
  - Read $\pi^*_L(R_i)$ one page at a time; for each tuple t read,
    - Hash t into bucket B_j with hash function h' ($h' \neq h$)
    - Insert t into B_j if $t \notin B_j$
  - Output tuples in hash table



Hash-based Approach Partition Overflow

- **Partition Overflow Problem**: When hash table for $\pi^*_L(R)$ (Partitioned table) is larger than available memory buffers.
- Recursively apply hash-based partitioning to the overflowed partition.

Hash-based Approach: Partition Overflow

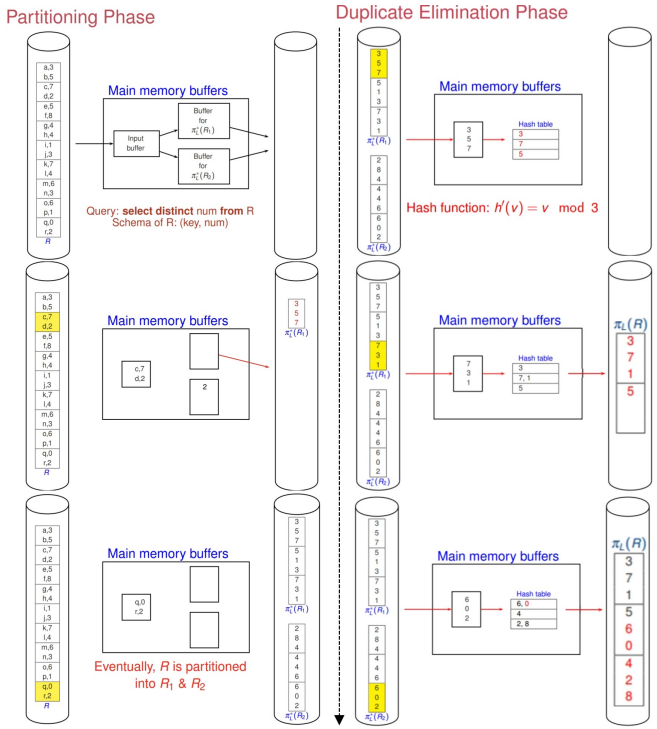


- **Example**: Partition R_2 overflows

Notation

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

Illustration of Partitioning, Duplicate Elimination Phase



Analysis of Hash-based Approach

- Approach is effective if **B is large relative to** $|R|$.

Hash-Based Approach Analysis	
Approach is effective if B (available buffer pages) large relative to |R| (number of pages in r). **Optimal size of B:** • Assume that h distributes tuples in R uniformly • Each R_i has $\frac{\|\pi^*_L(R)\|}{B-1}$ pages • Size of hash table for each R_i = $\frac{\|\pi^*_L(R)\|}{B-1} \times f$ ★ f = *fudge factor* • Therefore, to avoid partition overflow, $B > \frac{\|\pi^*_L(R)\|}{B-1} \times f$ ★ Approximately, $B > \sqrt{f \times \|\pi^*_L(R)\|}$	**Cost assuming no Partition Overflow:** • Partitioning Phase: $R\| + \|\pi^*_L(R)\|$ • Duplicate Elimination Phase: $\|\pi^*_L(R)\|$ • **Total Cost:** $\|R\| + 2\|\pi^*_L(R)\|$

Sort-based vs. Hash-based Analysis

1. Hash-Based	2. Sort-Based
Assume no partition overflow, i.e. $B > \sqrt{f \times \|\pi^*_L(R)\|}$ Then, Cost = $\underbrace{\|R\| + \|\pi^*_L(R)\|}_{\text{partitioning phase}} + \underbrace{\|\pi^*_L(R)\|}_{\text{duplicate elimination phase}}$	Output is sorted. Good if many duplicates, or distribution of hashed values non-uniform. If $B > \sqrt{\|\pi^*_L(R)\|}$, ★ Number of initial sorted runs $N_0 = \lceil \frac{\|R\|}{B} \rceil \approx \sqrt{\|\pi^*_L(R)\|}$ ★ Number of merging passes = $log_{B-1}(N_0) \approx 1$ ★ Sort-based approach requires 2 passes for sorting ★ Cost = $\underbrace{\|R\| + \|\pi^*_L(R)\|}_{\text{pass 0}} + \underbrace{\|\pi^*_L(R)\|}_{\text{pass 1}}$ ★ Both hash-based & sort-based methods have same I/O cost

Projection Operation: using Indexes

- If there is an index whose search key (+ include columns) contains all wanted attributes, **replace table scan with index scan**.
- If index ordered (B+-tree), search key includes wanted attributes as prefix, scan data entries in order, compare adjacent duplicate data entries.
- E.g. B+-tree index on R with key (A, B) to evaluate query $\pi_A(R)$.

