CS 186 FINAL Cheatsheet

SQL:

- WHERE auto-filters NULL entries in col.
- SELECT <cols>
  FROM <table> AS <name>
  WHERE <predicate>
  GROUP BY <cols>
  HAVING <predicate>
  ORDER BY <cols>
  LIMIT <num>
- Only select grouped/aggregated cols after GROUP BY.
- WHERE <string col> LIKE "..." compares to regex string. ["%" = 0+ characters and "_" = 1 character]
- FROM A, B is cross product w/ join condition
- FROM A INNER JOIN B ON <predicate> [cross-join with join cond. in WHERE = inner join]
- A LEFT OUTER JOIN B ON <...> preserves A's cols. [similar RIGHT OUTER JOIN, FULL OUTER JOIN]
- A NATURAL JOIN B = inner join with implied condition on columns with same name

Subqueries

- SELECT *
  FROM classes
  WHERE EXISTS(
    SELECT *
    FROM enrollment
    WHERE classes.num = enrollment.num
  );
- WITH <name> AS (
    <query>
) ...

Disks & Files:

- Disk: R/W arm assembly on platters spinning 15k rpm. SSD: stationary, faster alternative.
- Heap file: Linked-list implementation: Header Page → Full pg 1 → Full pg 2 → ... full pages; Header Page → Free pg 1 → Free pg 2 → ... free pages [Remember to update header on insertions]

  Page directory implementation: Linked list of header pages. Header page = (Free space → Data, Free space → Data, full page → Data) [faster insertions]
- Sorted file: Page directory, with ordering enforced: Better for search (I/O: log N); Worse for insertion (I/O: log N + N; log search, N insert)
- Count Header I/O's when file type specified.
- Fixed Length Records: store fixed length data types (int, date, bool, etc.)
- Variable Length Records: store fixed first then variable length (varchar) with pointers to end of varchar. VLR in same schema can have diff length
- Packed FLR pages: records stacked, store pointer to end of last record.
- Unpacked FLR pages: slots stacked store bitmap of what slots are free.
- Varchar(X) can have 0-X bytes size
- VLR pages have footer (slot directory, slot count, free space ptr)
  - [ptr, record len]
  - Each VARCHAR has a 4-byte ptr.
  - Ptrs are int-size(offsets)
- max capacity of VLR page: $\frac{\text{page size} - 8}{\text{record size} + 8}$ [with int = 4 bytes]
- slot dir size: $4 + 4 + (4 + 4) \cdot \#\text{records} = 8 + 8r$ (slot count + free space ptr + (ptr + len))

B+ Trees

- Each node has $[d, 2d]$ entries (sorted) assuming no deletes (so $[d+1, 2d+1]$ child ptrs)
- If insertion overflow: split node into d, d+1 records. copy if leaf, else push.
- Only delete within leaf nodes.
- Alt 1) By value: leaf nodes contain records. (key, record) [No support for multiple index] [file must be sorted]
  Alt 2) By ref: leaf nodes point to records. (key, [page num, record num])
  Alt 3) By list of ref: list of ptrs with each key. (key, list of [" "]).
  [leaf nodes are data pages]
- Clustered: Data pages sorted same as in index. (Alt 1 has to be clustered, Alt 2/3 don't care)
- Bulk loading: fill leaf nodes upto 2fd. fill inner nodes completely.
- Leaf nodes contain nextLeaf ptr.

MAX LEAVES = $(2d+1)^h$
MAX RECORDS = $2d(2d+1)^h$ [$2d+1)^h$ data pages for Alt 1]

Buffer Management

- Frame Id: RAM Address of frame
- Page ID: MemoryAddr of Page in frame (set by requester of page i.e. file/index mgmt. system)
- Dirty Bit: modified?
- Pin #: number of threads using page at same time. (decremented by requestor of page)
- Least Recently Used: costly, sequential scanning means 0 hit rate (ABCDABCDABCD)
- Most Recently Used: good for sequential scan, bad for most uses.
- Clock Policy: approximates LRU
  - set clock to first unpinned frame on start. set ref bit to 1 when page is initially read in.
  - If hit, set ref bit to 1 without moving clock hand.
  - To evict: traverse cache until first ref 0 is found, setting 1's to 0 when passed. skip pinned pages. when ref bit 0 found, evict, replace, set to 1, move clock to next frame.
- Sequential flooding: when a sequential scan has 0 hit rate.

