File:
- Fixed length records (hard to delete, free space end at end of file)
- Variable length records (head stores #entries, EOFS, array of entry location / size, shift required)

Sequential file I/O: use overflow block, reorganize when physical / search key order gets lost
Database Buffer: pool of pages / blocks

- Replacement strategy: LRU, MRU, clock (doesn't' require maining usage info, bookkeeping only needed when page is replaced) [flip reference bit after clock hand hits] [if dirty, flush]
- vs OS: virtual memory manager cannot predict pattern of future accesses; DBs can determine blocks that are required in the short term future so it can use alternatives to LRU
- buffer frame dirty – have been modified, has to be flushed (written back to disk)
- buffer frame pinned – not allowed to be written back to disk
- OS: locking – for synchronization purposes, nobody else can access "lock" except for thread of execution.

Indexing:

- Advantage = faster; disadvantages = index itself might be large; space overhead , maintaining index sucks
    - Every index has to change when file gets modified, and seq. scan on secondary indices are I/O heavy
- Primary, secondary : index whose search key defines the sequential ordering of the file
- Dense, sparse: index appears for every search key, or only some of the search keys.
    - Sparse indices, less space overhead and insertions / deletions are easier. Usually use 1 index / block)

B+ tree: multilevel sparse index

- Insert / delete avoids file-reorganization cost since file doesn't have to be sequential
- If we have a secondary index on the structure, splitting nodes need file organization and require maintaining, so the secondary indices should reference p-key instead of the actual address.

Extendible hasing: b-bit hash prefix with bucket address table, increasing prefix while splitting

- Requires additional level of indirection (mod 2^global) to calculate where to go (done in memory)

Linear hashing: instead of splitting everything, only split the level the "this pointer is on"

- Doubles over the course of a round; more splits and uses collision table
- Avoid directory of local level; mod by 2^global only requires one extra indirection at most

Tradeoffs: access types, time, insertion / deletion frequency, space overhead

- Reorganization acceptable = use regular hash or index sequential. Otherwise, B+ tree / extendable / linear hash
- Types of queries
- Optimize average access time with worse worst case access time
- If insert / delete infrequent, or tight for space, use simple dense index.

I/O costs: B+ tree traversal = ceiling(log(ceiling[n/2](N)) max; extendible hashing = 1- N lookups

Query Optimization:

- create equivalent expression trees
- pay attention to join ordering
- estimate through statistics, numTuples, numBlocks, tupleSize, V(A, r) (distinct values for attribute A in r)
- use histograms, estimate selection size (/2, /10), join size, selection cost, aggregation cost
- use selection / projection to filter out asap, avoid nested sub queries (use join or make new table instead)

ACID:
Atomicity: if one part of the transaction fails, the entire transaction fails, and the database is unchanged. A committed database therefore appears atomic (indivisible).
Consistency: any transaction will bring the database from one valid state to another, and follow any constraints.
Isolation: concurrency control; every transaction executes independent of one another.
Durability: once a transaction has been committed, it will remain so.

Query Evaluation:

Selection:

- Linear search; ts + br * tt
- Tree; (hi + 1)(ts + tt)
- Secondary tree; (hi * (ts + tt) + b * tt
- Secondary nonkey; (hi + n) * (tt + ts)
    - Secondary index should only be used if very few records are selected.
- Conjunctive: insert RIDs, use composite index, or search linearly
- Disjunctive: union RIDs (requires each index to return this), or search linearly

Merge sort:

- Phase 1
    - Ceiling(br / M) runs; sort M blocks at a time, creating a run each
- Phase 2
    - logM-1(Br/M) runs Sort M-1 runs at a time, each pass decreases # of runs by factor of M-1
- Total: 2Br(ceiling(logM-1(ceiling(Br/M))))

Join: (r is always outer) *if one relation fits; put it inside and transfer = Br + Bs; seek = 2*

- Simple nested loop join: Br + Nr Bs transfers, Nr + Br seeks.
- Block nested loop join: Br + Br Bs (put smaller relation outside, unless it fits)
- Index nested loop join: Br(Tt + Ts) + Nr * c (smaller r_n outside, but the indexed one must be inside)
- Merge join: Br + Bs to Br * Bs (if they are all the same)
    - Start merging at last pass of sorting
    - Join on B+ tree RIDs
- Hash join:
    - Phase 1, hash into buckets R0 to RM-1, reduce size by factor of M-1
        - Repeat till largest bucket < M – 2; only one of the relations have to all fit
        - numPasses >= logM(Br) – 1; each pass 2Br + 2Bs
    - Phase 2, with one bucket fully in memory read in the other bucket and write if match
        - Read all Br and all of Bs
    - Total cost; (2Br +2Bs) (ceiling(logM(Br)) – 1) + Br + Bs
- Conjunctive: do theta1, use output as input
- Disjunctive: use union

Projection: *Duplicate elimination*: external sort merge, hash, hash join

Union, Intersection:

- sort / scan approach: Br + Bs + sorting
- hash join approach: union / intersect during probing

Aggregate:

- same cost as duplicate removal
    - add to a running total instead of remove (min / max / sum / count / avg)

Materialization: execute each operation, store results temporarily for next input. Cost = intermediate + writing

Pipelining: when a tuple of results are generated, pass it immediately to the next operation