Join: $R_{\bowtie_\theta} S$

- Join is where there is match between values of specified columns.
- Two-table joins, multiple-table joins, self-joins etc.
- Example: `SELECT * FROM customer c, orders o WHERE c.name = o.name;`

Join: $R \bowtie_\theta S$

Database Schema:
- Employee (eid, ename, city, did)
- Department (did, dname, city, managerId)

Example 1: Find *(eid, managerId)* pairs where *managerId* is manager of *eid*.

$\pi_{eid,managerId}$ (Employee $\bowtie_\theta$ Department)
where θ: Employee.did = Department.did

Example 2: Find *(eid, did)* pairs where *eid* and *did* co-located in same city.

$\pi_{eid,did}$ (Employee $\bowtie_\theta$ Department)
where θ: Employee.city = Department.city

General Join Conditions

- Multiple equality-join conditions
  - Example: (R.A = S.A) and (R.B = S.B)
  - Algorithms:
    - ⋆ **Index Nested Loop Join**: use index on all or some of join attributes
    - ⋆ **Sort-Merge Join**: need to sort on combination of attributes
    - Other algorithms essentially unchanged
- Inequality-join conditions
  - Example: (R.A < S.A)
  - Algorithms:
    - ⋆ **Index Nested Loop Join**: requires a B⁺-tree index
    - ⋆ **Sort-Merge Join**: not applicable
    - ⋆ **Hash-based Joins**: not applicable
    - Other algorithms essentially unchanged

Join Algorithms

- **Iteration-Based**: Block nested loop.
- **Index-Based**: Index nested loop.
- **Partition-Based**: Sort-merge join, hash join.

Join Algorithms Analysis

- **Factors to Consider**:
- *Type of join predicates* (equality predicates e.g. $R.A_i = S.B_j$), (inequality predicates e.g. $R.A_i < S.B_j$)
- *Size of join operands*
- Available *buffer space*, available *access methods*.
- Given a join $R_{\bowtie_\theta} S$, R is **outer relation**, S is **inner relation**

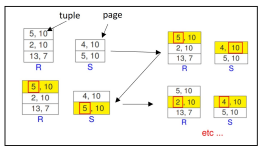
Tuple/Page-based Nested Loop Join

- Simplest join algo is **tuple-at-a-time** nested loop evaluation. We scan outer relation R, and for each tuple in R, scan inner relation S.
- Simple refinement is **join page-at-a-time**. For each page of R, retrieve each page of S, and write out matching tuples.
- Importance of page-orientated operations for minimizing disk I/O.
- **Observation**: Choose outer relation R to be smaller of two relations.

Tuple-based Nested Loop Join

- Tuple-based Algorithm:

```
for each tuple r ∈ R do
    for each tuple s ∈ S do
        if (r matches s) then
            output (r, s) to result
```



Tuple-based join **scans S once for every tuple in R.**

- I/O Cost Analysis:

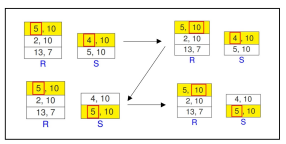
$$\underbrace{|R|}_{\text{scan R}} + \underbrace{|R| \times |S|}_{\text{scan S}}$$

Single line | R | is number of pages in R
Double line || R || is number of tuples in R

Page-based Nested Loop Join

- Page-based Algorithm:

```
for each page P_R of R do
    for each page P_S of S do
        for each tuple r ∈ P_R do
            for each tuple s ∈ P_S do
                if (r matches s) then
                    output (r, s) to result
```



Page-based optimization of tuple-based nested loop join.
- Scan over S once for every page in R.
- As a result, it prevents S from being scanned as many times, thus decreasing the number of disk reads.

- I/O Cost Analysis:

$$\underbrace{|R|}_{\text{scan R}} + \underbrace{|R| \times |S|}_{\text{scan S}}$$

Block Nested Loop Join

- Simple nested loops join algo does not effectively utilize buffer pages.
- If enough memory, read whole of R (smaller relation), use one of extra buffer page to scan larger relation S. Last buffer page used as output buffer. Each relation scanned just once, optimal!
- **Generalization**: Break relation R into *blocks* that can fit into available buffer pages, and scan all of S for each block of R.
- R is **outer relation**, as it is scanned only once. S is **inner relation**, scanned multiple times.