Relational Algebra:

- π [Select]
- − [set minus]
- × [cross product]
- ⋈<pred> [Inner Join]
- ⋈ [Natural Join]
- ∪ [Union]
- ∩ [intersection]
- σ [where for equality predicates with ∧, ∨]
- ρ_{a→b} [rename col a to b]

Hashing:

- Divide phase:
  - N input values, stream in using 1 input buffer, hash using h_1, stream output through B-1 output buffers, to B-1 partitions.
  - r_2 = sum of pages in partitions w/ >B pages each. Use h_2 to rehash. So on until all buffers <B.
- Conquer phase:
  - For each partition, use all B input buffers to load in, hash using h_r to re-order within partition, flush all B to memory.
- I/O cost: $\sum_{i=1}^{\#\text{ of passes}} (\text{Input pages for hash} + \text{Output pages for hash}) + 2(\text{Final output pages})$ (R, W)

Sorting:

- First sort each page, then pass 0: (conquer) use all B as input buffers and merge B pages at a time, creating sorted runs of size B. (in-memory)
- Pass 1+: merge B-1 runs at a time. use 1 as output buffer, B-1 as input. [the diff is bc merge is out-of-memory]
- I/Os: (merge) $\#\text{R\&W} \cdot 2N(1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$

Joins:

assume cost(S):
- [R] is pages in R, |R| is records in R
- Simple nested loop join: [R] + |R|[S]
- Block nested loop join: $[R] + \lceil \frac{[R]}{B-2} \rceil [S]$. (use B-2 input buff for R, 1 for S, 1 output)
- Index nested LJ: [R] + |R|(cost to look up records in S)

equijoin:
- Grace Hash Join: Partition R and S until $\min([Part(R)_i], [Part(S)_i]) \leq B-2$ $\forall i$. Then, $\forall i$, rehash smaller partition and naive hash join. (I/O: hashing + hashed [R]+[S]) (divide only) doesn't work with key skew.
- Sort Merge Join: $2[R](1+\lceil \log_{B-1} \lceil \frac{[R]}{B} \rceil \rceil) + 2[S](1+\lceil \log_{B-1} \lceil \frac{[S]}{B} \rceil \rceil) + [S] + [R]$ or [R]+|R|[S] in worst case
  - advance R or S pointers until match
  - mark the match in S, check further until non-match
  - go back to Mark in S, repeat advancement until match
  - Optimized SMJ: sort both R and S until $\max(\text{runs}(R), \text{runs}(S)) \leq B-1$.
    If runs(R) + runs(S) $\leq B-1$, then merge all in one pass. I/O: SMJ - 2[S] - 2[R]
    Else if $\max(\text{runs}(R), \text{runs}(S)) \leq B-2$, sort larger one, merge with runs of smaller. I/O: SMJ - 2max([R], [S])
    Else if $\min(\text{runs}(R), \text{runs}(S)) \leq B-2$, sort smaller... I/O: SMJ - 2min([R], [S]).
    Else SMJ.

Query Optimization:

