SQL - Single Table Queries

[]= optional

SELECT [DISTINCT] <columns>
FROM <table>
[WHERE <predicate>]
[GROUP BY <columns>]
[HAVING <predicate>]
[ORDER BY <columns> [DESC/ASC]]
[LIMIT <amount>]

Logical Processing Order

1. FROM <table> - table we are drawing data from
2. WHERE <pred> - only keep rows where <pred> is satisfied
3. GROUP BY <columns> - group together rows by their values of <columns>
4. HAVING <pred> - only keep groups that satisfy <pred>
5. SELECT <columns> - select columns to keep
6. DISTINCT - keep only distinct rows
7. ORDER BY <columns> [DESC/ASC] - order output by value of <columns>, ASC by default
8. LIMIT <amount> - limit output to the first <amount> rows

Columns can only be selected if they are part of the GROUP BY clause or it's an aggregate (AVG, MIN, MAX, SUM, COUNT, etc.)

LIKE can be used for <pred>
_ : Any single character
% : zero or more characters

LIKE 'exp', LIKE '%2%', etc.

INNER JOINS

- Return rows where the ON condition is true

LEFT AND RIGHT JOINS

Each row in the left table appears, with any matching rows in the right table

Vice Versa for Right Joins

FULL JOINS

- Returns all rows from both tables, matching rows are combined, non matching rows appear w/ nulls.

SQL Tips

- No aggregates in WHERE
- Use HAVING only with GROUP BY, use WHERE otherwise
- If using GROUP BY, can only SELECT aggregate columns or columns in the GROUP BY.
- DISTINCT removes all duplicate rows

Page/Record Formats

- Tables are stored as logical files consisting of pages, each of which contains a collection of records.

Pages and I/Os

- Pages are managed
  - in memory by the buffer manager: higher levels of database only operate in memory
  - in disk by the disk space manager: reads and writes pages to physical disk/files
- The unit of accesses to physical disk is the page.
  - cannot fetch fractions of a page.
- I/O: unit of transferring a page of data between memory and disk (read or write 1 page = 1 I/O.

Data Types

Fixed or Var?	Data Type	Size
Fixed	Integer	4B
Fixed	float	4B
Fixed	boolean	1B
Fixed	char(20)	20B
Fixed	byte	1B
Variable	text	>=0B
Variable	varchar(20)	<=20B

1 bit for every field that can be null, rounded to bytes.

Ptr = 4 bytes

Fixed Length Records

- Records made up of multiple fields
- fixed length records are when field lengths are consistent

Ex: Students (sid integer PRIMARY KEY,
enrolled boolean,
grad_yr integer);

Record1 = 1234 | True | 2023

Storing FLRs

- packed: no gaps between records, records placed continuously one after the other
- unpacked: allows gaps between records, use a bitmap to keep track of where the gaps are.

Variable Length Records

- field lengths not consistent
- record lengths not fixed

Ex: Students (sid integer PRIMARY KEY,
name text, grad_yr integer);

Record1 = 1234 | Ben | 2023

Storing VLRs

- Each record contains a record header
- Variable length fields are placed after fixed length fields
- Record header stores field offset indicating where each variable length field ends.

Pages

Page header - portion of each page reserved to keep track of the records in the page.
- # records in page (4B)
- pointer to segment of free space in the page (4B)
- bitmap indicating which parts of the page are in use

Accessing Disk

seek time = 2-3 m/s
rot delay = 0-4 m/s
transfer time = 0.25 m/s

SSDs have limited erasures for writes

Locking

S lock - lets a transaction read a resource. Many transactions can hold S locks on a resource at once.

X lock - lets a transaction modify a resource. No other transactron can have any type of lock while a transaction has an X lock.

If conflicting lock, pause and wait until the other transactron releases the conflicting lock.

Deadlock is when a bunch of transactions are waiting on each other in a cycle

Deadlock avoidance: T_i wants a lock but T_2 holds a conflicting lock
wait-die: if T_i higher priority, T_i waits for T_2 to release lock.
if T_i lower priority, T_i aborts
wound-wait: if T_i higher priority, it causes T_2 to abort
if T_i lower priority, it waits for T_2 to finish

Priority by age: now - start time

Deadlock Detection: Build graph, node per transaction, if T_i holds a lock that conflicts with the lock that T_2 wants, add edge from T_2 to T_i. Cycle indicates deadlock. Abort one to end deadlock.

View Serializability

View Serializability $\subseteq$ Conflict Serializability $\subseteq$ Serializability.
- Conflict Serralizability but blind writes can be reordered.
- Blind writes: writes with no reads in between.

Checks: ① Same initial reads - same transaction reads the initial value of each resource.
② Same dependent reads - each read gets its value from the same transaction in both schedules. If T1 reads from T2's write in S1, it must do the same in S2.
③ Same winning writes - same transaction writes the final value of each resource.

Phantom Problem is when the same query returns different rows within the same transaction due to concurrent insert/delete.

