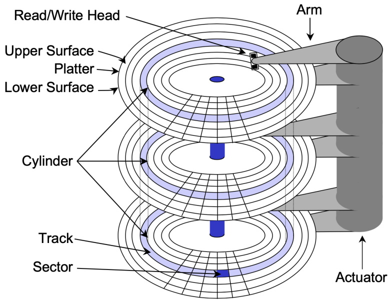
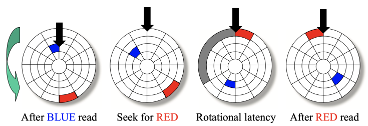
Storage

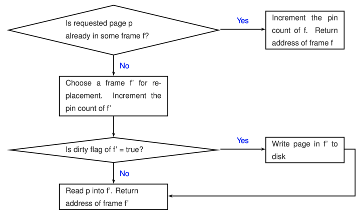
Metadata. Relation schemas, view definitions, indexes, stats.
Log files. Information maintained for data recovery.
Primary Memory. Registers, static RAM (caches), dynamic RAM (physical memory).
Secondary Memory. Magnetic disks (HDD), SSD.
Tertiary Memory. Optical disks, tapes.
Storage. DBMS stores data on non-volatile disk for persistence. Processing data in main memory (RAM).



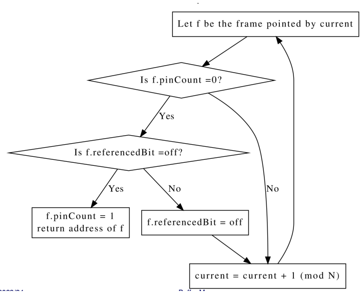
Seek time. Moving arms to position disk head on track.
Rotational Delay. Waiting for block to rotate under head. Depends on rotation speed. Average delay is time for half revolution.
Transfer time. Actually moving data to/from disk surface. number of requested sectors on same track $\times$ time for 1 revolution divided by number of sectors per track.
Access Time. Sum of seek time, rotational delay and transfer time.
Response time. queueing delay + access time.



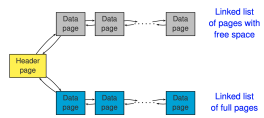
Sequential IO. Everything written on same track.
Random IO. Things may be written on different tracks. If the size of the data is grater than the track, store in the same cylinder.
SSD. No moving parts so faster random IO. Updates to page requires erasure of multiple pages before overwriting. Limited number of times a page can be erased.
Disk blocks. Data is stored/retrieved from here. Each is a sequence of ≥ 1 contiguous sectors.
File layer. Deals with organisation and retrieval of data.
Buffer Manager. Controls reading/writing of disk pages.
Disk Space Manager. Keeps track of pages used by file layer.
Buffer Pool. Main memory allcoated for DBMS.
Frames. A partition of the buffer pool into block-sized pages.
Dirty page. Page in the buffer that has been modified and not updated on disk.
Pin count. Number of clients using the page initialised to 0.
Dirty flag. Whether page is dirty initialised to false.



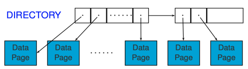
Pinning. Incrementing pin count by pinning the requested page in its frame.
Unpinning. Decreasing pin count. Dirty flag should be true if page is dirty.
Replacement. Only when pin count is 0. Before that write to disk if dirty flag is true.
Replacement Policy. Decide which unpinned page to replace.
LRU. A queue of pointers to frames with pin count 0. Exploits locality. Spatial at the page level and temporal at the record level.
Clock. Current variable points to some buffer frame. Each frame has a reference bit (on when pin count is 0). Replace a page that has reference bit off and pin count 0.



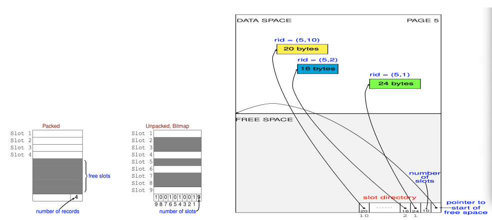
Files. Each relation is a file of records. Each record has a unique record identifier called RID.
Heap file. Unordered file.



