# Disk access time

**Command processing time:** interpreting access command by disk controller

Diagram of a structure with text

Description automatically generatedDiagram of a machine with text

Description automatically generated with medium confidence

**Write order:** write to fill a cylinder first before moving to the next cylinder

**Read order:** all blocks read in cylinder before next cylinder; all tracks on one side read first before the next side; all blocks read as contiguous sectors

**Blocks:** sequence of one or more contiguous sectors

**Seek time:** moving arms to position disk head on track

* Seek per cylinder (if sequentially stored)
* Average: 5 – 6 milliseconds

**Rotational delay:** waiting for block to rotate under head

* Incurred once if sequentially stored/read
* Depends on rotation speed (rotations per minute)
* Average: time for ½ revolution

**Transfer time:** moving data to/from disk surface

* N = number of requested sectors on same track
* N \* (time for one revolution / # of sectors per track)
* Applicable per track/sector/block
* Average: 100 – 200 microseconds

**Access time:** seek time + rotational delay + transfer time

**Response time:** queuing delay + access time

# Storage manager components

* Data stored & retrieved in units called disk blocks/pages

|  |  |
| --- | --- |
| **Files & access methods layer (file layer)** | Deals with organization and retrieval of data |
| **Buffer manager** | Controls reading/writing of disk pages |
| **Disk space manager** | Keeps track of pages used by file layer |

# Buffer manager

* Check if requested page in frame, if so, increment pin count and return address of f
  + Otherwise, use replacement policy to find victim page, increment count, replace page, and write to disk if victim page was dirty
* Coordinates with transaction manager for data correctness and recoverability

**Buffer pool:** main memory allocated for DBMS

* Partitioned into block-sized pages: frames
* Clients:
  + Request for disk page to be fetched into buffer pool
  + Release a disk page in buffer pool

**Frame information:**

* Pin count: number of clients using page (default 0)
* Dirty flag: if page is dirty (modified, not on disk) (default false)

**Pinning:** incrementing pin count of request page in frame

**Unpinning:** decrementing pin count

* Should update dirty flag to true if page is dirty

**Page replacement:** only if pin count is 0

# Buffer replacement policies

**Others:** random, first in first out (FIFO), most recently used (MRU)

**LRU:** uses a queue of pointers to frames with pin count = 0

* Sequential flooding: evicting all the time, same effect as having 1 buffer page

**Clock:** variant of LRU

* Current variable: points to a buffer frame
* Each frame has a referenced bit which is turned on when pin count is 0 (i.e. during unpin)
* Replace page with referenced bit off and pin count is 0
* Reference bit turned off once visited

A screenshot of a computer

Description automatically generated

Figure 2: Clock replacement policy

# Files

* Each relation is a file of records
* Each record has a unique record identifier (RID/TID)
* Each relation has a file, each file has many pages, each page has many records

**File ­­organization:** heap file (unordered), sorted file (ordered by search key), hashed file (located in blocks via hash function)

**A diagram of a data flow

Description automatically generated**

# Page formats

* Determines how records are organized within a page
* RID = (page id, slot number)

**Fixed length records:** may result in internal fragmentation, packed and unpacked organization (use bit map)

* **Packed organization:** costly delete operation matters, shift and move to fill in a gap

**Variable length records:** slotted page organization

* Divide single data pages into data space and free space
* Contains data and slot directory
* Slot directory points to entries in the data space, storing the byte offset of each entry and left address
* Slot directory: <entries> <# of slots> <free space start ptr>
  + Filled from both directions
* Space managed using compaction (to remove fragmentation)

# Record formats

* Determines how fields are organized within a record

**Fixed length records:** fields are stored consecutively

**Variable length records:** delimited by special symbol or stored with a prefix array of offsets (indicates the offset to beginning of a field)

# Index

* Data structure to speed up retrieval of data records based on search key

**Search key:** sequence of *k* data attributes, *k >= 1,* composite (k>1)

**Unique index:** search key is a candidate key

**Storage:** stored as a file (separate from original relation file) with records in an index referred to as data entries

* Simple index: index + pointer to record in relation in a linked list
  + Stores more records per page since record size smaller