VLRs: Slotted Pages
- move page header to be a footer to allow space for records to grow
- store length and pointer to start of each record in footer (4B each)
- Store #slots and ptr to free space

Deleting Records
- Unpacked Layout: nothing changes, but there is empty slot
  - can cause fragmentation
    - 2 free bytes in one place, 2 more else. Since slots are kept, cannot insert 4 byte record.
- Packed Layout: slot removed, record length, ptr pair removed, free space ptr updated, slot count decreases

File Organization
- Heap file is a file w/ no order enforced. Records are placed arbitrarily across pages
- Sorted file is sorted on a key (subset of the fields)

Implementing Heap Files
- Linked list: each page has 2 ptrs: free space and data. You have 2 linked lists of pages, both connected to a header page.
  - List of full pages
  - List of pages w/ free space.
- Page directory: linked list of header pages. Each entry of a header page contains a ptr to a data page, and amt of free space for the data page.

If implementation given, include I/O cost of reading/writing the file's header pages
2 I/Os per modified page

Indices
- DS that allows for fast lookup on a search key
- lookup can be equality, 1D, 2D, etc.
- search key is a subset of columns
- may store pointers to heap file (key, recordID) or entire records

B+ trees O(h) to find key
- Leaf nodes contain data entries (key, recordID)
- Inner nodes are only for lookup, left is <, right is >=
- d = order of tree
- Each node contains up to 2d values
- Height = # of pointers from root to a leaf node.
- Max fanout = 2d+1
- Occupancy Invariant: every inner node (except root) must be at least half full
- may also add sibling ptrs between leaf nodes

Insertion
1. Find correct leaf L
2. Put data entry into L
3. If enough space, done! Else, split into 2 nodes with d and d+1 entries.
4. Copy leftmost entry of right child to parent node and add references.
5. If inner node full, split in 2 and push middle val up

Max # data entries:
$\underbrace{2d}_{\text{node cap}} \cdot \underbrace{(2d+1)^h}_{\text{max fanout}}$

Bulkloading
When fill factor exceeded, add to parent node, and create new node and add there also. If inner node full (not fill factor), split and send up middle val. Adjust ptrs as needed.

2 Phase Locking (2PL)
- From start to until a lock is released, the transaction only acquires locks. Then until the end, it only releases locks.

Cascading aborts can occur if a transaction reads an uncommitted values which are later aborted. All transactions that directly or indirectly read its data must also abort.
Strict 2PL avoids this by only allowing locks to be released at the end of the transaction, when it commits.

Data Storage
Alt 1: store actual records
Alt 2: store <key, recordID>
Alt 3: store <key, list of matching recordIDs>
RecordID is page #, slot #
Actual record contains key

Clustering
- Clustered index: actual data is roughly sorted in order of search key
  - inner nodes correspond to leaf nodes on left and so on
  - 1 I/O per page of records
- Unclustered index: data not in order of search key
  - 1 I/O per record

Cost to full scan B+ tree = # data pages

Disk vs Memory
Disk - slow, lot of space
Memory - fast, limited space
Read/Writes - in memory
Use Disk for data pages not needed immediately

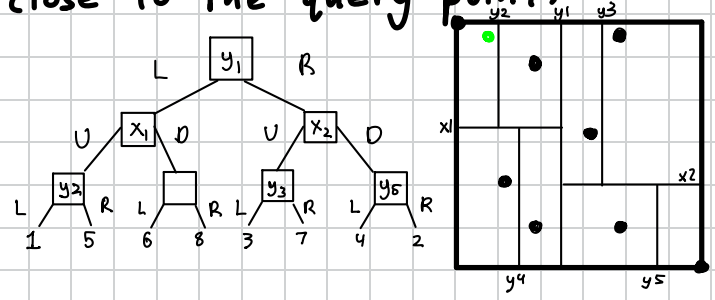
Buffer Manager
- manages which pages are loaded in memory
- controls when pages are read from and written to disk
  - decides pages to evict based on page replacement policy

Replacement Policies
LRU - evict least recently used pages, good when certain pages are popular.

Clock - arrange frames in circle. Pass in a circle, if ref bit is 1 set to 0, if its 0, you can evict. If its pinned just skip it. Do not move clock hand if hit.

MRU - evict most recently used page, good for sequential scans

Spatial Indexes O(log N)
K-d trees - used to perform nearest neighbor and range queries over high dimensional data. Partitions points into sep boxes. Nearest neighbor search involves searching boxes close to the query point.