- Streaming Operators: small cost to get next tuple (eg- SELECT)
- Blocking Operators: can't produce any output before consuming full input (eg- SORT)
- Selectivity estimation: what % of pages make it through operator
  - $X=a$: $\frac{1}{|\text{unique}(X)|}$
  - $X=Y$: $\frac{1}{\max(|\text{unique}(X)|, |\text{unique}(Y)|)}$
  - $X>a$: $\frac{\max(X)-a}{\max(X)-\min(X)+1}$
  - cond1 AND cond2: selectivity 1 * selectivity 2
- Join Selectivity: $\frac{1}{\max(|\text{unique}(A.id)|, |\text{unique}(B.id)|)}$ (A.id = B.id) · Output tuples (WHERE) = |A|·|B|·selectivity (SELECT).
- Heuristics for optimization: Push down selects (σ) and projects (π); only consider left-deep plans; cross joins only if no other option.
- System R (Sellinger Optimizer):
  - Pass 1: scan each table using full/index scan. advance optimal + interesting access plans. only left table can be result of join.
  - Pass 2-n: do same thing for joins and not scans. [only SMJ can produce interesting orders. SNLJ, INLJ preserve left relation order].
    interesting plans produce interesting order (sorted on col. used in SORT By, GROUP By, unevaluated join).
- Join I/O cost depends on:
  - whether intermediate relations (from prev. operator) are materialized or streamed. System R doesn't materialize.
  - interesting orders from prev. operators.

↓ for NLJ, only materialize right relation

Transactions & Concurrency

- Inconsistent Reads (Write-Read conflict): reads part of what was updated (eg- read R pre-update but S post-update)
- Lost Update (Write-Write conflict): Overrides someone else's update (eg- $p \leftarrow p+5$, $p \leftarrow 2p$ instead of $p \leftarrow 2p+5$)
- Dirty Reads (Write-Read conflict): reads an update that isn't committed.
- Unrepeatable Reads (Read-Write): transaction has multiple reads, but another transaction modifies record b/w reads.
- Atomicity: transaction either commits or aborts. Either way, either all changes happen or none.
- (A,D) ensured by recovery (ARIES) Consistency: if DB starts out consistent, it ends up consistent after Xaction.
- Isolation: each Xact should execute as if it ran by itself. No other Xact should affect it.
- Durability: if Xact commits, its changes persist despite failures.
- Concurrent execution increases throughput and reduces latency.
- Serial schedule = NO interleaved transactions.
- schedule serializable iff equiv. to serial schedule.
- Schedules are equivalent iff
  - same transactions
  - operations ordered the same within Xactions.
  - same resulting DB.
- Conflicts occur when: 2 operations from diff Xactions operate on same resource, and ≥1 of them is write.
- Conflict-equivalent: schedules that order conflicts the same. (d/w about seq. reads)
- conflict-equiv ⇒ equiv o/w conflict-serializable ⇒ serializable
- conflict serializable iff dependency graph acyclic.
- dependency graph
  - Xacts are nodes
  - edge from T_i to T_j if O_i and O_j conflict, and O_i appears earlier
- View-equivalent: same initial reads, same dependent reads, same winning writes
- View serializable finds all conflict-serializable schedules plus some serializable schedules w/ blind writes
- Blind writes: sequential writes with no interleaved reads.
- Two-Phase Locking (2PL): ensures conflict-serializable schedules.
  - Xact must acquire Shared lock before read, exclusive lock before write
  - Xact cannot acquire new locks after releasing any lock.
- 2PL is prone to cascading aborts (T_1 write + release, T_2 read, T_1 abort, T_2 forced to abort).
- Strict 2PL ensures all locks released together. (prevents cascading aborts).
- Lock Management

Resource	Granted	Mode	Queue
A	$\{T_1, T_2\}$	S	$T_3(X), T_4(X)$
B	T_6	X	$T_5(S), T_7(S)$

- Deadlock Avoidance: assign priority based on age.
  - Wait-Die: If ($T_i > T_j$) T_i waits : T_i aborts. [In this example, T_i wants what T_j has]
  - Wound-Wait: If ($T_i > T_j$) T_j aborts : T_i waits.

Deadlock is a cycle of Xacts waiting for locks to be released by each other.
Waits-for graph
- $T_i \rightarrow T_j$ iff T_j has or is ahead in queue for lock T_i wants!
- T_k will only wait for T_i on locks before T_j

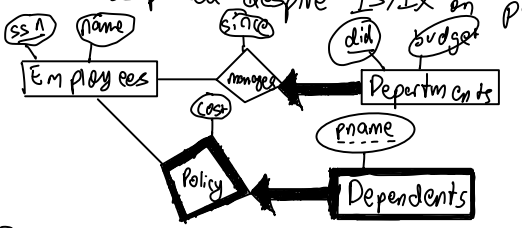
- IS: intent to place S on finer granularity
- IX: " " " X " " "
- SIX: S and IX together. S at this level, so no other Xact can get X on anything in that subtree.
- Transactions can't acquire new locks while waiting.