**Types:**

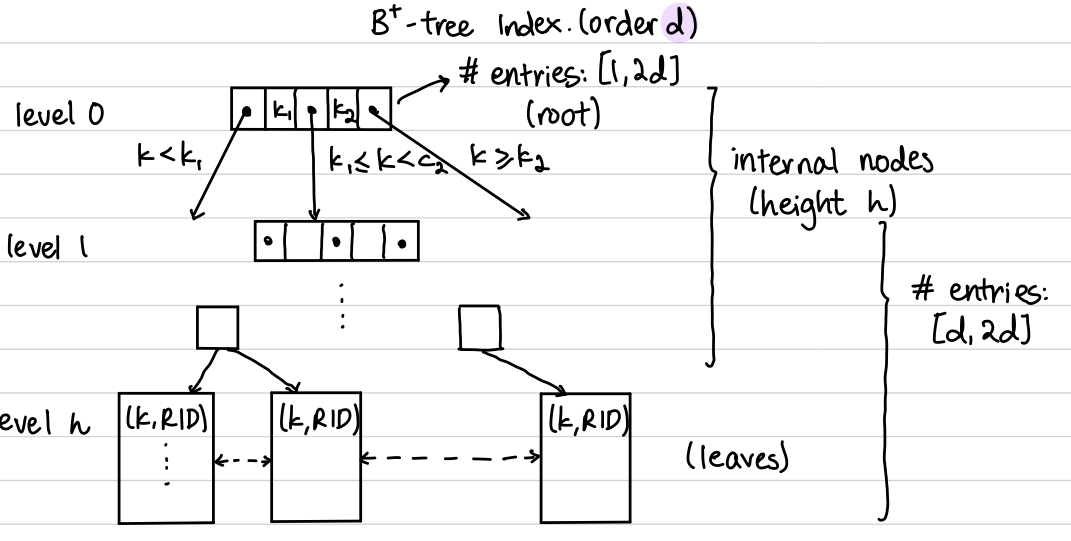
1. Tree-based index: based on sorting of search key value
2. Hash-based index: access data entries with hashing function

**Considerations:** search performance, storage overhead, update performance

* Types of search: equality (k = v), range (v1 <= k <= v2)
  + Hash-based index cannot support range search

# B+-tree index

* Every node is a data page/disc block
* Dynamic index structure so data updates update index



**Leaf nodes:** store data entries; doubly linked list

**Internal nodes:** store index entries (p0, k1, p1, k2, p2, …, pn)

* K1 < K2 < … < Kn
* *pi:* disk page address (root node of index subtree *Ti*)
* Index entry: (ki, pi), ki is separator between of p(i-1) and pi
  + p0 is not index entry

**Properties:** height balanced

* Order of index tree, *d*
  + Each non-root node: *m* entries, *m* in [d, 2d]
  + Root node: *n* entries, *n* in [1, 2d]

**Include columns:** including column data with leaf entries

**Minimum leaf nodes:** at level i, 2(d+1)^(i-1) (all lower bound)

**Maximum leaf nodes:** at level i, (2d+1)^i(all upper bound)

**Calculating order:** 2d(key size) + (2d+1)(index size)

**Duplicate handling:** use format 3 or external linked list or use composite key for index instead

**Data entry format:** what each leaf data entry stores

1. Format 1: k\* is actual data record
2. Format 2: k\* = (k, rid), rid is record identifier
3. Format 3: k\* = (k, rid-list), rid-list is list of record identifiers

**Overflow:** node contains 2d entries and a new entry is inserted

* Resolution: attempt re-distributing (only leaf) first, before splitting (both leaf and internal)

**Redistribution:** distributing entries from the overflowed node to a non-full adjacent sibling node (leaf only); try right first

* If sibling has < 2d elements, keep 2d entries in original node and move the first (right) or last (left) entry to sibling and update separator node
* For internal nodes, perform rotation (like underflow) but moving separator node into sibling internal node while pushing overflow to parent (become new separator node)