External Algorithms
- designed for the case that there is more data than space in memory
- Can't access values whenever, disk accesses are expensive.
- Strategy: Divide and Conquer
  - Start w/ chunks that do fit in memory
  - Sort using External Merge Sort

General External Merge Sort
- Efficiently sort N pages with B buffer pages in memory
- Use a conquer and merge strategy w/ n passes: ceil(N/B) runs of B pages
  - Pass 1: sort more pages at once, so fewer runs to merge
    - can sort B pages at once
  - Pass 2-n: merge more runs at once, so finish faster
    - can merge B-1 runs at once → ceil(#runs from prev pass / B-1)

I/O Cost = $2N * (1 + \lceil \log_{B-1}(\lceil N/B \rceil) \rceil)$ (2nd term = total # passes; runs)

Hashing
- Want to be able to group together tuples with the same key value
- Partition the data with hash functions applied on the key
  - all tuples w/ a certain key will be in the same partition
- Useful for removing duplicates, grouping data

External Hashing
$3[R] + 2[R]$ for each (extra pass)
- Use a hash function h_p to partition the data.
- Write the partitions to disk
- Can split into B-1 partitions
- If the partitions are small enough to fit in memory ($\leq$ B pages), load them in and make an in-memory hash table for each one, one at a time.
  - Use a different hash function, h_r, that is independent of h_p.
- Assume perfect hash functions (equal partitions)
- Recursive partitioning keeps applying hash functions until partitions fit in memory
- All hash functions must be indep.
- Terminate algorithm if all values in partition are the same.
- At each pass, you write all the orig pages to disk + B-1 pages per partition.
- Partition is done when $\leq$ B pages while being read. Write it out and don't read it next time.

Iterators
- No support for random accesses
- pulls records one at a time
- Stream: records read one by one (filter)
- Materialize: Operator must see all input before producing output. (sort, hash)

Joins
- Take one relation and matching each tuple with tuples from another relation
- The join condition determines what rows in the other relation match to a row in the first relation
- Exclude the final write's IO cost, might not be materialized

Notation
$[R]$ = # of pages in R

p_R = # of records per page in R

$|R|$ = # of records in R (cardinality)
- $|R| = p_R * [R]$

Simple Nested Loop Join (SNLJ)
$R \bowtie_\theta S$:

```
for row r in R:
    for row s in S:
        if θ(r,s):
            output r joined w/ s
```

IO Cost = $[R] + |R| \cdot [S]$
- Flipping order can change IO cost.

Page Nested Loop Join (PNLJ)
$R \bowtie_\theta S$:

```
for page P_R in R:
    for page P_S in S:
        for row r in P_R:
            for row s in P_S:
                if θ(r,s):
                    output r join s
```

IO Cost = $[R] + [R][S]$

Block Nested Loop Join (BNLJ)
$R \bowtie_\theta S$:

```
for block of B-2 pages C_R in R:
    for page P_S in S:
        for row r in C_R:
            for row s in P_S:
                if θ(r,s):
                    output r join s
```

(before selectivity → after selectivity)

IO Cost = $[R] + \lceil [R]/(B-2) \rceil [S]$

Index Nested Loop Join (INLJ)
$R \bowtie_\theta S$: index on S

```
for row r in R:
    for row s in S that satisfies
    θ(r,s) (found using the index):
        output r join s
```

IO Cost = $[R] + |R|$ * cost to find matching S tuples

Cost to find matching S tuples:

Alt 1: Cost to traverse root to leaf + read all leaves w/ matching tuples

Alt 2/3: Same as Alt 1 + 1 IO per page if clustered, else, 1 IO per tuple

Sort Merge Join (SMJ)
1. External Sort R and S by the join key.
2. Iterator on R, Iterator on S.
   If R's key < S's key, advance R.
   If S's key < R's key, advance S.
   If keys are equal, output match
3. Mark first match s of any row r in R. Continue iterating as normal, when R advances, if the key of r is the same, reset S to the mark. Otherwise, release the mark.

IO Cost = sort R + sort S + [R] + [S]

Optimized: Stream sorted output into merge instead of materializing
- Saves 2[R] or 2[S] or 2([R]+[S])
- Only works w/o duplicates

IO Cost = sort R + sort S - [R] - [S]

Buffer req to stream both:
runs(R) + runs(S) $\leq$ B-1

Buffer req to stream one:
runs(R) $\leq$ B-2

Grace Hash Join
1. Hash both R and S using hash function h_b into B-1 partitions. Write all partitions to disk.
2. For each partition pair (R_i, S_i):
   Load R_i into memory and build an in memory hash table using h_r. Stream in tuples of S_i, probe the in memory hash table, output matching tuples.

Partitions of R must fit in B-2 pages. 1 to stream S partition, 1 for output.

If S is smaller do $S \bowtie_\theta R$. Also can recursively partition.

NLJs: null check if right.hasNext store rightRecord, compare to left, if 0, L.concat(r) else if left has more, advance left, reset right. else, return null

Relational Algebra
- always closed (always returns relation)

Unary Operators:

Projection (Π): $\Pi_{col1, col2}(R)$ returns R but with only col1 and col2 like a SELECT.
- If you project away the columns that 2 tuples differ by, they collapse into one tuple.

Selection (σ): $\sigma_{pred}(R)$ keeps only tuples from R that satisfy pred. → commutative
- same idea as WHERE

Renaming (ρ): $\rho_{id \to sid}(R)$ returns R but w/ the id attribute renamed to sid.
- SELECT id AS sid

Binary Operators:

Union ($\cup$): $R \cup S$ contains all tuples in R and all tuples in S. Only defined when both relations have the same attributes.
- if a tuple appears in both R and S, collapses into 1.
- same as UNION

Multi-granularity Locking
- allow for locking at diff levels (DB, table, page, tuple)

Intent Locks - To S lock tuple, IS DB, table, page first. To X lock tuple, IX DB, table, page first

An IX lock doesn't prevent placing S lock on a diff tuple, but an IS lock prevents you from placing an X lock on tuple.

