Files

- Unordered heap file: records have no coherent ordering
- Clustered heap file: page/records are grouped in some way
- Sorted file: records appear in sorted order by a column
- Index file: store pointers/records across diff files
- Page layout: header (# records, free space, next/prev pointer, bitmap/slot table)
- Fixed length packed: GET: constant time by calculating the offset, ADD: add to end of record list, DELETE: remove from list, repack if needed
- Fixed length, unpacked: include bitmap of free/occupied
  - ADD: iterate through bitmap looking for 1st free
  - GET: Access w/ page & slot #, DELETE: clear bit
- Variable length: use a slot directory w/ # records, pointer to free space, pointer to start/length of records
  - ADD: add record to free space, add to slot directory, update free space pointer
  - DELETE: null pointer to a record, GET: retrieve a record using page # & slot #
  - Typically store pointer to # of VL fields @ the beginning of a record

IO Cost: Heap/Sorted File

Heap: Full Scan — N (all records), Equality: worst case of N, Best case 1 ⇒ Ave = 0.5 N, Range search: N b/c need to go through all, Insert: 1 just need to add to given position, Delete: 0.5 N + 1 b/c need to find record(s) & write to page to remove it

Sorted: Full Scan: N (all records), Equality: worst case takes $\log_2(N)$ b/c performs binary search, Range: $\log_2(N)$ + (# pages) finding start then look onward, Insert: $\log_2 N + N$ to find right location to insert and shift remaining records to right (need to read/write those), Delete: $\log_2 N + N$ repeat records to right on ave N/2 records

B+ Tree Insert

- Insert data into leaf node, split into nodes w/ n, n+1 apiece, (left right)
- Recursively split inner nodes, pushing up the n+1th node until reaching the root
- Scan Cost: 3/2 N since only 2/3 full,
- Equality Search: $\log_F(BR/E)+2$ since BR/E = # leaf pages, F = fanout, need to access root node & find record (+2),
- Range Search: $\log_F(BR/E)+3 \cdot$ pages since need to read through 3/2 # pages & scan on leaf level
- Insert: $\log_F(BR/E)+4$ since need to find right location, read root, modify data page, +2 to write to index/heap page
- Delete: $(\log_F(BR/E)+4$ for same as Insert

B+ Bulkloading

- Keep filling up the leaf nodes until we have exceeded fill factor, then split w/ only 1 value in new leaf node. For inner nodes, split on traditional "middle" value
  - Such left subtree isn't touched again after being bulkloaded
  - Need to first sort data before inserting

Best/Worst Case IOs

- Heap U Best Insert: read header, read/write data = 3
- Heap U Worst Insert: read header, read N data, write 1 = N+2 if no data pages added
  - Read header, read N data, write final, read/write new page, write prev final data ⇒ N+4
- Heap Page Directory Best: Read header, R/W data, write header = 4 IOs
- Heap File Page Dir Insert Worst: Read all headers, find on last page, write to last, write to header to include pointer to last ⇒ # headers + 3
- Heap PD Worst Full Scan: # header + # data
- Alt 2 Index Scan WC: Height + # leaf nodes + 3 · (# leaves · 2d) where # leaf nodes = $(2d+1)^{\text{height}}$
- Insert into Index WC: Read 1 node on each level, Write 2 new nodes each level, write 1 new level = 3H+1
- B+ Insert, Alt 3, Unclustered WC: Read # height nodes to reach leaf, read # records w/ same key, 1 record to data, 1 to leaf = Height + # Records + 2 IOs
- Heap PD WC Delete: Read all header, read all data pages, write all data, write all header = 2(H+D)
- Heap Select on Primary Key BC: Read 1st HP, find entry, read that data page ⇒ 2 IOs
- Heap PD Range Search BC: (HP + DP) IOs where DP = min num to store N records & HP = min num to store pointers to DP data pages

Additional SQL Notes

- **Subqueries**: Select * From (Select col1 from t1) as temp where col1 = 'val';
- **Distinct**: Select Count(Distinct S.name) vs. Select Distinct(Count(S.name))
  - Option 1 will provide unique IDs, sumhow manny of them there are
  - Option 2: provides a single row (the count) --> counts # of rows of the summed version of the table (which will be one)
- **String comparison syntax**: S.col LIKE 'B_%' or S.col ~ '^B.*'
  - _: exactly 1 char, %: matches any # of chars
- **Correlated Queries**: Select S.col1 From S Where Exists (Select * From R Where R.ID = 1 AND R.ID = S.ID) using some property of the outer query within the inner query
- **Common Table Expression (CTE)**: create a "temporary" table to query
  - With R As (Select...) Select col1, col2 from B, R allows us first to create a temporary table R that can be joined w/ another table
  - With R1 As (Select...), R2 As (Select...) Select * From R1, R2
- **Null Logic**: N&T=N, N&F=F, N&N=N, N|T=T, N|F=N, N|N = N