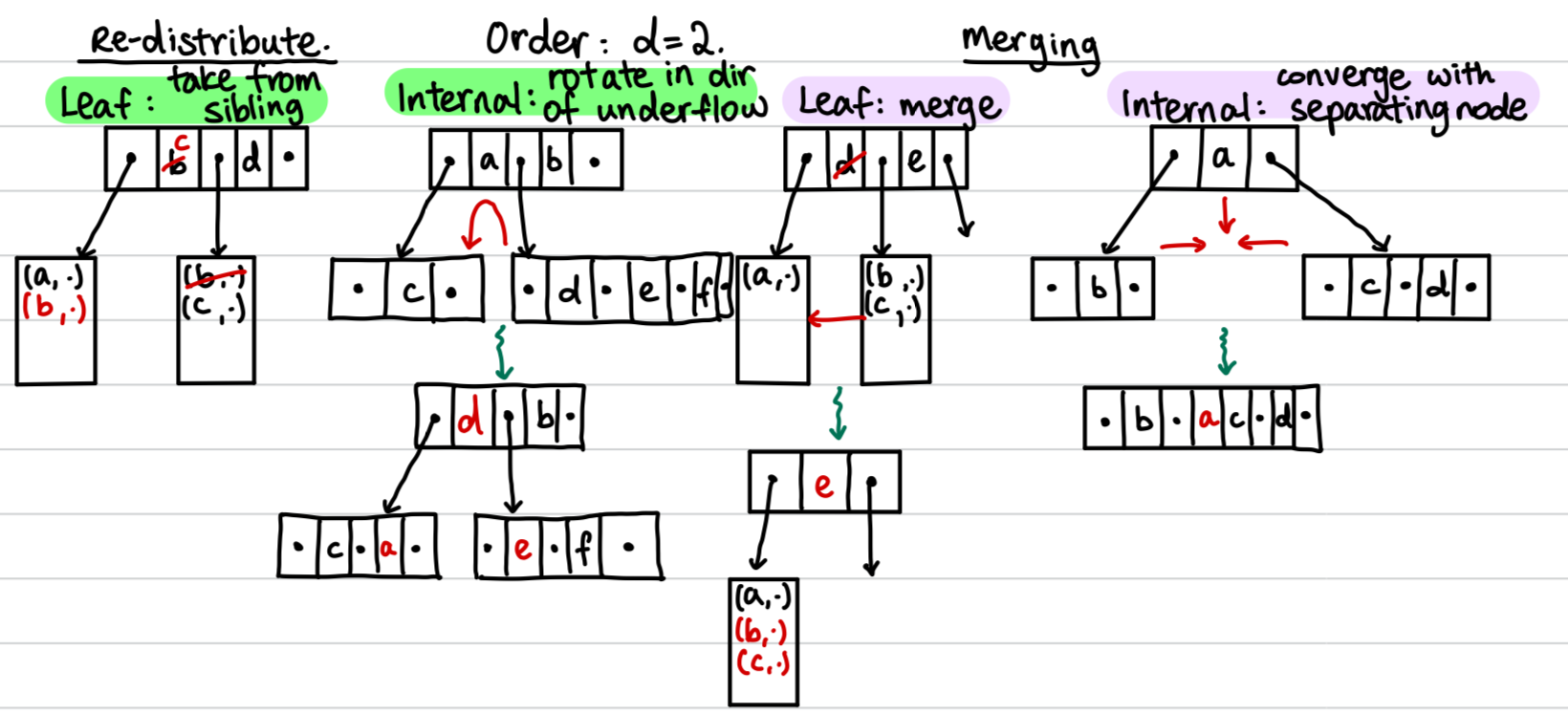
**Splitting:** split leaf node by distributing d+1 entries to new leaf node

* Push new leaf node as index entry
* If internal (parent) overflow, push middle key (d+1) to parent
  + The overflowed nodes both do not contain the middle key
* If root node overflows, create new root

**Underflowed node:** existing entry deleted from node with d entries

* Resolution: try re-distributing first (right first; leaf & internal), before merging (only leaf; target must be at most d + 1 entries)

**Delete operations:**



**Inserting into empty B+-tree:** start with single leaf/root node, on overflow, create new root and split

**Bulk loading:**

* Efficient construction, leaf pages allocated sequentially
* Only advantageous during loading, all disk info is separate