SIX lock combines S and IX to scan table, update page, etc. Remove all S and IS locks that are descendants of the SIX lock.

Parent → Child	
NL	Nothing
S	Nothing
X	Nothing
IS	IS, S
IX	IX, X, IS, S, SIX
SIX	IX, X

Set Difference (—): R—S returns R but with any tuple that appears in S removed. Only defined when R and S have the same attributes
- EXCEPT

Cross Product (×): R×S returns a relation with a tuple (r,s) for every tuple r in R and every tuple s in S.
- Defined even when R and S have diff attributes $R \times (S \cup T) = (R \times S) \cup (R \times T)$

Compound Operators:

Intersection (∩): R∩S returns a relation that has only tuples appearing in both R and S.
- Only defined when R and S have the same attributes.
- Equal to R−(R−S)

Theta Join ($\bowtie_\theta$): $R \bowtie_\theta S$ returns the inner join of R and S on the join condition θ
- Equal to $\sigma_\theta(R \times S)$
- R INNER JOIN S ON θ

Natural Join (⋈): R⋈S returns the natural join of R and S, which is on every column with the same name.

- $\sigma_{R.col1 = S.col1 \ldots R.colN = S.colN}(R \times S)$

Group by / Aggregation (γ):
- $\gamma_{col, COUNT(*)>2}(R)$ returns R grouped by col, and filters groups that don't have COUNT(*) > 2
- $\gamma_{MAX(col)}(R)$ returns max over R on col
- $\gamma_{col, AVG(col2), COUNT(*)>2}(R)$ is the same as SELECT col, AVG(col2) FROM R GROUP BY col HAVING COUNT(*) > 2

Query Optimization

Naive plan: scan A, scan B, join, select

Alternate plans:
① Push selections/projections down the tree.
② Materialize intermediate relations (write to a temp file)
③ Use indices

Only SNLJ, INLJ, and SMJ preserve interesting orders.

BNLJ/PNLJ
If σ on left relation, pushing down helps, otherwise doesn't help.
If materialized, pay write cost once, but reduced size for future scans.

Total Cost = scan costs + materialize cost + join cost of pages that are fed.
Plan Space = Set of plans considered by optimizer
Cost estimation: IO Cost per plan
Search Algorithm - how to efficiently find best plan

Selectivity Estimation

Selectivity - fraction of tuples satisfying a predicate (0 to 1)
- default is 1/10
- Assume data is uniformly distributed across the range of values it takes on
- Assume predicates on different columns are independent of each other
  - Predicates on the same column are often dependent like x<10 and x>20.

Let |c| = # of distinct values in column c

If you have index on column, assume you know |c|, max(c), min(c)

output tuples/records = ⌊(selectivity)(total # of tuples)⌋

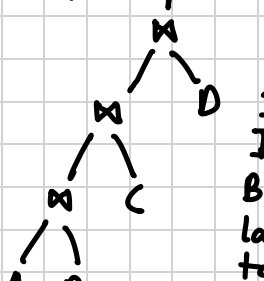
Pred	Selectivity
c=v	$1/\lvert c\rvert$ if you know |c|
c1=c2	$1/MAX(\lvert c1\rvert, \lvert c2\rvert)$ if you know both; $1/\lvert ci\rvert$ if you know one
c<v integer	$\frac{v - min(c)}{max(c) - min(c) + 1}$
c>v integer	$\frac{max(c) - v}{max(c) - min(c) + 1}$
c<=v integer	$\frac{v - min(c) + 1}{max(c) - min(c) + 1}$
c>=v integer	$\frac{max(c) - v + 1}{max(c) - min(c) + 1}$
c>=v float	$\frac{max(c) - v}{max(c) - min(c)}$
c<=v float	$\frac{v - min(c)}{max(c) - min(c)}$
p1 AND p2 independent	$S(p1) \cdot S(p2)$
p1 OR p2	$S(p1) + S(p2) - S(p1 \text{ AND } p2)$
NOT p	$1 - S(p)$

Build a graph with 1 node per transaction. If an operation in T_i conflicts w/ an operation in T_j, and the op in T_i comes first, add an edge from T_i to T_j. All conflict serializable schedules have an acyclic dependency graph.
Just check for cycles

Selinger Optimizer

Apply selectivity on leaf nodes

Plan Space: Only left deep trees, avoid joins with no join cond
Search Algorithm: Dynamic Prog.
Left Deep: ((A⋈B)⋈C)⋈D



Runtime: $O(n \cdot 2^n)$

Interesting Order: If query has ORDER BY, GROUP BY, or a JOIN later, might be worth to sort by that attribute now.

Drop plans if no join cond.