Sorting / Hashing

- **External Merge Sort**: Pass 0 splits N records into ~N/B runs of size B apiece (last run is leftover). Then, merge B-1 runs together, shrinking # runs & inc size of runs by a factor of B-1 until all data is on a single merged run
  - Total IOs = $2N(1 + \log_{B-1}(N/B))$
- **External Hash**: Hash function partitions records across B-1 partitions (ideally uniformly) → keep recursively partitioning all partitions w/ size > B. Finally, rehash all these partitions (more time, writing to disk)
  - IO cost: read N pages, write $\lceil N/B-1 \rceil \cdot (B-1)$ pages during partition phase. Write N pages in final

SQL Notes

- **String Comparison**: S.col like 'B%' or S.col~'B.*' both finding strings starting w/ B followed by any chars
- **Union / Union All**: Union finds all unique elements b/w 2 tables (removes duplicates). Union all preserves duplicates
- **Intersect**: Get unique records existing in both tables.
- **Intersect All**: Minimize cardinality across overlaps
- **Except All**: Difference in cardinality, lower bound of 0
- **Exists**: Check if table has records
- **In**: Check if column(s) of given table match inner table
- **Any**: Check if column value adheres to the predicate against ≥1 entry in the nested table
- **All**: Adheres to predicate against all entries in table
- **Join Syntax**: T1 <Inner/Natural> <Left/Right/Full> <Outer> Join T2
  - Inner/natural only preserves records w/ matching column values in joined table
  - Left/Right/Outer: Records w/o match are still kept in new table w/ non-match cols as NULL
- **Null Values**: Not Null = Null, T&N = N, F&N = N, T|N = T, F|N = N

Relational Algebra

- **Projection**: $\pi_{<cols>}(Table)$ takes in 1 input table & preserves only columns selected
- **Selection**: $\sigma_{pred}(Table)$ preserves only rows in the table satisfying the predicate
- **Union**: (Table1 ∪ Table2) requiring same schema & does not keep duplicates
- **Set Difference**: Table1 − Table2 which again gets rid of duplicate entries
- **Intersection**: (Table1 ∩ Table2) for overlap
- **Cross Product**: (Table1 × Table2) finds all possible combos of records across the 2 tables
- **Join**: Table1 $\bowtie_{<cols\ to\ Join\ On>}$ Table2 matches cols in Table1 & Table2 when creating merged table
- **Group By**: $\gamma_{col(s), Aggregate}(Table)$ which groups by col(s) specified → then apply aggregate function

Counting IOs (B+ Tree)

1. Read from root to leaf (height IOs)
2. Read # data pages (DP IOs, accounting for clustering & type of index)
3. Write to each data page as needed (WPP IOs)
4. Update Index page to point to new DP (1 IO)

- Alt 1: Leaf itself contains data pages so do not need leaf & data pages separately

Join Costs

1. BNLJ = [R] + [R/(B-2)] · [S] where R is the outer query
2. INLJ = [R] + |R| · numMatches · Cost/Match
   - Unclustered: # matches = # records
   - Clustered: # matches = # pages
3. SMJ: Sort(R) + Sort(S) + Merge(R,S) using external sorts (counting # passes) & merge # of pages in [R]+[S]
4. GHJ: Partition(R) + Partition(S) + Conquer(R,S)
   - Partition R,S according to external hash
   - Keep going until size of R,S ≤ B-2, the number of buffer pages
   - During "conquer" — larger sent to memory to be probed by smaller table
   - Conquer phase takes # IOs = # pages in the partitioned R,S after partition phase

- BNLJ only one that doesn't need equijoin
- SMJ, INLJ can produce "interesting" orders

Searching for optimal plan

- Only want to consider left deep plans, meaning they can only join 1 table @ each level
  - At each level, keep only cheapest plan & any that may create "interesting" order
  - Order matters if used in "order by", "group by", or for a later join

- Weak entity: entity that uses primary key of another entity set to be uniquely defined
  - Partial keys should be underlined & bold arrow from weak entity to primary b/c each weak entity should be uniquely identified by owner

Logging (Write Ahead Logging)