1. Sort data entries by search key
2. Load leaf pages with sorted entries (group into page size)
3. Initialize B+-tree with empty root
4. For each leaf page, insert its index entry into the rightmost parent-of-leaf level page
   * If parent overflows, move splitting key to new parent and continue

# Clustered vs unclustered index

* Order of its data entries is the same as or ‘close to’ the order of data records (in original file)
* Format 1 index is always clustered index
* Requires explicit cluster command & future entries may not be clustered
* At most 1 clustered index per relation
* Clustered: incurs less I/O as matching data entries are contiguous (so retrieving page j for data contains all other related records)
* Unclustered: incurs more I/O to retrieve pages that are not contiguous

# Dense vs sparse index

* There is an index record for every search key value in data
* Unclustered => dense
* Format 1 is always sparse, format 2/3 is always dense
* Format 1 treats only internal nodes as index records
* Format 2 treats leaf nodes as index records

# Linear hashing

* Dynamic hashing that grows hash file linearly by systematic splitting of buckets
* Overflow pages needed, overflowed bucket may not be split

**Overflow:** if all pages in Bi (primary + overflow) are full

**Split:** add bucket Bj (split image of Bi) and redistribute entries in Bi between Bi and Bj

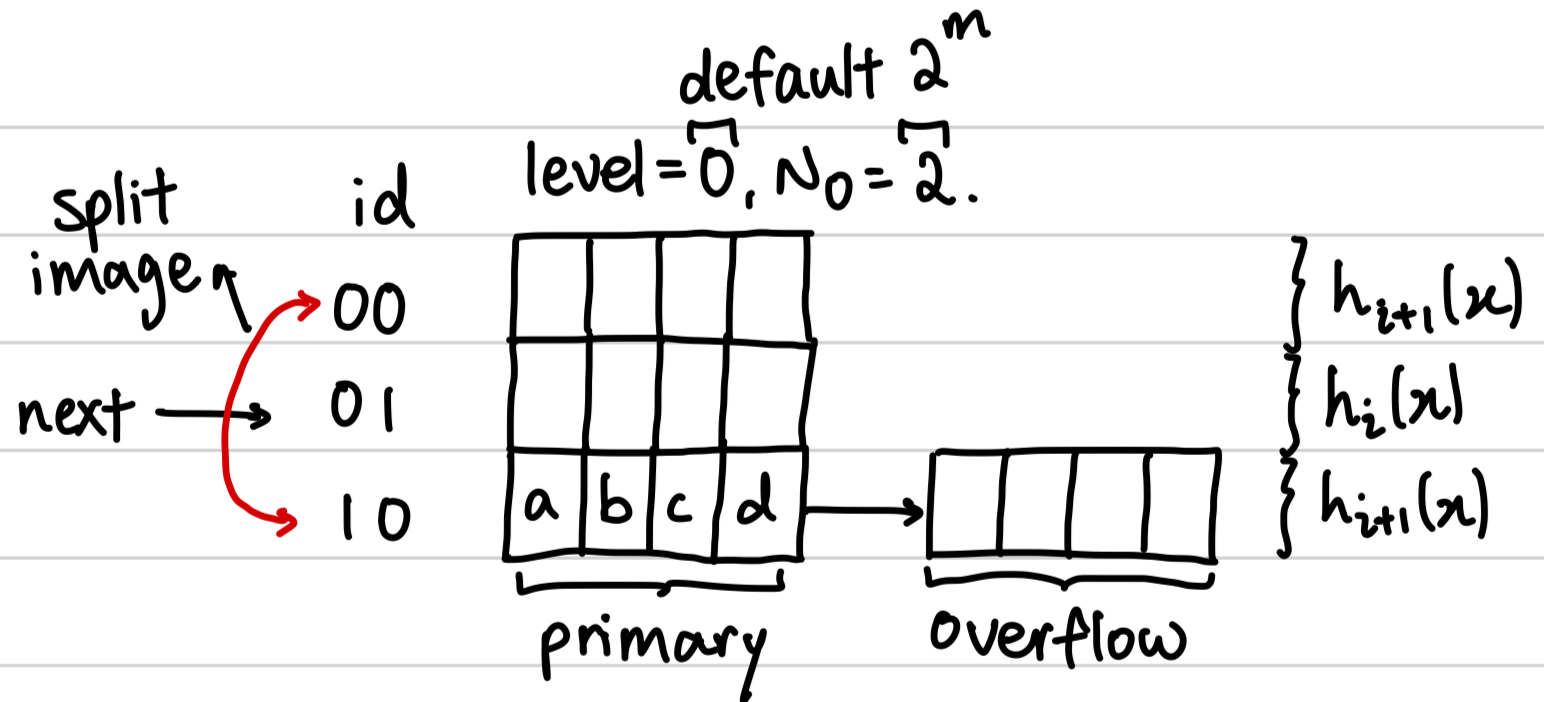
* Triggered when bucket overflows
* Split image of Bj is B(Ni + j) where Ni new buckets added
* After splitting, e remains iff last (m + i + 1) bit of h(k) is 0