Pass 1: For each (relation, interesting order) pair, find min cost access method.
Drop plans if they don't have an interesting order AND it's not the cheapest of the relation

Pass i: For each (subset of size i of relations, interesting order), find min cost access method.
Drop plans if they don't have an interesting order AND it's not the cheapest method for the subset. Interesting orders expire, if interest comes from S.x=T.x but you are joining S&T

{A, B} = {B, A} but A BNLJ B and B BNLJ A might have diff costs.

Transactions

Collections of operations that are a single logical unit.
- We want all operations to happen at once.

Atomicity - All operations in a trans happen or none of them.
Consistency - Transaction must leave DB in a state that satisfies all rules of the DB
Isolation - If two transactions run concurrently, they shouldn't interfere w/ each other.
Durability - A committed transaction should persist.
Commit - successful transaction (save changes)
Abort - unsuccessful (revert changes)

Serializability

Schedule - order in which operations of a set of transactions are executed.
Serial Schedule - every transactions runs start to finish without interleaving.
Equivalent schedules involve same transactions, final state is the same, and each transaction has the same order of ops in both.
Isolation → schedule is equivalent to a serial schedule.

We check if schedules are conflict serializable.
Conflict if ① At least one operation is write
② diff transactions
③ work on same resource

Conflict Equivalent: Let S1 have conflict where $T_1 R(B)$ comes before $T_2 W(B)$. Then S_2 will also have $T_1 R(B)$ before $T_2 W(B)$. This must apply for all conflicts.
If conflict equivalent to serial schedule → conflict serializable

Query Parallelism

Inter-query parallelism works using queries as the unit of parallelism
- Ex: running 5 queries in parallel

Intra-query parallelism works within a single query
- Ex: Scanning 2 relations for a query in parallel.

Intra-query Parallelism

Inter-operator parallelism works using operators as the unit of parallelism
- Ex: Running 2 scans in parallel

Intra-operator parallelism works within a single operator
- Ex: Speeding up a single scan by having multiple threads or machines read different parts at the same time.

Inter-operator Parallelism

Pipeline parallelism is inter-operator parallelism on operators in a pipeline
- Ex: Projection applied on selection applied on scan: selection depends on output of scan, and projection depends on output of selection.

Bushy Tree parallelism is inter-operator parallelism on operators that don't depend on each other.
- Ex: $(A \bowtie B) \bowtie (C \bowtie D)$

Partition parallelism is inter-operator parallelism that works by partitioning the data and operating on partitions in parallel.

Network Cost

The amount of data needed to send over the network (from one machine to another) to perform an operation.

Units: Usually KB

Unlike I/O cost, you don't need to send data over the network in units of pages. For example, you can send one tuple's worth of data across the network.

Partitioning Data

Range partitioning on a key divides data based on which range the key belongs to.

Hash partitioning divides data based on a hash function.

Round robin partitioning just cycles through the partitions in order as data comes in.

Parallel Sorting

1. Partition the data over machines with range partitioning
2. Perform external sorting on each machine independently

Parallel Sort Merge Join

1. Partition the data for both relations over machines with range partitioning.
   - Use the same ranges for both relations
2. Perform sort merge join on each machine independently

Parallel Hashing

Use a hash function to partition the data over all the machines (hash partitioning), then run external hashing on each machine independently

Parallel Hash Joins

Use hash partitioning on both relations, then perform a grace hash join on each machine independently

Join Pipelining

For both parallel SMJ and GHJ, the sort or hash must complete before completing the join. Therefore, a machine has to wait for all other machines to finish sorting/hashing.

Symmetric Hash Joins

- Streaming Algorithm - meaning it doesn't need all tuples of a relation to be present before joining.
- No writes to disk, only in memory.
- Requires each R and S partitions to fit in memory $(B-1)$ pages
  - Get more machines to make partitions smaller

Basic Idea: Build 2 hash tables at the same time.

When a tuple from R arrives, probe S's hash table for matches. If there is a match or multiple matches output them, then add the tuple into R's hash table. Same thing for S but vice versa.

Parallel Aggregation

To calculate aggregate functions (SUM, COUNT, etc.), use hierarchical aggregation.
- Decompose aggregate into global and local.
- Apply local aggregate on each machine independently.
- Apply global aggregate on local aggregate values to get final aggregated value.

Asymmetric Shuffles

Data is already partitioned the way we want, so no need to repartition or send any data across the network for that relation.

Broadcast Joins

When 1 table is huge and 1 is small, if the large table is not partitioned properly, it might be easier to broadcast the entire tiny table than to repartition the large table.

Distributed Computing

1. Shared nothing parallel architecture
   - No shared memory/disk, all comms over network
2. Network is unreliable
   - Packets may be delayed, duplicated, reordered, dropped
3. Clocks are not synchronized
   - Cannot just use clock to order messages - may be 12:03:05 on one machine and 12:01:55 on another
4. Cannot accurately check if a node is up or down.
   - How do you distinguish between slow network, node that is busy (but will respond eventually), and a node that is down.
5. Data is partitioned over multiple nodes.
   - If any node has a problem, then you cannot save all changes to preserve atomicity, so you have to abort.

Two Phase Commit (2PC)

Phase 1 (Voting): Coordinator asks participants to vote.
- YES to commit the transaction, NO to abort.