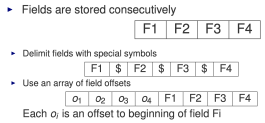
Header page contains metadata about the file. Once a page with free space is used, move it to the full pages.



Each disk block has an entry for a disk page. Go through directory page to find a data page. To insert, look for a data pages in the same directory that has space. Otherwise go to the next directory.

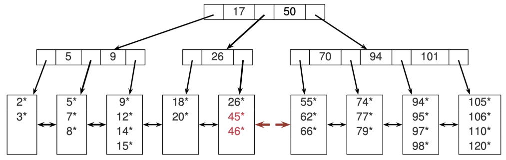


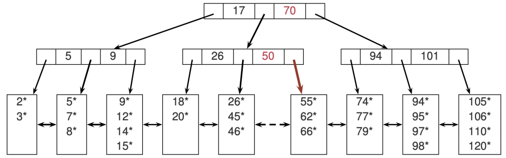
RID. Tuple pair of page id and slot number.
Packed Org. Store records in contiguous slots.
Unpacked Org. Uses a bit array to maintain free slots.



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. Otherwise non-unique.
Data entries. Index stored as a file an records in an index file.
B^+ tree. Leaf nodes are at the bottom most level. Root node is at level 0.
Height of tree. Leafs nodes are at level h.
Sibling nodes. Nodes at the same level if they have the **same parent nodes**.
Leaf Nodes. Store sorted data entries of tuple pair of search key value and RID. Doubly linked.
Internal nodes. Stored in the form $p_0, k_1, p_1, \ldots$. Keys are sorted $k_1 < k_2 < k_3, \ldots$. p_i is the disk page address. For $k*$ in index subtree of T_i rooted at $p_i, k* \in [k_i, k_{i+1})$
Index entry. A k_i, p_i pair. k_i is the separator between node contents pointed to by p_{i-1}, p_i.
Order. Controls space utilisation. Each non-root node conains m entries where $m \in [d, 2d]$. Root node contains m entries, where $m \in [1, 2d]$.
Equality search. At each internal node, find the largest key of the node such that $k \geq k_i$. If k_i exists, search subtree at p_i otherwise search subtree at p_0.
Range search. Find the node based on equality then traverse the doubly LL to find the remaining.
Format 1 data entry. $k*$ is an actual data record with search key value of k.
Format 2 data entry. $k*$ is a pair of search key value k and RID.
Format 3 data entry. $k*$ is a pair of search key value k and a list of RIDs.
Insertion. Find the leaf node that to insert into by performing searching.
Overflowed node. Node already has $2d$ entries and a new entry is being inserted into it.
Splitting. Get all the records in the leaf (including the one to be inserted) and sort them ascending. Then take the $d+1$ entries and put into a new leaf node. Create a new index using the **smallest** key in the **new** leaf node. Insert it into parent node of overflowed.
Propagation of splits. Internal nodes overflow when the number of keys exceed $2d$. Get all the keys and sort them asecnding. Take the middle key and push it to the parent.
Redistribution. Only at leaf nodes. Perform distribution to a non-full adjacent sibling and update the index accordingly. Try right sibling first then left otherwise use splitting.
Deletion. Find the leaf node that the record is in and remove it.
Underflowed. Non-root node. If the result of deletion causes the number of entries to be $d-1$.
Redistribution of leaves. An underflowed nodes can be redistributed using an adjacent sibling entry. Try right sibling then left. Otherwise split.
Merging Condition. Both siblings have d entries. Merge with the right one.
Merging at leaves. If can merge, remove the parent key and combine with the sibling.
Propogation of nodes merging. If an internal node underflows after merging the leaves, pull down the separating key from parent node to form a merged nodes.
Balancing underflowed node. Try the right sibling first. Push the smallest one up to the parent. Then bring the middle key down.