|  |
| --- |
| 1. Redistribute entries of B\_next into B\_(next+Ni) using h\_(i+1) 2. Next += 1 3. If (next = Ni), level += 1; next = 0 |

**Round of splits:** file size doubles

* Round i (level), using 2 hash functions, h\_i and h\_i+1
* (typically taking the last (m + i) bits of h(k))

**File size at beginning of round i:**

****

**Insertion:** if there is space in bucket (given by *(m + i)* bits), add to bucket, otherwise, split the bucket at *next* and redistribute between *next* and its split image (using *(m + i + 1)* bits)

* Given that the split bucket may not be the bucket to insert, create overflow page if necessary
* If split bucket is target bucket, use *(m + i + 1)* bits to decide if it stays or goes to split image

**Deletion:** locate bucket and delete entry

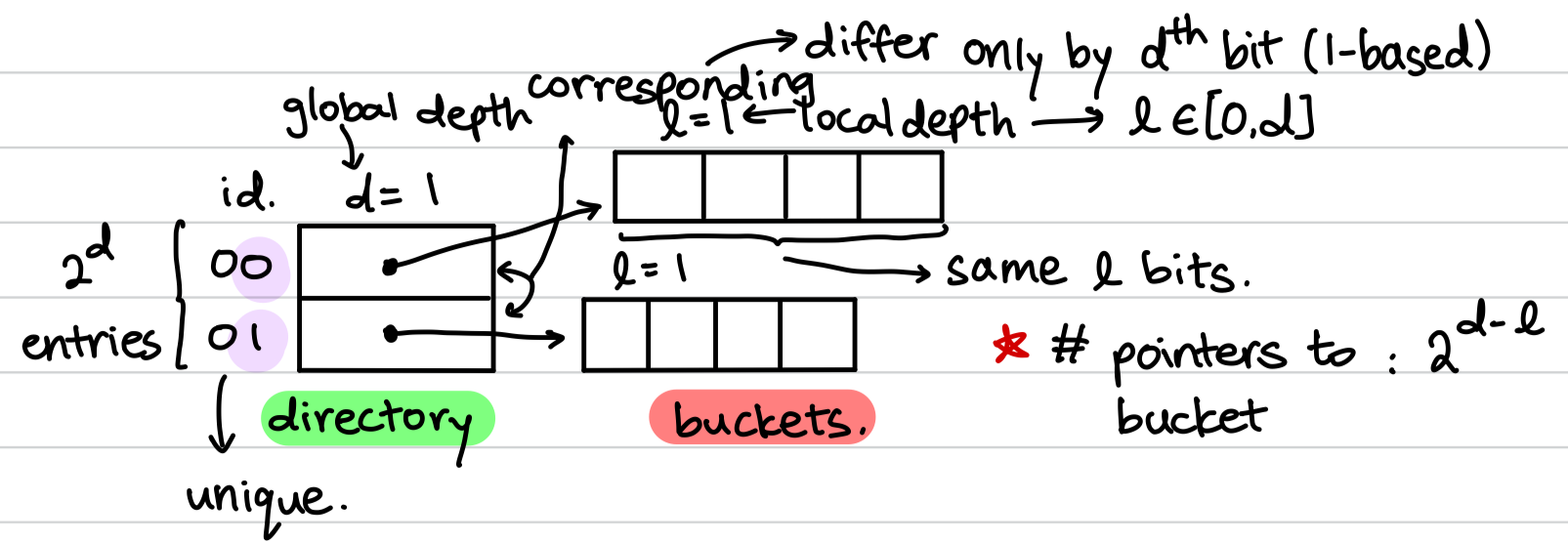
* If last bucket becomes empty, remove it
  + If next > 0, decrement by 1
  + If (next = 0) & (level > 0), update next to point to last bucket in previous round and decrement level by 1 (undoing a round)