Phase 2 (Results): Coordinator tells participants the result of the vote.
- Requires unanimous agreement; if any participant cannot commit, no nodes commit to preserve atomicity.

In Phase 1, each participant logs and flushes either a PREPARE record (for YES) or ABORT record (for NO). Flush means force to disk. The log entry includes the coordinator ID. If a participant is voting NO, it can immediately start cleaning up (releasing locks, undoing changes) without waiting since the transaction won't go through.

In Phase 1, the coordinator collects all votes, decides to commit or abort, then logs and flushes a COMMIT or ABORT record to its own log before sending out the result. The log entry includes all participant IDs.

In Phase 2, the coordinator sends either a commit or abort message to all participants. Each participant logs and flushes a COMMIT or ABORT record to its own log. Then, they execute it and send acknowledgement to the coordinator.

In Phase 2, after all participants have acknowledged, the coordinator logs an END record to its log and removes the transaction from the transaction table. The END log record is eventually flushed, but doesn't have to be flushed immediately.

2PC Handling Failures

- Assume all machines eventually recover.

If the coordinator thinks a participant went down:
1. If the participant didn't vote yet, ABORT
2. If waiting for acknowledgement, perform recovery

If a participant thinks the coordinator went down:
1. If PREPARE record not logged, vote NO (abort)
2. If PREPARE record logged, perform recovery

2PC Recovery

If there is a COMMIT/ABORT record, execute the decision. If it's the coordinator, it still has the participant IDs, so it keeps sending COMMIT/ABORT until its acknowledged

If there is a PREPARE record, but no COMMIT/ABORT, it's a participant. The participant sends a message to the coordinator asking for the status of the transaction.

If there are no log records, as a participant, abort the transaction since you never voted YES. If you receive YES/NO messages, it's the coordinator, which should respond with ABORT.

2PC Optimization

ABORT in absence of information is called presumed abort.

Presumed Abort: Assume that a transaction aborts if we have no log records.

When a transaction aborts,
1. Coordinator cleans up locally, and can remove transaction from table without ACKs.
2. Participants that receive ABORT messages don't send ACKs.
3. If participants don't hear from coordinator about status, send inquiry. If transaction not in coordinator's transaction table, return ABORT.
4. Participant IDs do not need to be stored in abort records (since we aren't waiting for ACKs.
5. Abort records do not need to be flushed (if we crash and don't see an abort record, we still assume it aborts).

2PC Blocking

When a node goes down during Phase 1, any participant that voted yes, has to keep locks, waiting for commit/abort message.

If a participant doesn't recover after coordinator sends COMMIT message, coordinator respawns participant, recovers from log and continues. If the old instance comes up, tell it to recycle itself.

If the coordinator doesn't recover, use Paxos

Paxos

Phase 1 (Election):
- Proposer picks a ballot id.
  - Ballot id must be unique per proposer and higher than any ballot id seen so far
- Proposer sends PREPARE(ballot_id) to all participants
- Each participant keeps track of highest ballot id received so far (max_bid)
- If participant has already received a higher ballot id (max_bid > ballot_id) do nothing
- Else, write ballot_id to log and flush log record to disk.
  - Send PROMISE(ballot_id) back to proposer
- If already sent an ACCEPT(old_ballot, value) where ballot_id > old_ballot, then send PROMISE(ballot_id, (old_ballot, value))
- If majority respond PROMISE(ballot_id), then proposer is the new leader
- If majority respond PROMISE(ballot_id, (old_ballot, v)):
  - Select v with highest old_ballot value as the proposed value v in phase 2. Still use ballot_id though

Phase 2 (Proposal):
- Leader sends proposed value v by sending PROPOSE(ballot_id, v) to all participants
- If a participant has already received a higher ballot id (max_bid > ballot_id), do nothing.
- Else, write ballot_id to log and flush log record to disk
  - Send ACCEPT(ballot_id, v) to proposer

Phase 3 (Decision):
- If leader hears a majority of ACCEPT(ballot_id, v):
  - Writes ballot_id to log and flush log record to disk
  - Lets everyone know the decision by broadcasting COMMIT(ballot_id, v)
- Participants can now act on v.
- If no such majority exists, then the round failed and a new leader was chosen
- If there was a majority, any subsequent round chooses v as the consensus value.

N = # of nodes/machines
$N \geq 2F+1$ to tolerate F failures, nodes fail by stopping

Requirements:
1. Safety: Only a proposed value is chosen, and only one value is ever chosen
   - Majority = F+1, if F nodes fail, at least 1 node lives and carries the accepted value and will include it in future PROMISES.
2. Liveness: Replicas eventually converge to an agreed upon value.
3. If $N < 2F+1$, then no guarantees, too many failures for Paxos to work.