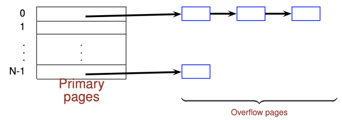




Bulk loading. Sort all data entries by search keys and load the leaf pages with sorted entries. Construct the leaf pages with $2d$ entries. Insert pages into rightmost parent-of-leaf level page of tree.
Advantages. Efficient construction algorithm. Leaf pages are allocated sequentially.
Clustered Index. Order of data is the same or close to the order data records. Otherwise its **unclustered**. Format 1 is clustered. At most 1 clustered index for each relation.
Dense index. There is an index record for every search key value in the data. Otherwise its **sparse**. Unclustered index must be dense. Clustered can be sparse or dense.

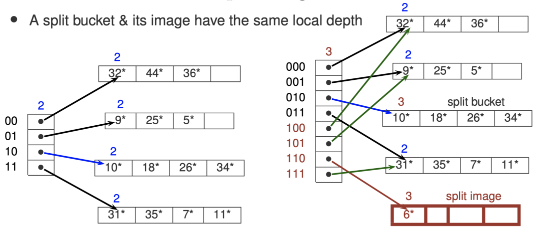
Hash Index

Static Hashing. Hash function to identify which bucket data should be stored. 1 primary data page and overflow pages are chained to the primary data page. Perform modulo of N to decide which bucket.



Linear Hashing. Grow linearly by splitting of buckets.
Splitting a bucket. Add a new bucket (split image) and redistribute across split image and the original. Increase the value of next by 1. If the value of next becomes the same as level, reset value of next and increase level by 1.
Hash functions. Each round has 2 hash functions. If a bucket has been split, use the new hash function else use the current one.
Redistribution. To determine whether a record should move or not, and if the original table size is s, perform the record mod $2s$. If the result is 1, move it to the split image else remains the same.

Insertion. Find the bucket to insert into. If the bucket has space, just insert it. Otherwise, look at the value of next. Then split the bucket that overflowed (may not be the one at next). If its the one at next, may have to insert it into the split image.
Deletion. Find the entry and delete the entry. If the bucket becomes empty, delete it. If the value of next > 0, reduce next by 1. else next $= 0$ and level > 1 then update next to be the last bucket and level $--$ 1. if level is 0, dont do anything.
Performance. 1 disk IO unless theres overflow pages. On average, 1.2 disk IO for uniform distributed data. Worst case IO is linear in number of data entries.
Extendible Hashing. Double the entire directory when attempting to resolve overflow.
Global Depth. d. Size of directory always 2^d. Each directory has unique d bit address (last d bits).
Correspondance. 2 directory entries correspond if only the dth bit is different.
Local depth. Maintained by each bucket. All entries in the bucket have the same last l bits where l is the local depth.
Overflow bucket. Allocate a new bucket called split image. Then redistribute everything (including new entry) between split bucket and split image.
Splitting. l increases by 1 when the bucket is split. Split bucket and image have the same local depth. When directory is doubled, each new directory entry (except split image) points to the same bucket of its corresponding.
Directory expanding. Directory only expands when the global depth is the same as local depth. The result is a directory that is double the size. Number of directories pointing to a bucket is 2^{d-l}.



Directory is expanded by insertion of 6* (110)

Splitting (case 2). When local depth is smaller than global. Increase the local depth by 1. Create the split image and redistribute.
Deletion. Merging can happen if the number of entries can fit into the same bucket. If each pair point to the same bucket, directory can be halved. d can decrease by 1.
Performance. At most 2 IO for equality. At most 1 if disk IO directory fits in main memory.