- Logs serve this exact purpose: have some way of redoing/undoing operations & tracking when we have a need
- **Write Ahead Logging (WAL)**: abides by the rules of ensuring all log records are written to disk before Xact can commit & need to make sure all changes are properly logged before modifying disk
- Example: assume we have T1: Start, W(A), W(B), ... Commit
    - First, record that T1 is starting
    - Then, write (T1, A, 99, 88) to indicate the transaction, resource, prior value, and new value. Only after the log is written do we modify the value in the buffer pool
    - Do the same w/ B: (T1, B, 5, 10)--modify value in memory
    - Then, write the WAL to disk
    - After, commit --> change the values of A, B on disk

Undo Logging

- Log records contain: Start, Commit, Abort, <T, X, v>) w/ old value
    - Need only previous value b/c we only care about what to revert back to--not the new value it takes on
- Undo logging relies on steal, force policy since we need to make sure that the effects of committed transactions commit but we're fine with having to get rid of "junk" modifications
    - Every time we modify a resource, record previous value
- **Undo Algorithm**: go through Xacts in Xact table, starting at largest LSN in Xact table and working way back up through the log
    - **Check:** Xact is running & in Xact table, and is an update operation that doesn't already have CLR
    - Check met? Write record: CLR Undo <Xact>: LSN #. Also include prevLSN (most recent LSN that referenced this Xact) & undoNextLSN (in prevLSN)
    - If the prevLSN of a given record is null -> write end record

Redo Logging

- Idea: write logs <T, X, v> with the new value that gets set in addition to the <Commit T> record before flushing the new value
- Relies on no steal, no force policy since we don't want "junk" modifications to be committed (no way of backtracking) but we're fine with having to re-execute operations that didn't go through
- **Redo Algorithm**: Start at the smallest LSN in DPT. Work way through all update/CLR operations in log records using check condition
    - **Check**: page in DPT, recLSN <= LSN & DiskLSN < LSN

ARIES

- Uses **steal, no force** policy which forces us to use undo & redo logs
- Can flush pages whenever & don't need to flush on commit
- Each new log record contains: LSN, prevLSN, XactID, type
- Also need pageLSN, the LSN of last log record that modified page
- Stored on page, gets pushed to memory when page is changed. Gets flushed to disk when page does too
- Contains an **Xact table**: maps each active Xact (identified by XID) to latest log record of a Xact in table along w/ status of each transaction (whether it's running, aborting, or committing)
- **Dirty Page Table**: maps PageID to record LSN, which is the first page record that dirtied a page
- Has 3 phases: analysis, redo, undo
- **Analysis Phase**: used to reconstruct Xact/DP table. At the end of this phase, change all non-committed Xacts status abort. Start @ beginning of right after checkpoint:

1. Record is not an end record? Add the corresponding transaction to the transaction table if not already present. If present, update the lastLSN in the transactions table
2. Abort/commit record? Change status in transaction table
3. Record is update record? Add page to dirty page table, setting recLSN to the LSN of this record
4. Record is end record? Remove Xact from the transaction table

- **External Merge Sort**: Pass 0 splits N records into ~N/B runs of size B (last run is leftover). Pass 1 onward merges B-1 runs together, shrinking # runs & increasing size of runs by a factor of B-1 until all data is on single merged run
    - Total IO Cost = 2N(1+log_B-1(N/B))
- **External Hash**: Hash function partitions records across B-1 partitions, recurse until while partition size > B (divide). Rehash all partitions 1 more time to write to disk (conquer)
    - Total IO Cost = read N, write ceil(N/(B-1)) * (B-1) pages. Recurse. Write N pages in final

Selectivity

- **Col = val**: sel = 1/NKeys(Col)
- **Col1 = Col2**: sel = 1/max(nkeys(col1), nkeys(col2))
- **Col > val**: sel = (High-Val)/(High-Low + 1)
- **Default selectivity**: 1/10 (given insufficient information)
- **Sel(P1&P2...Pn)** = P(P1) * P(P2)... * P(Pn)
- **Sel(P1 U P2)** = P(P1)+P(P2) - P(P1)*P(P2)
- **Float queries**: Sel(Col in range) = (maxVal-minVal)/(globalMax-globalMin)

Joins

Joins

- Typically looking at inner/equi joins, to take advantage of the join algorithms we have
- Join: matching tuples of 2 different relations
- Join Condition: dictates how we match rows across tables
- Notation: [R] = # pages, pR: # records/page, |R| = # records
- Don't include final write in IO cost. INLJ, SMJ, GHJ only possible w/ equijoins

Simple Nested Loop Join (SNLJ)