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	✗
IX	✓	✓	✗	✗	✗
S	✓	✗	✓	✗	✗
SIX	✓	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

- To get S/IS on node, hold IS/IX on parent.
- To get X/IX on node, hold IX/SIX on parent.
- Locks are released in bottom-up order.
- Queue skipping: when a lock is released, allow all compatible locks, checking in order of wait queue.
- X can be placed despite IS/IX on parent.

DB Design

- Entity-Relationship Models
  - Key constraints: arrows represent one-to-many (from 1 to many).
  - Participation constraints: thick lines represent at-least one.
  - Weak entities have partial key instead of primary key. They also have many-to-one with an 'owner' entity. owner's primary + weak's partial can uniquely identify each weak entity. (bolded)



- Functional Dependencies: $X \to Y$ implies X determines Y.
- X is superkey iff $X \to R$ (where R is full relation).
- candidate key is group of cols that form superkey (can't remove any cols and still form).
- Boyce-Codd Normal Form: R is in BCNF if $\forall X \to A \in F^+$, either $A \subseteq X$, or X is a superkey of R.
  - Input: R, F (FDs for R)
  - Algo: $\forall r \in R$ where r not BCNF:
    a) Pick $f: X \to A$; $X, A \in r$.
    b) Let $R_1 = X^+$, $R_2 = X \cup (r - X^+)$
    c) Remove r from R.
    d) Insert R_1, R_2 into R. (as indep tables)
    e) Recompute F as FDs over all $r \in R$.

($X \in R_1$ AND $X \in R_2$ ←)

- F^+ is the set of all FDs implied by F. It is the set of all table relationships in closure of F.
- Lossless decompositions are reversible. (eg- BCNF).
  - R to X, Y is lossless wrt F iff F^+ contains $X \cap Y \to X$ or $X \cap Y \to Y$ [$X \cap Y$ is the set of cols in both X and Y]
  - R to X, Y is dependency preserving wrt F iff $(F_X \cup F_Y)^+ = F^+$.

Recovery

(Logging CLRs is optional, for convenience)

- Force Policy: force all modified pages to disk before Xact commits.
- No-force Policy: only write back to disk when page needs to be evicted from buffer pool.
- Steal Policy: allow writing to disk before Xact finishes.
- No-steal Policy: pages can't be evicted from buffer pool until Xact commits.

Steal, no-force used fastest, undo+redo possible.

- Write-Ahead Logging: log records written to disk before data pages written. (for atomicity)
  " " " " " when Xact commits (for durability).
- Log records can be start, commit, Abort, or Update (<XID, pageID, offset, length, old_data, new_data>)
- Log Sequence Numbers:
  - prevLSN: stores LSN of last action from same Xact in log records
  - flushedLSN: stores LSN of last log record flushed to disk.
  - pageLSN: stores LSN of last log record where this page was modified.

[Page i can only be flushed if $pageLSN_i \leq$ flushedLSN. log flushed before page]

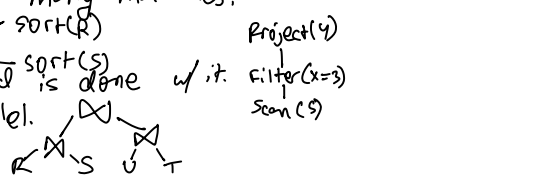
- Abort Process: Write ABORT record → undo actions bottom up and write Compensation Log Records for each.
- Transaction Table: X ID, status, lastLSN (most recent action from this Xact).
- Dirty Page Table: pageID, recLSN (first operation to dirty page).
- UPDATE records are written before dirty page. COMMIT records written after all pages written to disk.
- Undo Logging: after crash, scan log from bottom. (UPDATE log records represent pre-modification value)
  - for COMMIT/ABORT T: mark T as completed
  - for UPATE T, X, v: if (T not completed) write X=v to disk: ignore [undo incomplete Xacts]
  - for START T: ignore.
- Redo Logging: no-force, no-steal [UPDATE log records rep. post modification values]
  (redo updates of committed Xacts) → COMMIT records written before dirty pages flushed