**Performance:** one disk I/O unless overflow pages exists

* On average: 1.2 disk I/O for uniformly distributed data
* Worst case: I/O cost linear in the number of data entries
* Poor space utilization with skewed data distribution (one bucket storing everything)

# Extendible hashing

* Adds a bucket (split image) when bucket overflows
* Does not require as many overflow pages except when number of collisions exceed page capacity
* Directory entries are transitive so if inserting into bucket j and directory entry of j points to bucket k, then insert into bucket k first



**Insertion:** if there is space in bucket (given by *d* bits), insert entry into bucket, otherwise, increase local depth of target bucket, create split image with same local depth and re-distribute across original and split using *l* bits

* If local depth > global depth (after update), then double directory and re-distribute all entries by the local depth bits (*l* bits)
* Must maintain the invariant that the number of pointers to a given bucket is 2d-l

**Deletion:** locate Bi containing e and delete e

* Merge Bi and Bj if all entries fit in 1 bucket (use the original)
  + l -= 1
* If all corresponding entries point to same bucket, halve directory (all l < d)
  + d -= 1

**Performance:** at most 2 disk I/Os for equality selection

* At most 1 if directory first in main memory

**Last split bucket:** same local depth, last l – 1 bits same, l bit is 0 and 1, and total entries in both buckets more than page size

# External merge sort

* Given file of *N* pages with *B* buffer pages (B >= 3)
* Sorted runs are temporary tables

1. Creation of sorted runs (pass 0): read and sort B pages in memory
   * Creates sorted runs with each run being at most B pages
2. Subsequent passes: merge sorted runs using B-1 pages for input and 1 page for output; (B-1)-way merge

**Analysis:** number of I/O is 2 \* N read/write pages \* # of passes

**Blocked I/O:** reading/writing in units of buffer blocks of *b* pages

* Trade-off between maximizing the merge factor with reducing I/O cost
* Allocate *b* for output (no closed formula for *b*)
  + Use remaining for blocked input: with at most that many sorted runs each pass
* **Analysis:** *N* pages in file, *B* buffer pages, *b* block size
  + number of runs that can be merged per pass
  + total passes
* Given *j* input buffers each *m* size and *k* output pages and *s* seek time + rotational delay (typically m = k)

**Achieving k merge passes:** given N pages, buffer pages needed is

# Sorting using B+-tree

**Format 1:** scan leaf pages of B+ tree and return

**Format 2/3:** scan leaf pages and for each leaf page visited, retrieve data records using RIDs

# Selection

* Select rows from relation R that satisfy predicate p

# Access path

* Way of accessing data records/entries
  + Table scan, index scan, index intersection

**Selectivity:** number of index & data pages retrieved to access data records/entries

* Most selective => smallest selectivity => retrieve fewest pages

# Covering index

* Covering index I for query Q if all attributes referenced in Q are part of the key/include column(s) of I
  + Includes the SELECT, WHERE, GROUP BY, etc
* Q can be evaluated without RID lookups => index-only plans

# Index scan

* Find the leftmost element that satisfies the given query and traverse the leaf pages till the predicate is no longer true
* Works with either covering index or format 1 index

# Index scan + RID lookups

* Similar to index scan but for each element, perform RID lookups (incurring additional I/O cost)
* Works if non-covering index or format 2/3 index

# Index intersection

* Used when there is AND predicate
* Perform index scans for each predicate and find the intersection of all results
* Does not need to be same type of index