Block Nested Loop Join

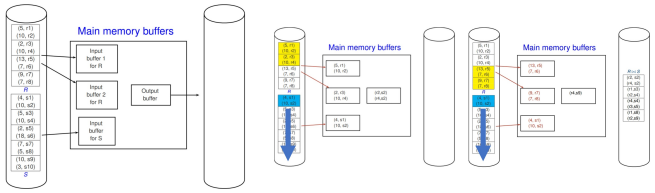
- **Motivation**: How to better exploit buffer space to minimize number of I/Os?
- Assume $|R| \leq |S|$. So choose R as outer & S as inner
- **Buffer space allocation**: Allocate one page for S, one page for output & remaining pages for R

Algorithm (using B buffer pages):

```
while (scan of R is not done) do
    read next (B − 2) pages of R into buffer
    for each page P_S of S do
        read P_S into buffer
        for each tuple r of R in buffer and each tuple s ∈ P_S do
            if (r matches s) then output (r, s) to result
```

I/O Cost: $|R| + (\lceil \frac{|R|}{B-2} \rceil \times |S|)$

Block Nested Loop Join: Example



Index Nested Loop Join

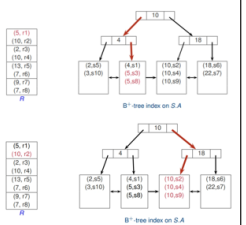
- If exists index on one of the relations on the join attribute(s), take advantage by making **indexed relation to be inner relation (S)**.
- For each tuple in R, use index to retrieve matching tuples of S.
- Cost of scanning R is M, and cost of retrieving matching S tuples depends on kind of index and number of matching tuples.

Index Nested Loop Join

Precondition: There is an index on the join attribute(s) of inner relation S.
Idea: for each tuple ϵ R, use r to probe S's index to find matching tuples.

Analysis:

- Let $R.A_i = S.B_j$ be the join condition
- Uniform distribution assumption:
  each R-tuple joins with $\lceil \frac{||S||}{||\pi_{B_j}(S)||} \rceil$ number of S-tuples
- For a format-1 B⁺-tree index on S,
  - ⋆ I/O Cost = $\underbrace{|R|}_{\text{scan R}} + \underbrace{||R|| \times J}_{\text{join each R-tuple with S}}$
  - ⋆ $J = \underbrace{\log_F(\lceil \frac{||S||}{b_d} \rceil)}_{\text{search index's internal nodes}} + \underbrace{\lceil \frac{||S||}{b_d \, ||\pi_{B_j}(S)||} \rceil}_{\text{search index's leaf nodes}}$



Cost of retrieving matching S tuple for each R tuple:

1. If the index on S is a B+ tree index, the cost to find the appropriate leaf is typically 2--4 I/Os. If the index is a hash index, the cost to find the appropriate bucket is 1-2 I/Os.
2. Once we find the appropriate leaf or bucket, the cost of retrieving matching S tuples depends on whether the index is clustered. If it is, the cost per outer tuple $r \in R$ is typically just one more I/O. If it is not clustered, the cost could be one I/O per matching S-tuple (since each of these could be on a different page in the worst case).

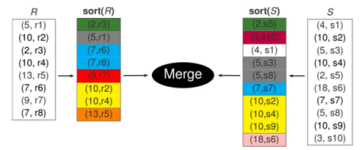
Sort-Merge Join

- Basic idea: Sort both relations on join attribute and look for qualifying tuples by essentially merging the two relations.
- Exploit partitioning, compare R tuples with only S tuples in same partition (rather than all S tuples), avoid enumeration of cross-product of R & S. (Works only for equality join conditions.)

Sort-Merge Join

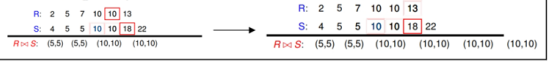
Idea: Sort both relations based on join attributes & merge them.
Partition: Sorted relation R consists of *partitions* R_i of records, where records have same values for the join attribute(s)

Sorting: Use external merge sort



Merging:
- Assume R is outer relation & S is inner relation
- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a "smaller" tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer

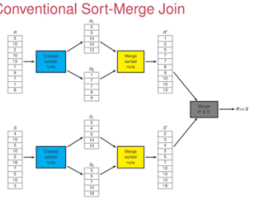
Example:



Analysis: I/O cost = Cost to sort R + Cost to sort S + Merging Cost

- **Cost to sort R** = $2|R|\,(\log_m(N_R)+1)$ if using external merge sort
  - N_R = number of initial sorted runs of R, m = merge factor
- **Cost to sort S** = $2|S|\,(\log_m(N_S)+1)$ if using external merge sort
  - N_S = number of initial sorted runs of S, m = merge factor
- If each S partition is scanned at most once during merging,
  - **Merging cost** = $|R|+|S|$
- Worst case occurs when each tuple of R requires scanning entire S!
  - **Merging cost** = $|R|+||R||\times|S|$

Conventional Sort-Merge Join

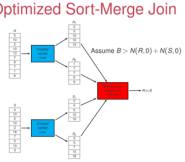


Conventional Sort-Merge Join:
- **Sort R:** Create sorted runs of R; Merge sorted runs of R
- **Sort S:** Create sorted runs of S; Merge sorted runs of S
- **Join R & S:** Merge sorted R and sorted S.

