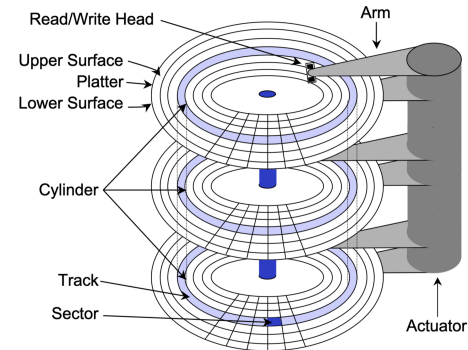


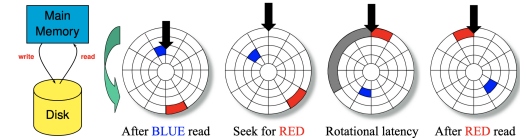
## 01. DBMS STORAGE

- store data on non-volatile disk
- process data in main memory (RAM) (*volatile storage*)

### Magnetic HDD



- disk access time** =
  - seek time** → move arms to position disk head on track.  
 $\text{Total(sequential)} = \text{num of cylinders} \times \text{seek time}$
  - rotational delay** → wait for block to rotate under head
    - average rotational delay = time for  $\frac{1}{2}$  revolutions
    - $= 0.5 \times \frac{60}{\text{rotational speed (RPM)}}$
  - transfer time** → move data to/from disk surface  
 $= \text{time for 1 revolution} \times \frac{\# \text{ of requested sectors on the same track}}{\# \text{ of sectors in track}}$
- response time** for disk access = queuing delay + access time



- command processing time: interpreting access command by disk controller (part of access time, considered negligible)
- small requests are dominated by seek time; large requests dominated by transfer time
- access order**:
  - contiguous blocks within the same track (same surface)
  - cylinder tracks within the same cylinder
  - next cylinder

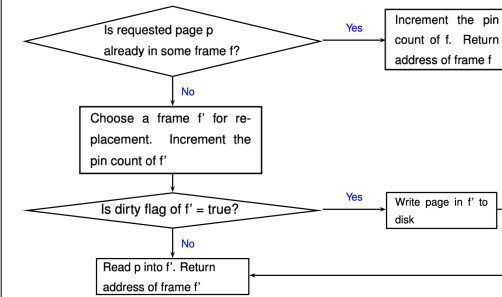
### SSD (Solid-State Drive)

- no mechanical moving parts
- advantages: ✓ significantly faster than HDD  
 ✓ higher data transfer rate ✓ lower power consumption
- disadvantages: ✗ update to a page requires erasure of multiple pages before overwriting page  
 ✗ limited number of times a page can be erased

### Buffer Manager

- data is stored & retrieved in **disk blocks** (pages)
  - each block = sequence of  $\geq 1$  contiguous sectors
- buffer pool**: main memory allocated for DBMS

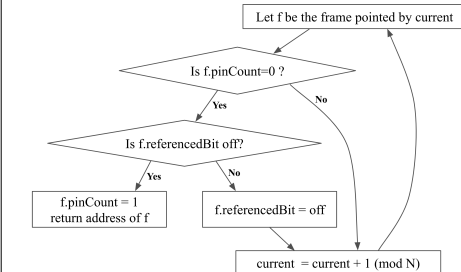
- partitioned into **frames** (block-sized pages)
- pin count**: number of clients using page (initialised 0)
  - $> 0 \Rightarrow$  page is utilised by some transaction; don't replace
- dirty flag**: initialised false
  - dirty** → page is modified & not updated on the disk
  - dirty page must be written back to the disk if the transaction has committed



! unpinning: update dirty flag to true if page is dirty

### replacement policies

- decide which unpinned (pinCount==0) page to replace
- LRU** uses a queue of pointers to frames with pinCount==0
- clock**: cheaper than LRU, used in postgres
  - referenced bit - turns on when pinCount==0
  - replace page with referenced bit off && pinCount==0

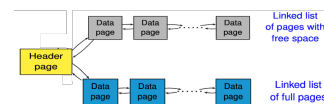


### File abstraction

- each relation is a file of records
- each record has a unique record identifier, **RID**
- heap file** → unordered file
  - vs sorted/hashed file: records are ordered/hashed