# CNF predicate

**Term:** form R.A op c or R.Ai op R.Aj

**Conjunct:** one or more terms connected via OR

**Disjunctive conjunct:** conjunct that contains OR

**Conjunctive Normal Form (CNF) predicate:** consist of one or more conjunct with AND

# Matching predicate: B+-tree

* Given index I = (K1, K2, …, Kn) and non-disjunctive CNF predicate p (i.e. cannot contain OR conditions)
* I matches p iff p is in form:
  + Follows index attribute list order, cannot skip attributes
  + Zero or more equality predicates (must follow attribute list)
  + At most 1 non-equality predicate (must be the last attribute of attribute list present, but can be multiple non-equality checks on the same attribute)
* If I matches p, use index scan since matching data entries will be in contiguous order

# Matching predicate: Hash index

* I matches p iff p is in form:
* Does not allow range based predicates

# Primary conjuncts

* Subset of conjuncts in p that I matches
* Can re-arrange to find the subsets that match

# Covered conjuncts

* All attributes in C in p in the key or include column(s) of I
  + Do not have to match
* Primary conjuncts are a subset of covered conjuncts

# Evaluating non-disjunctive conjuncts

1. Table scan: read straight from heap file
2. Hash index scan: use hash to read for information
3. B+ index scan: use B+ tree to read for information
4. Index intersection: combine results from index scans

* For OR predicates, must use covering index or index intersection, else use table scan

# Cost of B+-tree index evaluation

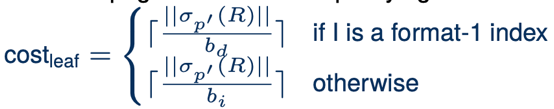
* p' = primary conjuncts of p, pc = covered conjuncts of p

1. Navigate internal nodes to locate first leaf: height of tree

A close-up of a number

Description automatically generated

1. Scan leaf pages to access all qualifying data entries: number of pages that contain qualifying data entries



1. Retrieve qualified data records via RID lookups: number of records

A close up of a sign

Description automatically generated

* Cost of RID lookups can be reduced with clustered data records (sorted data) since don’t need repeated I/O cost:
* If no primary conjunct, perform full leaf scan

# Cost of hash index evaluation

* May be more due to thrashing/collisions
* Format 1: at least (number of buckets)
* Format 2: at least and cost to retrieve data records:
  + Cost is 0 if I is a covering index, else
* Range-based queries default to table scan
* Maximum I/O with overflow is 1 + # of overflow pages

# Projection

* Project columns given by list L from relation R
* π(R): removes duplicates
* π\*(R): preserves duplicates (default of DBMS)

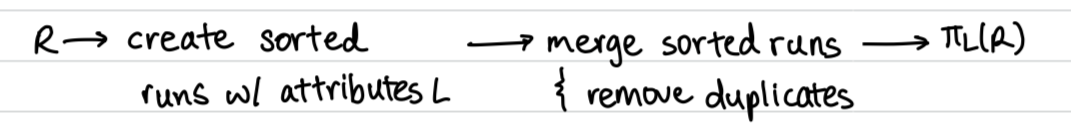
# Projection using sorting

* Results in sorted output
* Good if there are many duplicates or distribution of hashed values is non-uniform (increases chance to overflow)

A close-up of a note

Description automatically generated

**Optimization:** break up step 2 and merge with 1 and 3 respectively



* If , sort based performs similar to hash-based
  + initial sorted runs
  + merge passes