Optimization: Combine merge phase of sorting & merge phase of join.
- It's not necessary to merge sorted runs into a single run before performing join
- If $B > N(R,i) + N(S,j)$ for some i & j, sorting of R and S can stop
  - ⋆ N(R,i) = total number of sorted runs of R at the end of pass i of sorting R
- **Sort R:** Create sorted runs of R; Merge sorted runs of R partially
- **Sort S:** Create sorted runs of S; Merge sorted runs of S partially
- **Join R & S:** Merge remaining sorted runs of R & S and join at same time.

Optimized Analysis:

Optimized Sort-Merge Join



- Assume $|R| \leq |S|$
- If $B > \sqrt{2|S|}$
  - Number of initial sorted runs of S < $\sqrt{\frac{|S|}{2}}$
  - Total number of initial sorted runs of R and S < $\sqrt{2|S|}$
  - One pass is sufficient to merge and join the initial sorted runs R & S
  - I/O Cost = $\underbrace{2 \times (|R|+|S|)}_{\text{pass 0}} + \underbrace{(|R|+|S|)}_{\text{pass 1}} = 3 \times (|R|+|S|)$

Hash Join

- Hash join algorithm, like sort-merge join, identifies partitions in R & S in partitioning phase, and compares tuples in corresponding partitions.
- **Idea**: Unlike sort-merge join, use **hashing to identify partitions** over sorting. Hash both relations on join attribute using **same hash function** h.
- Partitioning (building) phase of hash join similar to partitioning in hash-based projection.

Grace Hash Join

Hash Join, $R \bowtie_{R.A=S.B} S$

Idea:

- Partition R and S into k partitions using some hash function h
  - $R = R_1 \cup R_2 \cup \cdots \cup R_k$, $t \in R_i$ iff $h(t.A) = i$
  - $S = S_1 \cup S_2 \cup \cdots \cup S_k$, $t \in S_i$ iff $h(t.B) = i$
  - $\pi_A(R_i) \cap \pi_B(S_j) = \emptyset$ for each R_i & S_j, $i \neq j$
- Joins corresponding pair of partitions
  - $R \bowtie S = (R_1 \bowtie S_1) \cup (R_2 \bowtie S_2) \cup \cdots \cup (R_k \bowtie S_k)$

Grace Hash Join, $R \bowtie_{R.A=S.B} S$

Three Phases:

1. Partition R into $R_1, \cdots, R_k$
2. Partition S into $S_1, \cdots, S_k$
3. Probing phase: probes each R_i with S_i
   - Read R_i to build a hash table
   - Read S_i to probe hash table

R is the **build relation.**
S is the **probe relation.**

Partitioning (building) phases

```
initialize a hash table T with k buckets
for each tuple r ∈ R do
    insert r into bucket h(r.A) of T
write each bucket R_i of T to disk
initialize a hash table T with k buckets
for each tuple s ∈ S do
    insert s into bucket h(s.B) of T
write each bucket S_i of T to disk
```

Probing (matching) phase

```
for i = 1 to k do
    initialize a hash table T
    for each tuple r in partition R_i do
        insert r into bucket h'(r.A) of T
    for each tuple s in partition S_i do
        for each tuple r in bucket h'(s.B) of T do
            if r and s matches then output (r, s)
```

Grace Hash Join Analysis:

To minimize size of each partition of R_i: Set k = B − 1 (B buffer pages, k partitions).

Assuming uniform hashing distribution:

- size of each partition R_i is $\frac{|R|}{B-1}$
- size of hash table for R_i is $\frac{f \times |R|}{B-1}$, where f is a fudge factor
- During probing phase, $B > \frac{f \times |R|}{B-1} + 2$
  (with one input buffer for S_i & one output buffer)
- Approximately, $B > \sqrt{f \times |R|}$

Partition Overflow Problem: If hash table for R_i does not fit in memory:

- **Solution:** Recursively apply partitioning to overflow partitions

I/O Cost = Cost of Partitioning Phases + Cost of Probing Phase

- I/O Cost $= \underbrace{2 \times (|R| + |S|)}_{\text{partitioning}} + \underbrace{(|R| + |S|)}_{\text{probing}} = 3 \times (|R| + |S|)$

  if there's no partition overflow problem