Sort-Select

Sorted Run. Writing sorted data records to a file on disk. **External Merge sort.** Use B number of buffer pages.
Creating sorted runs. Read in and sort B pages at a time. Number of sorted runs created is $\lceil \frac{N}{B} \rceil$. Size of each sorted run is B, maybe except the last.
Merging sorted runs. Use $B-1$ buffer pages for input and 1 for output. Perform $B-1$ way merge.
Analysis. Total number of passes is $\lceil log_{B-1}(N_0) \rceil + 1$. Total number of IO is $2N$ times number of passes. Each pass reads and writes N pages.
Optimisation with blocked IO. Read and write in units of buffer blocks of b pages. Use 1 buffer block for output.
Analysis. Number of runs that can be merged at each pass is $\lfloor \frac{B}{b} \rfloor + 1$. Number of passes is ceiling of log base number of runs, then add 1.
Sorting with B-tree. When table to be sorted has a B-tree index on the sorting attribute.
Format 1 B-tree. Sequentially scan leaf pages.
Format 2,3 B-tree. Sequentially scan leaf pages and for each leaf page visited, retrieve data records using RID.
Selection. Selecting rows that satisfy a predicate.
Access Path. Method of accessing data records/entries.
Table scan. Scan all the data pages.
Index scan. Scan all index pages.
Index intersection. Combine results from multiple index scans.
Selectivity of access Path. Number of index and data pages retrieved to access data records/entries. The most selective access path retrieves the fewest pages.
Covering Index. An index for a query where all the attributes referenced in the query are part of the key or include columns of the index. The query can be evaluated using the index **without** any RID lookup. Known as **index-only plan**.
Term. Operations between an attribute and a constant or between attributes.
Conjunct. 1 or more terms conencted by OR.
Disjunctive. Conjunct that contains OR.
Conjunctive Normal Form(CNF) predicate. Consists of 1 or more conjuncts connected by AND.

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

B+ tree Matching predicates. If a B+ tree has the following index $K_1, K_2, \dots$ and we have a non-disjunctive CNF predicate p, then the index 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 $K_1, \dots, K_i$ is a **prefix** of the key of I **and** there is **at most one** non-equality operator which must be on the **last** attribute of the prefix (K_i).
Hash Index Matching predicates. Hash index $I = K_1, K_2, \dots$. Non-disjunctive CNF predicate p. Then I matches p if p is of the form

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

Primary Conjuncts. Subset of conjuncts in a selection predicate that matches an index.
Covered Conjunct. All attributes in the conjunct in the predicate appears in the key or include columns of the index. Primary conjuncts is a proper subset of covered conjuncts.
Cost of evaluating p in B+ tree. Navigate internal nodes to locate first leaf page

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

Scanning leaf pages to access all qualifying data entries

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

Retrieving qualified data records via RID lookup

$$\text{cost}_{\text{RID}} = \begin{cases} 0 & \text{if I is a covering format-1,} \\ ||\sigma_{p_c}(R)|| & \text{otherwise} \end{cases}$$

Reduce cost of RID lookup by first sorting the RID (making it clustered)

$$\lceil \frac{||\sigma_{p_c}(R)||}{b_d} \rceil \leq \text{cost}_{RID} \leq \min\{||\sigma_{p_c}(R)||, |R|\}$$

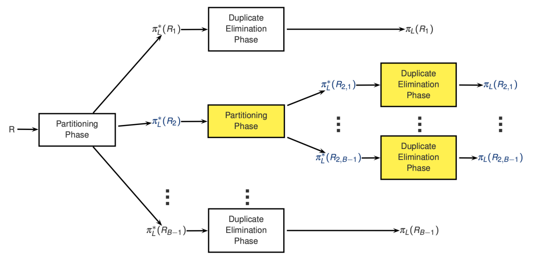
Cost of hash index evaluation. For format 1: cost to retrieve data records $\geq \lceil \frac{||\sigma_{p'}(R)||}{b_d} \rceil$ For format-2, cost to retrieve data entries $\geq \lceil \frac{||\sigma_{p'}(R)||}{b_i} \rceil$.
Cost to retrieve data records is

$$\begin{cases} 0 & \text{if I is a covering index,} \\ ||\sigma_{p'}(R)|| & \text{otherwise} \end{cases}$$