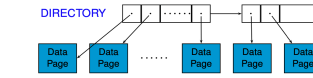
### heap file implementations

- linked list** implementation
  - header page: metadata about the file



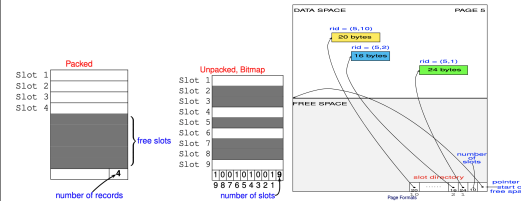
- page directory** implementation: more efficient
  - maintain directory structure with one entry per page
    - stores address of and amount of free space on page
  - insertion: scan directory to find page with enough space to store the new record

- insertion worst case: scan number of pages + data page itself (vs LL worst case: entire list)



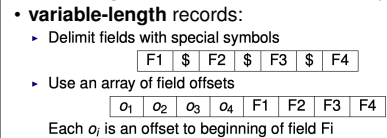
### Page Formats

- RID = (page ID, slot number)
- fixed-length** records
  - packed organisation: inefficient deletion (transferring last record to deleted record changes RID of record)
- variable-length** records: **slotted page organisation**



### Record formats

- fixed-length** records: store consecutively
- variable-length** records:



### Data entry formats

- $k^*$  is an actual **data record** (with search key  $k_i$ )
- $k^*$  is of the form  $(k, \text{RID})$  - fixed length  $(k, \bullet)$
- $k^*$  is of the form  $(k, \text{RID-list})$  - e.g.  $(k, \{\text{RID11}, \text{RID12}\})$

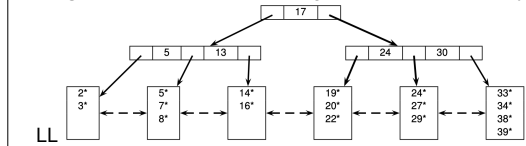
## 02. TREE-BASED INDEXING

- search key** → sequence of  $k$  data attributes,  $k \geq 1$ 
  - composite search key** → if  $k > 1$
- unique index** → search key is a candidate key
- clustered index** → order of data entries  $\approx$  order of records
  - Format-1** is always clustered
  - at most one clustered index for each relation
- dense index** → there is an index record for every search key value in the data. *unclustered index* must be dense

### B<sup>+</sup>-tree Index

- leaf nodes: sorted data entries ( $k^*$  is of form  $(k, \text{RID})$ )
- internal nodes: stores index entries  $(p_0, k_1, p_1, \dots, p_n)$  for  $k_1 < k_2 < \dots < k_n$  where  $p_i$  is the page disk address
  - each  $(k_i, p_i)$  is an **index entry**
  - for  $k^*$  in index subtree  $T_i$  rooted at  $p_i$ ,  $k \in [k_i, k_{i+1}]$
- order** of index tree,  $d \in \mathbb{Z}^+$ 
  - each non-root node contains  $m$  entries,  $m \in [d, 2d]$
  - root node contains  $[1, 2d]$  entries
- equality search**: at each internal node  $N$ , find the largest  $k_i$  s.t.  $k \geq k_i$ . search subtree at  $p_i$  if  $k_i$  exists, else  $p_0$

- range search**: find first matching record; traverse doubly



### insertion: splitting

- splitting leaf node: distribute  $d + 1$  entries to a new leaf node
- if parent overflows: push the middle  $(d+1)$  key up to parent
- root node overflows: create new root (parent of current root)

### insertion: redistribution (of leaf nodes only)

- try right sibling first, then left sibling, else use splitting
- sibling** → two nodes at the *same level* & *same parent* node

**deletion: redistribution** - try right sibling, then left, else merge

**deletion: merging** (siblings have  $d$  entries) - try right first

- if leaf underflows: delete parent key, combine with sibling
- if internal node underflows: pull down its index entry in parent, combine with sibling, push a key back up
  - becomes the new root if parent is root & becomes empty

### finding order of B<sup>+</sup> tree with page byte size

$$\geq 2d(\text{key byte size}) + (2d + 1)(\text{disk page addr byte size})$$

### min number of nodes at level i

$$= 2 \times (d + 1)^{i-1} \text{ for } i \geq 1$$

### max number of nodes at level i

$$= 2(d + 1)^i$$

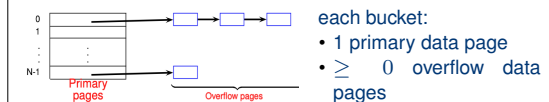
### Bulk Loading a B<sup>+</sup>-tree

- sort data entries by search key and store sequentially
- construct leaf pages with  $2d$  entries
- construct internal pages by attempting to insert leaf pages into rightmost parent page