- For every row in t1, iterate through every row in t2 --> match each row on the join condition to determine what to include in the output table
- **Cost**: [R] + |R| * [S] since we read in a page of R, then for every record on that page, read all the pages of S, repeating for each subsequent page in R

Page Nested Loop Join (PNLJ)

- Idea: read in a page of R, read in a page of S --> match that page in R against each page in S. Bring in new page of R, repeat the same process until we read all pages in R
- **IO Cost** = [R] + [R][S]

Block Nested Loop Join (BNLJ)

- BNLJ only uses 3 memory pages since we use 1 buffer to read R, 1 buffer to read S, feed output into 1 more buffer
- What if instead we fetched more pages of R to match against a given page of S? Maximum possible is B - 2 pages
- Algorithm: iterate through pages in R, iterate through pages in S, iterate through records on R page, iterate through records on S page --> matchy matchy
- **IO Cost** = [R] + ceil([R]/(B-2))*[S]
    - Justification: nee to read pages in R, then for each chunk of pages of R of size ([R]/(B-2)) we want to match against a given page in S

Index Nested Loop Join (INLJ)

- Theme: why go through all the records in the inner table when we could instead use an index to determine the relevant records to match against?
- Algorithm: iterate through outer table, iterate through rows that satisfy the predicate using an index --> join matches
- **IO Cost** = [R] + |R|*numMatches * (Cost of matching in S)
    - Cost of matching depends on type of index: Alt 1 = tree height + # relevant leaves, Alt 2/3 = tree height + # relevant pages (1 IO/record if unclustered, 1 IO/page if clustered)

Sort Merge Join (SMJ-interesting order)

- Idea: first sort the tables, then joins will be more efficient by matching the records of the sorted tables
- Generic Algorithm: move to next record in R while R record is too small; move to next record in S while S record is too small. Mark given record in S, match records in S against current R record. Then reset S to original mark, S.next(), R.next()
- **IO Cost** = Cost of R Sort + Cost of S Sort + ([R] + [S])

Grace Hash Join

- We first partition R and S into B - 1 partitions using the same hash function for both tables, build a hash table over a given partition of R --> find matching records by streaming in records of S, probing hash table & spitting out the matches
- **IO Cost** = Partition(R) + Partition(S) + Join(HashR, HashS). We keep hashing until one of the size of the records in the partition is <= B - 2
    - Keep reducing the size of each table by a factor of B - 1 until the size of the partitions of one of the table is <= B-2. Build & Probe: Size(R Partitions) + Size(S Partitions)

Steal/Force Policies

- Idea: when working w/ Xacts, we want to achieve 4 key properties: atomicity, concurrency, isolation, durability
- Steal/Force policies are ways of dealing w/ atomicity/durability
- **Steal**: uncommitted, changed pages can be flushed to disk which is faster but may require undoing operations
- **No Steal**: can't flush uncommitted pages to disk which ensures that atomicity property holds
- **Force**: upon commit, Xact forces modified pages to disk which ensures durability property is met
- **No Force**: Modiifed pages aren't necessarily forced to disk after a commit
- Quicker since don't need to flush as often, but then don't have guarantee of durability
- **Steal, No Force**: doesn't ensure atomicity or durability since changes can be flushed to disk pre-commit (not atomic) & no force means even upon committing changes aren't flushed pre-crash
- Steal, no force is preferred policy b/c it's the fastest but also leads to the need for undo/redo logging

ACID Properties

- **Atomicity**: all operations execute or none do
- **Consistency**: DB that starts consistent ends as such even after execution of a transaction
- **Isolation**: even if transactions are running concurrently, should pretend as though only one is
- **Durability**: changes of Xacts that commit persist
- **Wait Die**: Ti, Tj--Ti has higher priority & wants lock --> wait for Tj before obtaining the lock. Lower priority? abort
- **Wound Wait**: Ti has higher priority --> Tj is aborted. Ti has lower priority -> wait for Tj
- Wound Die/Wait: draw waits queue & Xact w/ locks held

Potpourri

- **View Serializable**: every initial read/dependent read, & winning writes (writes that have a lasting impact) are the same
  - Permits more schedules than conflict serializability but is harder to actually enforce
- Serializable -> View Serializable -> Conflict serializable
- **Latency**: given fixed resources, how long does it take to run a set of transactions
- **Throughput**: how much data can be processed by fixed resources in a given period of time
- **Redo** requires more memory since we need to wait for commit before flushing data pages but takes less time since we can flush pages pre-commit (don't have the bottleneck in terms of latency that **undo** suffers from)