Redo Phase
All updates with LSN > pageLSN are redone

- ARIES recovery Algo: Analysis → Redo → Undo
- Analysis Phase: to rebuild DPT and Xact table. scan logs from beginning and:
  - if not END: add to Xact table, set LastLSN.
  - if COMMIT/ABORT: update status in Xact table.
  - if UPDATE: if page not in DPT, add and set RecLSN.
  - if END: remove from Xact table.
  - after analysis phase, for any committing Xacts, add END log record, and remove from Xact table.
- Checkpointing writes Xact table and DPT to log to optimize recovery.
  - fuzzy checkpointing has <Begin_checkpoint> and <end_checkpoint> logs, and tables can be of any time b/w these. Thus, start recovery at begin_checkpoint.
- Then, Redo all Update/CLR operations where: Page in DPT, $RecLSN \leq LSN$, $pageLSN < LSN$. (regardless of commit)
- CLR records have undoNextLSN = prevLSN of operation being undone.

Parallel Query Processing

- Shared memory: every CPU shares memory + Disk
- Shared disk: shared disk, individual RAMs
- Shared nothing: (used in rest of note) CPUs only communicate via messages.
- Intra Query Parallelism.
  - Intra-Operator: divide data up and run one operator across many machines.
  - Inter-Operator: different op. diff. machines. eg. SMJ(R,S) — sort(R), sort(S) (sort(S) is done w/ it)
    - Pipeline parallelism: record passed up to parent as soon as child is done
    - Bushy tree parallelism: different branches of operator run parallel.



- Partitioning:
  - Sharding: each data page on 1 machine
  - replication: same page on many machines
  - range partitioning: each machine gets a range to store.
  - hash partitioning: self-explanatory.
  - round-robin partitioning: " (only benefit is uniform split ⇒ max parallelization)
- Parallel Sorting: Range Partition → local sort on each machine. I/O = $(1 + \lceil 1 + \log_{B-1}\lceil \frac{N}{mB} \rceil \rceil) 2N$ in best case. (depends on range dist.)
- Parallel hashing: Hash Partition → local hash on each machine. Range partition also works but hash easier.
- Parallel SMJ: Range Part. Reads using same ranges → local SMJ. I/O = $\underbrace{[R]+[S]}_{\text{Receive data on machines}} + \text{sort}(S) + \text{sort}(R) + \underbrace{[R]+[S]}_{\text{stream joined}}$
- Parallel GHJ: Hash " " " " hash fn → " GHJ.
- Broadcast join: send smaller relation to every machine and do local joins.
- All above joins need full input relations to produce output.
  Only pipeline-friendly join is symmetric hash join.
  - start with 2 empty hash tables.
  - when record from R arrives, probe S, generate outputs, add record to table for R. VV for S.
- Hierarchical Aggregation: Parallelized SUM, COUNT, AVG, etc.

Distributed Transactions

- Every machine has its own lock table.
- Superimpose all waits-for graphs to check for deadlock.
- 2 Phase Commit: need consensus to commit.
  - Prepare phase:
    - coord —prepare→ participants
    - participants generate PREPARE/ABORT record.
    - vote yes if PREPARE else no.
  - Commit/Abort phase:
    - coord —commit/abort→ part. [Based on yes/no votes]
    - " " COMMIT/ " "
    - send ACK (acknowledgement) back to coord.
    - coord generates END record after receiving all ACKs. (all records must be flushed before any network messages)

(logs and then)

  - Recovery:
    - Participant sees no PREPARE log → aborts
    - " " PREPARE log → ask coord. to repeat COMMIT/ABORT decision.
    - Coord " no COMMIT log → aborts
    - " " COMMIT log → resend commit message to part.s.
    - Part. " " " → resend ACK
    - Coord " END " → no recovery to do.
- Presumed Abort: Optimization for 2PC where no log record means abort. Hence, abort records never have to be flushed.
- NO votes occur when part. fails, avoids deadlock, etc.
- Even with another Xact of higher priority, YES votes must be honored. If vote yes and coord. asks to commit, must commit.