Livelock

When two proposers keep outbidding each other with higher ballot ids so no progress is made

Solution:
Dedicate a fixed leader as the only proposer.
- Elect through Paxos or other algorithm
- Elect new leader when leader dies/crashes
- Essentially, skip phase 1 of Paxos

Paxos Logging

On PROMISE: participant logs and flushes ballot id

On ACCEPT: participant logs and flushes ballot id and v

When majority accepts: proposer logs and flushes ballot id and v

Paxos Alt Ending

Normal Ending: Broadcast COMMIT message
- Pros: O(N) messages sent
- Cons: Low reliability and requires an extra round of communication

Alt Ending: Participants broadcast to other participants when it accepts a value
- Pros: One less round of communication and more reliable
- Cons: $O(N^2)$ messages sent

Workloads

Online Transaction Processing (OLTP)
- Transactional (every operation is ACID transaction)
- Fast processing (queries are simple and return quickly)
- Normalized (data split into tables, no redundancy)
- Current data (stores only whats needed now)

Online Analytical Processing (OLAP)
- Analytical (queries are read only and designed for analysis)
- Slow queries (scans massive amts of data with joins and aggregations)
- Denormalized (less tables, redundant data, less joins needed for analytical queries)
- Historical data (stores past data for trend analysis)

Scaling

Partitioning/Sharding: Split data among multiple machines to increase parallelism
- Fast writes (parallel writes)
- Slow reads (may need to query multiple machines to find data)

Replication: Copy data among multiple machines to increase fault tolerance
- Slow writes (must update all copies)
- fast reads (more machines to read from)

Distributed Systems

Desired Properties (CAP)
1. Consistency - all clients see the same data at the same time
2. Availability - every request gets a response
   - Only errors if input has error
3. Partition Tolerance - system keeps working even if messages between machines are delayed or dropped, or even if certain machines are disconnected from the network.

CAP Theorem: Can only guarantee 2/3 of the above.
- P is always kept
- Tradeoffs between C & A
  - If keep C, timeout if data might not be up to date
  - If keep A, may respond with stale data
- In practice, many systems prioritize availability and provide eventual consistency. Basically, all replicas converge to the same state once all updates stop

BASE Semantics

- Eventually Consistent Systems adhere to the 3 BASE guarantees instead of ACID.

1. Basic Availability: Reads/Writes available as much as possible but may be stale/inconsistent
2. Soft State: Data can change on its own (as updates spread) so the application only has a probability of knowing its state
3. Eventually Consistent: Given enough time (after all updates spread) all reads will be consistent

NoSQL

1. Key-Value Stores: Every piece of data has a unique key (string/integer) and a value (can be anything).

Operations:
- get(key), put(key, value)
- Operations on value are not supported due to value type flexibility

Partitioning (Hash Partitioning) - If there is replication, use more hash functions (i.e. 3 hash functions for 3 replicas)

2. Extensible Record Stores

Variant 1: key = rowID, value = record

Variant 2: key = (rowID, columnID), value = field
- Can have multiple columnIDs in the key

Each row doesn't have to have the same columns.

Operations:
- get(key), get(key, [columns]), put(key, value)

3. Document Stores - Key-Value but Value is always a structured document (JSON, XML, etc.)
- Documents stored in collections
  - Users collection stores user documents, etc.

JSON - JavaScript Object Notation

Supported Types:
1. Object: {}, key-value pairs, no duplicate keys
2. Array: [], ordered list of values
3. Atomic: a number (64 bit float), string, boolean, or null

JSONs are self describing - the schema elements are part of the data, which allows each document to have its own schema. JSONs can be represented as trees, due to its nested structure.

JSON vs Relational

	JSON	Relational
Flexibility	Very flexible, can represent complex structures and nested data	Less flexible
Schema Enforcement	Self-describing; Each document can have unique structure	Schema is fixed
Representation	Text-based (easily parsed and manipulated by many languages)	Binary representation (designed for efficient storage and retrieval from disk)
	Enforcing schema on read	Enforcing schema on write

Relational → JSON

name of table → key
rows → array of objects

Student

name	grade
Su Min	86
Sarah	55
Soumya	91

→

```
{"student":[
  {"name":"Su Min","grade":86},
  {"name":"Sarah","grade":55},
  {"name":"Soumya","grade":91}
]}
```

For many relations:
Option 1: Keep each relation as a separate JSON array
Option 2: Inline relations based on keys (Student → Classes Taken → Subjects - put all subject info into each classes taken object, and put all classes taken for a student into the student object.

MongoDB Query Language (MQL)

Stores documents which are dictionaries of key-value pairs. Same as JSON. Each document has a special "_id" field which represents its primary key. _id is indexed and the first attribute of each document. ↑ B+ Tree

MongoDB	DBMS
Database	Database
Collection	Relation
Document	Row/Record
Field	Column

- Operate on collections (input: collection, output: collection)
  - Usually one at a time
- Dot notation for queries
  - db.collection.op1(...).op2(...) where collection is the name of the collection and op1/2 is the name of the operation

Supported Value Types:
1. Nested Documents
2. Array: [], ordered list of values
3. Atomic: a number (64 bit float), string, boolean, or null

MQL Syntax:
1. "field.nestedField" → access field inside nested document
2. "array.i" → index into array (0-indexed)
3. "array.i.field" → field of the ith element in array of nested docs
4. "array.field" → if array of nested docs, returns array of that field from every element
5. Dot expressions must be in quotes
   - Ex: {"country.population": 1}