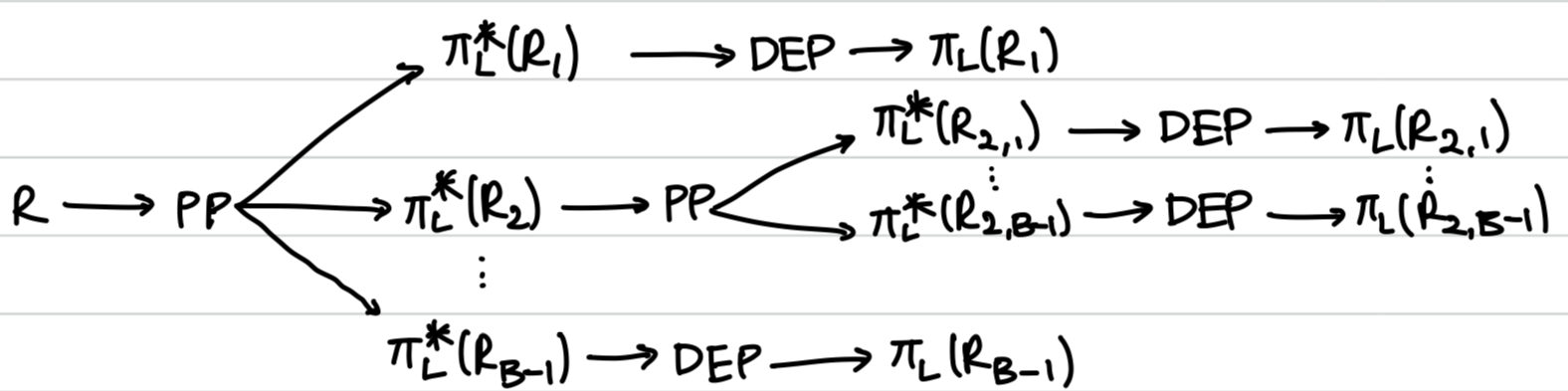
# Projection using hashing

A diagram of a diagram

Description automatically generated with medium confidence

**Partition overflow:** output hash tables too large than available memory buffer

* Recursively apply partitioning to overflowed partition



**Analysis:** effective if B is large relative to |R|

* B size: assume h distributes tuples in R uniformly
  + Approximately, B > (f is fudge factor)
* Assume no partition overflow: cost of partitioning + cost of duplicate elimination

# Projection using indexes

* Replace table scan with index scan iff there is an index with search key containing all projected attributes
* If index is ordered (like B+-tree) with projected attributes as prefix (no sorting needed)
  + Scan data entries in order
  + Compare adjacent data entries for duplicates
* If not all attributes included, create k partitions of distinct keys, sort individually, and merge (lower I/O cost overall)

# Join

A black text on a white background

Description automatically generated

**Considerations:**

1. Type of join predicate: equality, inequality
2. Size of join operands
3. Available buffer space
4. Available access methods

# Join algorithms

* Cost analysis ignores write & assumes R as outer relation
* Outer relation should always have less records
* Most optimal join: do it in-memory (|R| + |S|)

|  |  |
| --- | --- |
| **Type** | **Description/Cost** |
| Tuple-based | For each tuple r in R:  For each tuple s in S:  If (r matches s):  output (r, s) to result |
| |R| + ||R|| x |S| |
| Page-based | For each page Pr of R:  For each page Ps of S:  For each tuple of Pr:  For each tuple of Ps:  If (r matches s):  Output (r, s) to result  Brings page and uses it fully to provide more efficient I/O |
| |R| + |R| x |S|, min 3 pages |
| Block nested | While (scan of R is not done):  Read next (B - 2) pages of R into buffer  For each page Ps of S:  Read Ps into inner buffer  For each tuple r of R in outer buffer:  For each tuple s in Ps:  If (r matches s):  Output (r, s) to result  Allocate 1 page for inner, 1 page for output, remaining (B - 2) for outer |
|  |
| Index nested | For each tuple r in R:  Use r to probe S’s index to find matching tuples   * There exists an index on the join attributes of inner relation * Cost analysis involves uniform distribution assumption (every tuple in outer loop has the same number of matches in inner relation) * Cost analysis also assumes format 1 B+ tree index for S.A |
| |R| + ||R|| x J (cost to find matching entries)   * J is the height of the tree + search for leaf nodes |

# Notation

|  |  |
| --- | --- |
| r | Relational algebra expression |
| ||r|| | Number of tuples in output of r |
| |r| | Number of pages in output of r |
| bd | Number of data records that fit on page |
| bi | Number of data entries\* that fit on page |
| F | Average fanout of B+-tree index (i.e. # of pointers to child nodes) |
| h | Height of B+-tree index |
| B | Number of available buffer pages |

\* Data entries are (k, RID), data records are the full record from a relation

# Cost analysis notes

* If format 2, remember to include cost of RID lookups
* If unclustered, remember to duplicate cost of page lookup
* If clustered, cost of RID lookups can be simplified (to 1 per page)
* Think in terms of cost to read/write
* Blocked I/O -> reading/writing in blocks, 1 time I/O vs N time