## 03. HASH-BASED INDEXING

### Static Hashing

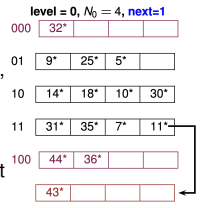
- hash record to  $B_i \in B_0, \dots, B_{N-1}$  with  $i = h(k) \bmod N$
- when full, reconstruct hash table with more buckets



### Linear Hashing (Dynamic)

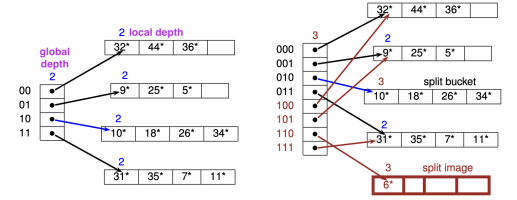
- grows **linearly**: split when some **bucket overflows**
- how to split bucket  $B_i$ :
  - add a new bucket  $B_j = B_i + N_i$  (split image of  $B_i$ )
  - redistribute entries in  $B_i$  between  $B_i$  and  $B_j$
  - next++; if next== $N_{\text{level}}$ : level++; next=0
- file size at the beginning of round  $i$ ,  $N_i = 2^i N_0$
- at round  $i$ , hash  $x = B_x$  has been split?  $h_i(k) : h_{i+1}(k)$
- $N_0 = 2^m$ ;  $N^i = 2^{m+i}$
- Look at  $m + i + 1$  bits if bucket has been split,  $m + i$  bits only otherwise

- performance:** 1 disk I/O (no overflow pages)
  - avg 1.2 I/Os (uniform distribn), worst case linear I/O cost
- removing last bucket (**deletion**):
  - if next > 0: next--;
  - else: next=(prev level last bucket); level--;



## Extendible Hashing (Dynamic)

- add a new bucket whenever existing bucket overflows
  - no overflow pages unless # collisions  $\geq$  page capacity
- directory of pointers to buckets -  $2^d$  entries ( $b_d b_{d-1} \dots b_1$ )
  - $d =$  **global depth** of hashed file
- corresponding** directory entries differ only in the  $d^{th}$  bit
- entries in a bucket of **local depth**  $\ell \in [0, d]$ : same last  $\ell$  bits
  - a split bucket & its image have the *same local depth*
- number of directory entries pointing to a bucket =  $2^{d-\ell}$
- Look at last  $\ell$  bits when adjusting pointers



- splitting bucket:  $\ell++$  (repeat until no more overflow)
  - if  $\ell = d$ : directory doubles;  $d++$
  - else  $\ell < d$ : redistribute and increment  $\ell$
- deletion: if bucket  $B_i$  becomes empty or  $B_i$  and  $B_j$  can merge,
  - deallocate  $B_i$  and decrement  $\ell--$  for split image  $B_j$
  - if each pair of corresponding entries point to the same bucket, the directory can be halved
- performance:** at most 2 disk I/Os (for equality query)
- collisions: when 2 data entries have the same hashed value
  - use **overflow pages** if # collisions exceeds page capacity

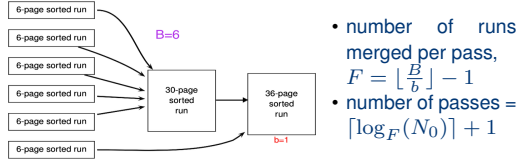
## 04.1 SORTING

### External Merge Sort

- sorted run**  $\rightarrow$  sorted data records written to a file on disk
- $N_0 = \lceil N/B \rceil$
- Num of passes =  $\lceil \log_{B-1} N_0 \rceil + 1$
- divide and conquer
  - create temporary file  $R_i$  for each  $B$  pages of  $R$  sorted
  - merge: use  $B-1$  pages for input, 1 page for output
- total I/O =  $2N(\lceil \log_{B-1} (N_0) \rceil + 1)$ 
  - $2N$  to create  $\lceil N/B \rceil$  sorted runs of  $B$  pages each
  - merging sorted runs:  $2N \times \lceil \log_{B-1} N_0 \rceil$

### optimisation with blocked I/O

- sequential I/O - read/write in *buffer blocks* of  $b$  pages
- one block ( $b$  pages) for output, remaining blocks for input



### Sorting with B+ -trees

- when *sort key is a prefix of the index key* of the B+ -tree
- sequentially scan leaf pages of B+ -tree
  - for Format-2/3, use RID to retrieve data records