NoSQL

- OnLine Transaction Processing: Transactions↑ Users↑, many updates.
- OnLine Analytical Processing: Read only workloads, data↑, more joins/aggregation.
- Migration from OLTP to OLAP via Exact Transform Load.
- Consistency: multiple clients viewing should get same view of data.
  Availability: every request must receive response that isn't error. (don't stop/wait)
  Partition tolerance: continue operation despite dropped/delayed messages or disconnection.
  Brewer's CAP theorem: distributed systems can ensure 2/3 of these at most.
- 2PC sacrifices availability.
  - Alternative: Eventual consistency (once communication is good, consistency ensured).

- NoSQL Data Models
  - Key Value Store: key is a string/int identifier, value is byte-array of record. (eg- AWS DynamoDB) only supports get(key) and put(key, value).
  - Wide-Column: key = (rowID, columnID). value = record. (eg- Cassandra) supports get(key, [columns]), put(key, value).
  - Document: KVS with values that are semi-structured (eg- JSON/XML)
    [Relational DB stores tuples in tables.
    Document stores store documents in collections]
- JSON supports Arrays, Atomics (string/boolean/null/number (64-bit float)), objects (dict of k-v pairs)
  - flexible - schema non-enforcing - text-based not binary-based representation
- Relational DBs use binary data formats that are strict, incompatible w/ each other. JSON is optimized to be shared by different apps and languages, not for storage and retrieval from disk.
- MongoDB: get good.

MapReduce [for selection, group by, summation, etc.]

- Distributed File Systems partition data into chunks (usually 64 MB) and distribute/replicate it in different nodes for fault tolerance.
- func map(key1, value1) → bag(key2, value2)). [bag is a collection/array]
- func reduce(key2, bag(value2)) → bag(key3, value3)). [works on output of map].
  - leader splits M partitions (by key1) and assigns M workers.
    ↳ workers write map output to R regions (by key2). (written to disk for fault tolerance)
    ↳ leader then assigns R workers, who write reduce output to disk.
  - straggler is a machine taking unusually long on task.
  - MapReduce pre-emptively starts new backup executions of last few remaining tasks, and writes whichever finishes first, killing rest.

Spark

- Uses semi-structured data objects called Resilient Distributed Datasets (immutable)
- Intermediate RDDs are not flushed to disk unless .persist() is called.
- Lineage (sequence of operations to construct RDDs) is logged and flushed.
- On failure, all non-persisted RDDs are recomputed from lineage.
- actions are eager: they are immediately executed (count, reduce, save, etc.)
- transformations are lazy: logged and built into optimization tree. Executed on collect.

A Partial List of Spark Transformations and Actions

Transformations:	
map(f : T -> U):	RDD<T> -> RDD<U>
flatMap(f: T -> Seq(U)):	RDD<T> -> RDD<U>
filter(f:T->Bool):	RDD<T> -> RDD<T>
groupByKey():	RDD<(K,V)> -> RDD<(K,Seq[V])>
reduceByKey(F:(V,V)-> V):	RDD<(K,V)> -> RDD<(K,V)>
union():	(RDD<T>,RDD<T>) -> RDD<T>
join():	(RDD<(K,V)>,RDD<(K,W)>) -> RDD<(K,(V,W))>
cogroup():	(RDD<(K,V)>,RDD<(K,W)>)-> RDD<(K,(Seq<V>,Seq<W>))>
crossProduct():	(RDD<T>,RDD<U>) -> RDD<(T,U)>

Actions:	
count():	RDD<T> -> Long
collect():	RDD<T> -> Seq<T>
reduce(f:(T,T)->T):	RDD<T> -> T
save(path:String):	Outputs RDD to a storage system e.g., HDFS

- Spark 2.0
  - Dataframes have rows, cols like relations. (support SQL-like API funcs such as group by)
  - Datasets have typed objects instead of rows.

k-D trees

- An example of Spatial Indexing
- Binary search tree split on k different cols.
- construct(Data)
  - if |Data| = R: return new Block(Data)
    else: divide Data by thresholding on some col
    return (col, threshold)
    / construct(Data$_1$) \ construct(Data$_2$).