$ indicates the string is a special keyword ($gt, $lte, $add)
- {"population": {$gte: 1000}}

MQL Retrieval

MongoDB Query Language	Relational Database Equivalent
find(<predicate>, optional <projection>)	SELECT <projection> FROM Collection WHERE predicate
limit(<integer>)	LIMIT
sort(<list of fields>)	ORDER BY

- find({status: "In Stock"}) - for atomic fields, just check if field equals that value exactly
- find({seats: {"num": 5, "type": "leather"}}) - for nested documents, checks exact match including field order.
- find({reviews: [5,5]}) - checks exact match of the whole array
- find({reviews: 5, reviews: 4}) - checks if reviews contains 5 and 4.
- find({$or: [{car: "Honda"}, {status: "In stock"}]}) - $or takes an array of predicates and returns docs matching any of them

Projection is the second argument to find
- 1 means return that field, 0 means don't.
- _id is returned by default.
- can't mix 1s and 0s other than for _id.
- find({...}, {car: 1}) returns car and _id fields
- find({...}, {car: 1, _id: 0}) returns only car
- find({...}, {car: 1, reviews: 0}) errors since 0s and 1s is mixed

Sort/Limit - dot operators: find(...).sort({"reviews.0": -1}).limit(1)
For sort, -1 is descending, 1 is ascending

MQL Aggregation

Syntax: (collection → replace w/ collection name)

```
db.collection.aggregate([
  {$stage1Op: {...}},
  {$stage2Op: {...}},
  ...
])
```

Pipeline of stages, each takes the output of the previous as input

$group = GROUP BY

```
{$group: {_id: "$fieldname", newField: {$aggFunc: "$fieldname"}}}
```
(_id: "$fieldname" → group by fieldname; newField: {$aggFunc: "$fieldname"} → compute newField)

aggFuncs: $sum, $avg, $max, $first, $push, $addToSet
- $first: first value per group (use after sort)
- $push: collects all values of fieldname into an array
- $addToSet: same as push but no duplicates

$lookup = JOIN

```
{$lookup: {from: "OtherCollection", localField: "field", foreignField: "field", as: "outputArray"}}
```

For each doc in current collection, finds all docs from OtherCollection where foreignField matches localField and puts them in outputArray.

$match = WHERE, $project = SELECT, $group = GROUP BY, $lookup = JOIN, $sort, $limit

MQL Updates

db.collection.insertOne/insertMany([{doc1}, {doc2}, ...])
- creates collection if it doesn't exist and auto adds _id to each doc if not provided

db.collection.updateOne/updateMany({condition}, {change})
- {condition} = predicate
- {change} = {$set: {status: "Delivered"}}

db.collection.deleteOne/deleteMany({condition})
- deletes docs matching {condition}

MapReduce

- process massive amounts of data in parallel

3 Phases:
1. Map: User provides MAP function
   - Input: (input key, value)
   - Output: bag of (intermediate key, value)

   System applies the map function in parallel to all (input key, value) pairs in the input file
2. Shuffle: internal to the system
   - Input: Pairs of (intermediate key, value)
   - Output: Pairs of (intermediate key, bag of values with the same intermediate key)

   Group all values with the same key
3. Reduce: User provides reduce function
   - Input: (intermediate key, bag of values)
   - Output: bag of output (values)

Workers, Fault Tolerance, and Implementation

Worker - single process handling one task at a time.
- 1 worker per CPU core

MapReduce Fault Tolerance:
1. Mappers write their output to their local disk.
2. Reducers then fetch and reshuffle those files. If the server crashes mid reduce, the task simply restarts on a different worker with no data loss

Implementation:
1. 1 master node orchestrates everything
2. Master splits the input into M splits, by key, and assigns them to workers
3. Workers write output to local disk, partitioned into R regions (1 per reducer)
4. Master assigns R reduce tasks to workers, who pull the relevant regions from each worker's disk from step 3.

Straggler - a machine that takes unusually long to complete one of the last tasks.

Solution: pre-emptive backup execution of the last few remaining tasks.

Spark

Similar to MapReduce but better for multistep jobs because it stores intermediate results in memory.

RDDs = Resilient Distributed Datasets
- Distributed (split across machines), unchangeable relation with its lineage
- Lineage: expression that says how that relation was computed.

- All inputs, outputs, and intermediates, stored as RDDs.
- If a server crashes and loses its RDD partition, the master can recompute the lost partition from the lineage.

Spark programs consist of:
1. Transformations (map, reduceByKey, join, flatMap, filter, groupByKey, union, cogroup, crossProduct) - lazy
   - take an RDD and produce a new one
   - describe what to do, but don't actually do it (lazy)
     - Operator tree constructed in memory instead
2. Actions (count, reduce, save, collect)
   - actually trigger execution and return a result (eager)
   - runs operator tree

RDD<T>: RDD collection of type T
- partitioned, recoverable through lineage, not nested

SEQ<T>: sequence of type T, local to a server, can be nested