## 04.2 SELECTION: $\sigma_p(R)$

- $\sigma_p(R)$ : selects rows from relation  $R$  satisfying predicate  $p$
- access path**: a way of accessing data records/entries
  - table scan**  $\rightarrow$  scan all data pages
  - index scan**  $\rightarrow$  scan index pages
  - index intersection**  $\rightarrow$  combine results from index scans
- selectivity** of an access path  $\rightarrow$  number of index & data pages retrieved to access data records/entries
  - more selective = fewer pages retrieved
- index  $I$  is a **covering index** for query  $Q \rightarrow$  if all attributes referenced in  $Q$  are part of the key or include cols of  $I$ 
  - $Q$  can be evaluated using  $I$  without any RID lookup (**index-only** plan)

### Matching Predicates

- term**  $\rightarrow$  of form  $R.A \text{ op } c$  or  $R.A_i \text{ op } R.A_j$
- conjunct**  $\rightarrow$  one or more terms connected by  $\vee$ 
  - disjunctive** conjunct  $\rightarrow$  contains  $\vee$
- conjunctive normal form, **CNF predicate**  $\rightarrow$  comprises one or more conjuncts connected by  $\wedge$ 

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

### B+ -tree matching predicates

- for index  $I = (K_1, K_2, \dots, K_n)$  and non-disjunctive CNF predicate  $p$ ,  $I$  matches  $p$  if  $p$  is of the form  $(K_1 = c_1) \wedge \dots \wedge (K_{i-1} = c_{i-1}) \wedge (K_i \text{ op}_i c_i)$ ,  $i \in [1, n]$ 
  - zero or more equality predicates
  - at most one non-equality comparison operator which must be on the last attribute of the prefix ( $K_i$ )
- matching index: matching records are in contiguous pages
  - non-matching index: not contiguous  $\Rightarrow$  less efficient

### Hash index matching predicates

- for hash index  $I = (K_1, K_2, \dots, K_n)$  and non-disjunctive CNF predicate  $p$ ,  $I$  matches  $p$  if  $p$  is of form  $(K_1 = c_1) \wedge (K_2 = c_2) \wedge \dots \wedge (K_n = C_n)$

### Primary/Covered Conjuncts

- primary conjuncts**  $\rightarrow$  subset of conjuncts that  $I$  matches
  - e.g.  $p = (\text{age} \geq 18) \wedge (\text{age} \leq 20) \wedge (\text{weight}=65)$  for  $I = (\text{age, weight, height})$
- covered conjuncts**  $\rightarrow$  subset of conjuncts covered by  $I$ 
  - each attribute in covered conjuncts appears in key or include cols of  $I$
- primary conjuncts  $\subseteq$  covered conjuncts

## Cost of Evaluation

let  $p'$  = primary conjuncts of  $p$ ,  $p_c$  = covered conjuncts of  $p$

### B+ -tree index evaluation of p

- navigate internal nodes to find first leaf page
 
$$\text{cost}_{\text{internal}} = \begin{cases} \lceil \log_F(\lceil \frac{|R|}{b_d} \rceil) \rceil & \text{if } I \text{ is a format-1 index} \\ \lceil \log_F(\lceil \frac{|R|}{b_i} \rceil) \rceil & \text{otherwise} \end{cases}$$
- scan leaf pages to access all qualifying data entries
 
$$\text{cost}_{\text{leaf}} = \begin{cases} \lceil \frac{|\sigma_{p'}(R)|}{b_d} \rceil & \text{if } I \text{ is a format-1 index} \\ \lceil \frac{|\sigma_{p'}(R)|}{b_i} \rceil & \text{otherwise} \end{cases}$$
- retrieve qualified data records via RID lookups
 
$$\text{cost}_{\text{RID}} = \begin{cases} 0 & \text{if } I \text{ is a covering format-1 index,} \\ |\sigma_{p_c}(R)| & \text{otherwise} \end{cases}$$
  - reduce cost with **clustered** data records (sort RIDs):  $\lceil \frac{|\sigma_{p_c}(R)|}{b_d} \rceil \leq \text{cost}_{\text{RID}} \leq \min\{|\sigma_{p_c}(R)|, |R|\}$

### hash index evaluation of p

- format-1:** cost to retrieve data records  $\geq \lceil \frac{|\sigma_{p'}(R)|}{b_d} \rceil$
  - format-2:** cost to retrieve data entries  $\geq \lceil \frac{|\sigma_{p'}(R)|}{b_i} \rceil$
- cost to retrieve data records =  $\begin{cases} 0 & \text{if } I \text{ is a covering index,} \\ |\sigma_{p_c}(R)| & \text{otherwise} \end{cases}$