- B = # Blocks, R = # Records per Block
- Equality Search = Binary search $[\log_2(BR)]$ I/Os
- Insertion: log(BR) + 1 + 1 + 2 I/O
  find page to mod. Read page to mod. write modded page, update index.

	Heap File	Sorted File	Clustered Index	K-d Trees
Scan all records	B*D	B*D	3/2 * B * D	B * D
Equality Search	0.5*B*D	$(\log_2 B)$*D	$(\log_F (BR/E)+2)$*D	$(\log_2 (BR) + 2)$ * D
Range Search	B*D	$((\log_2 B)+pages)$*D	$(\log_F (BR/E)+3*pages)$*D	$(\log_2 (BR)+2*\#pages)$ * D
Insert	2*D	$((\log_2 B) + B)$*D	$(\log_F (BR/E)+4)$*D	$(\log_2 (BR) + 4)$ * D
Delete	(0.5*B+1)*D	$((\log_2 B) + B)$*D	$(\log_F (BR/E)+4)$*D	$(\log_2 (BR) + 4)$ * D

- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block
- **F:** Average internal node fanout
- **E:** Average # data entries per leaf

Eventual Consistency

BASE Semantics

- Eventually consistent systems opt for the 3 BASE guarantees instead of ACID guarantees:
  1. **Basic Availability**: reads and writes are available as much as possible; **however**, they are not guaranteed to be consistent (i.e. a read may not get the latest data and a write might not persist after all updates propagate). This is up to the application to fix.
  2. **Soft State**: a database can change even without inputs (e.g. as updates propagate), so the application only has a probability of knowing its state.
  3. **Eventually Consistent**: given enough time (after all updates propagate) all reads will be consistent.

Nearest Neighbor search: (for some point q)
- traverse tree down to find leaf node for q.
- current best estimate is best est., $\hat{q}$, in that block.
- calculate distance(q, threshold) for all boundaries of the partition. traverse all neighbors s.t. dist < dist(q, $\hat{q}$)

Consensus (Strong Consistency)

- consensus Protocols satisfy
  - Termination: All non-faulty processes eventually decide on value.
  - Agreement: All processes decide same val.
  - Validity: Chosen value is proposed by some participant of protocol.
- Paxos
  - assumptions:
    - messages eventually arrive (no bound on delay)
    - messages not corrupted (failed machines just stop)
    - to tolerate F failures, 2F+1 replicas must exist (i.e. majority of machines must be non-faulty)

Paxos Process

- Election: leader is elected.
  - candidate machines generate ballots with ballot_id.
    ballot_ids must be unique, increasing eg- MachineID + counter
  - proposer sends PREPARE(ballot_id) to majority participants.
  - if participant votes no, it ignores PREPARE and sends nothing.
  - if " " yes, it sends PROMISE(ballot_id, (old_ballot_id, old_v)) only if sent ACCEPT for some ballot before
    this is a promise to ignore future ballots with smaller id.
- Bill: Proposer produces bill.
  - if proposer receives majority vote, it becomes leader.
  - leader sends PROPOSE(ballot_id, v) to same people it sent last PROPOSE to. (v = value)
    (if it received old_v earlier, it must re-propose v = old_v with highest corr. old_ballot_id.
  - if participant has responded to PROPOSE from higher ballot_id, they ignore this.
    else, can ACCEPT(ballot_id, v)
- Law: Bill signed into law.
  - if majority, leader adds (ballot_id, v) to log and flushes.
  - sends COMMIT(ballot_id, v) to all participants.

Miscellaneous

- CLRs are counted when computing LastLSN.
- All operations except COMMIT, END, ABORT are redone (in ARIES).
- Don't undo already CLR-ed operations in undo phase.