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

Projection and Join

Projection. Select columns given by a list L.
Notation. $\pi_L(R)$ preserves duplicates while $\pi_L^*(R)$ does not.
Primary tasks. Remove unwanted attributes then eliminate any duplicates.
Sort-based Approach. Extract attributes L from records to get $\pi_L^*(R)$. Then sort the records using attributes L as sort key to get sorted $\pi_L^*(R)$. Remove the duplicates to get $\pi_L(R)$. Better if there are many duplicates or if distribution of hashed values is non-uniform.
Cost of extracting. Scan the records $|R|$ and the cost to output the result $|\pi_L^*(R)|$.
Cost of sorting records. $2|\pi_L^*(R)|(log_m(N_0) + 1)$ where N_0 is the number of initial sorted runs and m is the merge factor.
Removing duplicates. Cost to scan the records is $|\pi_L^*(R)|$.
Optimised sorting approach. Split the sorting into 2 steps - creating and merging the sorted runs. Combine the creation step with the extraction and merging with removing duplicates.
Hash-based. Build a main-memory hash table to detect duplicates. Cost will be the size of the relation if the table fits in memory.
Partitioning phase. Partition into $B-1$ parts. Use 1 buffer for input and remaining for output. Read 1 page at a time into input buffer. For each tuple in input buffer, project out unwanted attributes. Then apply the hash function to distribute the tuple into 1 of the output. Flush the output buffer to disk whenever buffer is full.
Duplicate elimination. For each partition R_i, initialise an in-memory hash table. Read $\pi_L^*(R_i)$ 1 page at a time. For each tuple read, hash into a bucket using a different hash function and insert it if its not duplicated. Output the tuples in hash table.
Partition Overflow. Hash table for $\pi_L^*(R_i)$ is larger than memory buffer. Recursively apply hash-based partitioning to overflowed partition.



Avoiding partition overflow. $B >$ size of hash table for $R_i = \frac{|\pi_L^*(R)|}{B_1} \times f$ approximately $B > \sqrt{f \times |\pi_L^*(R)|}$
Cost if no overflow. $|R| + |\pi_L^*(R)|$ for partioning. $|\pi_L^*(R)|$ for duplicate elimination.
Comparison with sort-based. 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.
Using Indexes. If theres an index whose search key (and any include columns) contains all wanted attributes, use index scan. If index is ordered and search key includes wanted attributes as a prefix, scan data entries in order and compare adjacent data entries for duplicates.
Join algorithm considerations. Types of join predicates (equality vs inequality), size of join, available buffer space and access methods. R is outer relation and S is inner then $R \bowtie_\theta S$
Tuple-based nested loop. For every tuple in R and for every tuple in S, if there is match in the tuple, output result.
Cost analysis. $|R| + ||R|| \times |S|$. Cost of scanning R then S.
Page-based nested loop. For every page in R and for every page in S do tuple-based nested loop.
Cost analysis. $|R| + |R| \times |S|$. Cost of scanning R then S.
Block nested loop. Assuming $|R| \leq |S|$, allocate 1 page for S, 1 page for output and remaining for R.
Algorithm. While scanning of R is not done, read the next $B-2$ pages of R into the buffer. For each page in S, read that page into the buffer. For each tuple of R that is in the tuple and for each tuple that was read from the page S, if there is a match, output result.
Cost analysis. $|R| + (\lceil \frac{|R|}{B-2} \rceil \times |S|)$
Index Nested Loop. Consider $R(A, B) \bowtie_A S(A, C)$. Suppose theres a B+ tree index on $S.A$. Use the tuple in R to probe the B+ tree to find matching.
Precondition. There is an index on the join attribute(s) of S.
Cost analysis. Assuming uniform distirbution

$$|R| + ||R|| \times \left(\log_F(\lceil \frac{||S||}{b_d} \rceil) + \lceil \frac{||S||}{b_d ||\pi_{B_j}(S)||} \rceil \right)$$