## 05.1 PROJECTION $\pi_{A_1, \dots, A_m}(R)$

- $\pi_L(R)$  eliminates duplicates,  $\pi_L^*(R)$  preserves duplicates

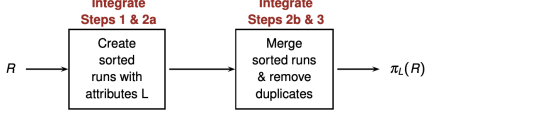
### Sort-based approach



### cost analysis

- extract attributes:  $|R|$  scan +  $|\pi_L^*(R)|$  output temp result
- sort records:  $2|\pi_L^*(R)|(\log_m(N_0) + 1)$
- remove duplicates:  $|\pi_L^*(R)|$  to scan records

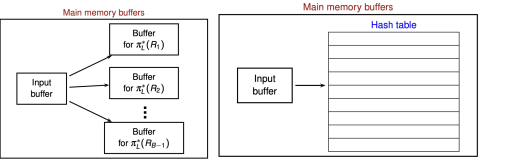
### optimised sort-based approach



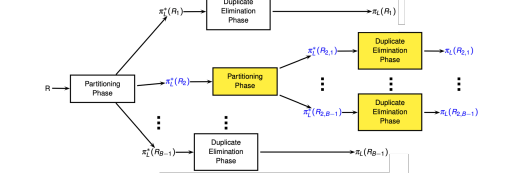
- if  $B > \sqrt{|\pi_L^*(R)|}$ , same I/O cost as hash-based approach
  - $N_0 = \lfloor \frac{|R|}{B} \rfloor \approx \sqrt{|\pi_L^*(R)|}$  initial sorted runs
  - $\log_{B-1}(N_0) \approx 1$  merge passes

### Hash-based approach

- partitioning phase:** hash each tuple  $t \in R$ 
  - $R = R_1 \cup R_2 \cup \dots \cup R_{B-1}$ 
    - for each  $R_i$  &  $R_j$ ,  $i \neq j$ ,  $\pi_L^*(R_i) \cap \pi_L^*(R_j) = \emptyset$
  - for each  $t$ : project attributes to form  $t'$ , hash  $h(t')$  to one output buffer, flush output buffer to disk when full
  - one buffer for input,  $(B-1)$  buffers for output
- duplicate elimination** from each partition  $\pi_L^*(R_i)$ 
  - for each  $R_i$ : initialise in-mem hash table, hash each  $t \in R_i$  to bucket  $B_j$  with  $h' \neq h$ , insert if  $t \notin B_j$
  - write tuples in hash table to results



- I/O cost** (no partition overflow):  $|R| + 2|\pi_L^*(R)|$ 
  - partitioning cost:  $|R| + |\pi_L^*(R)|$
  - duplicate elimination cost:  $|\pi_L^*(R)|$
- partition overflow: recursively apply partitioning
  - to avoid,  $B >$  size of hash table for  $R_i = \frac{|\pi_L^*(R)|}{B_1} \times f$ 
    - approximately  $B > \sqrt{f \times |\pi_L^*(R)|}$



### Projection using Indexes

- if index search key contains all wanted attributes as a *prefix*
  - index scan** data entries in order & eliminate duplicates

## 05.2 JOIN $R \bowtie_{\theta} S$

$R$  = outer relation (smaller relation);  $S$  = inner relation  
 ! for **format-2** index, add cost of retrieving record

### nested loop joins and UDA

- tuple-based** nested loop join:  $|R| + ||R|| \times |S|$
- page-based** nested loop join:  $|R| + |R| \times |S|$
- block nested loop join:**  $|R| + (\lceil \frac{|R|}{B-2} \rceil \times |S|)$ ,  $|R| \leq |S|$ 
  - 1 page output, 1 page input,  $(B-2)$  pages to read  $R$
  - for each  $(B-2)$  pages of  $R$ : for each  $P_S$  of  $S$ : check  $r, s$
- index nested loop join:**

$$|R| + ||R|| \times \left( \underbrace{\log_F(\lceil \frac{||S||}{b_d} \rceil)}_{\text{internal nodes}} + \underbrace{\lceil \frac{||S||}{b_d \times |\pi_{B_j}(S)|} \rceil}_{\text{leaf nodes}} + \underbrace{c}_{\text{lookup}} \right)$$

- joining  $R(A, B) \bowtie_A S(A, C)$  with B+tree index on  $S.A$ 
  - for each tuple  $r \in R$ , use  $r$  to probe  $S$ 's index for match
- In a  $B^+$  tree or Hash Index, assume each leaf/bucket has same amount of entries
- If  $R \bowtie_{R.a=S.b} S$  and  $S.b$  is the primary key. Assume each  $S$  tuple joins with  $||R||/||S||$  tuples

## 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