Functional Dependencies

- **Candidate Key**: smallest set of columns that det a table
- **Superkey**: Set of columns that determine a table
- **Functional Dependency (**X->Y): means that every record with the same value of X has the same value of Y
- **Closure (F+**): Set of all attributes that are F or implied by it
- **Test for losslessness**: decomposing R into X, Y is lossless IFF condition holds: [(X∩Y)->X OR (X∩Y)->Y]
  - That is to say, if we can decompose a table in a way such that the intersecting attributes determines one of the tables (acts as a key), then this decomposition is considered lossless
- **Decomposition Algorithm**: Assume that there is some violation FD (A->B): 1. Find A+, 2. Set R1 = A+, R2 = A U (R-A+), 3. Find FDs of R1, R2 --> iterate
  - Stop only when all the broken down tables adhere to BCNF form or have 2 attributes (must be BCNF by default) since any non-trivial dependency would act as a check across the entirety of the table

Locks Table

- When building a locks-held table and lock queue, make sure that upgrade requests are put at the front of the queue
- Even if a read lock is granted, if there's an X request in the queue and we encounter a request for another S lock, the S lock needs to be put in the queue

Transactions/Concurrency

- **Conflict**: Operations are from different Xacts on same resource w/ >=1W
- **Conflict Serializable**: conflict equivalent to a serilizable schedule
- **Conflict Equivalent**: all conflicting operations ordered same relatively
- **Conflict Dependency Graph**: nodes for each Xact w/ edge from Ti->Tj if Ti has conflict w/ Tj and Ti appears first
  - Conflict serializability is achieved if the conflict dependency graph has no cycles
  - Circle all Xacts on same resource: anytime there is at least one W on different resources draw an arrow from lower time to higher time
- **Wait For Graph**: Edge Ti->Tj if Tj has lock on X that Ti tries to get lock for but needs Tj to release the lock for first
  - Deadlock is present if there is a cycle in the waits for graph
  - To create graph: track locks held by each Xact, make a wait list for each resource by transaction & type of lock, drawing edge to Xact w/ the desired resource
- **Two Phase Locking (2PL)**: Xacts cannot get any new locks after releasing any look (fails to prevent cascading aborts), but is conflict serializable
- **Strict 2PL**: 2PL that releases every lock at the very end of an Xact

2 Phase Commit

- **Phase 1**: C sends prepare message, P creates & flushes either prepare/abort record (keeping C ID), P sends vote to C, C creates & flushes either commit/abort record (retaining P ID)
- **Phase 2**: C sends commit/abort verdict back to all P, P create & flush either commit/abort record, P sends ACK back to C, C creates & flushes end record

2 Phase Commit (Presumed Abort)

- **Presumed Abort**: Xact aborts if there's no log message
  - Xact abort: C removes Xacts w/o ACKs from table, P that receive abort verdict don't send ACK back, C checks if Xact in table to determine verdict, don't store P ID, don't flush abort records (no record? Assume abort)
- **New Phase 1**: C sends prepare, P creates prepare/abort record (only flush for prepare), P sends vote to C, C creates commit/abort record (only flush if comit)
- **New Phase 2**: C sends commit/abort result to Ps, P creates result log (flush if commit), P only sends ACK if commit, C only creates end record if commit

NOSQL

- NOSQL Properties: basic availability (application deals w/ partial failures), soft state (DB state subject to change w/o inputs), eventual consistency
- Supports get(key) & put(key, value) operations

MQL Syntax

- "table.col": used to get nested field/array w/in an object/document
- db.col.find(<pred>, <proj>) returns all documents filtering by the predicate
  - Ex: db.inventory.find{$or: [{status: 'D'}, {qty: {$lt: 30}} will keep all records that either have a status of D or have quantity < 30
  - Ex: db.inventory.find({"instock": {$elemMatch: {qty: {$gt: 10, $lte: 20}}}}) will find all entries that preserve: inventory.instock.qtt is w/in range: (10, 20]
  - db.t.find({}, {item: 1, _id: 0}) specifies which columns to keep/discard. Should use only 0's to discard or only 1's to keep except if want to exclude _id field
- db.table.aggregate: {$group: {_id: "$COL", newColName {$aggFunc: "$Col"}}}, {$match: colName: {colName: {$pred: val}}}, {$sort: {col: 1/-1}}])
- unwind: converts an array into separate entries for elements in the array

Mode	NL	IS	IX	S	SIX	X
NL	Yes	Yes	Yes	Yes	Yes	Yes
IS	Yes	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	Yes	No	No	No
S	Yes	Yes	No	Yes	No	No
SIX	Yes	Yes	No	No	No	No
X	Yes	No	No	No	No	No