CS 186 Midterm 1 Cheat Sheet — Jeffrey Shen

1 SQL - Basics Queries

Tables (relations): makes up relational databases
has name, rows, columns

name	age	num_dogs
Ace	20	4
Ada	18	3
Ben	7	2

Boolean Operators: NOT, AND, OR
NULL: special value any type, falsey

Grouping and Aggregation

- summarize cols of data (SUM, AVG, MAX, COUNT)
- input is name of col, ignores NULL except COUNT(*)

WHERE occurs before grouping, filter out rows
HAVING occurs after grouping, filter out groups
ORDER BY: default sort order ascending
can add DESC, add columns for breaking ties
Strings: LIKE %: 0+ chars, _ any char 's%'

2 SQL - Joins and Subqueries

Cross Join: combine every row from left w/ right
Inner Join: use ON clause to specify condition
Left Outer Join: every row from left in output
Full Outer Join: all rows from each in output
Natural Join: automatically equijoin on cols w/ same name
Subqueries: use new table inside the query

```
SELECT    <columns>
FROM      <table>
WHERE     <predicate>
GROUP BY  <columns>
HAVING    <predicate>
ORDER BY  <columns>
LIMIT     <nums>
```

```
WITH <table_name> (<values>) AS
  (SELECT ...),
  <table2_name> (<values>) AS
  (SELECT ...)
SELECT ...
```

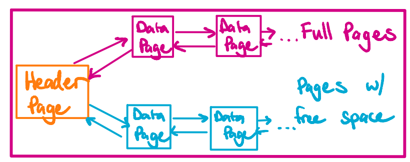
3 Disk and Files

Disk: READ/WRITE RAM ↔ Disk
↳ Platters spin at 15000 rpm so arm assembly reads track of sector size
Solid State Drives (SSDs) store data, organized into cells, support random fast R
Disk Space Management: lowest layer of DBMS, manages space on disk

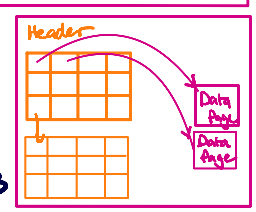
Files, Pages, & Records

Database data records organized into relations and can be modified in memory
Page: basic unit of data for disk
Table stored in file, records organized into pages in file

File Types



Heap File: no particular ordering of pages or of records on pages
↳ Linked List Implementation: data page has records, free space tracker, pointers next prev



↳ Page Directory Implementation: linked lists for header pages, entries are pointer to data page and free space left ↑ inserting records faster, only look through headers
Sorted Files: pages are ordered and records sorted by keys
↳ Page Directory Implementation: enforce ordering based on how records sorted
↑ Searching: log N I/O ↓ Insertion: log N + N I/O

-Count header page I/O when file type specified

Inserting requires checking key is unique ⇒ read all data pages

Record Types

Fixed Length Records (FLR): fixed length fields, same bytes
Variable Length Records (VLR): both fixed and variable length fields, stores fixed length first, pointers to end of variable length record id: [page #, record # on page]

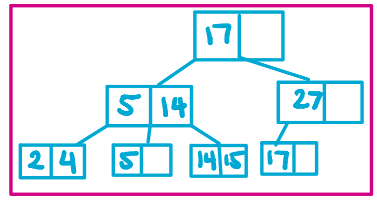
Page Formats

FLR Pages → Packed: calculate next pos for insertion, deletion requires moving records
→ Unpacked: Store bitmap and track open slots

records = ⌊(data page size − 8)/(record size + 8)⌋

VLR Pages
Page footer to maintain slot directory, tracks slot count, free space pointer, entries record len, pointer
↳ unpacked insertion: at free space pointer, new [pointer, length] pair set, periodically packed

4) B+ Trees



- order of a B+ tree d
- must have $d \le x \le 2d$ entries, sorted
- inner nodes at most $2d+1$ child pointers
- only leaf nodes contain records

Insertion 1.) Find leaf node L to insert, add key and record in order
2.) If overflow,
a) split L_1, L_2 where L_2 has $d+1$ entries
b) If L leaf node, COPY L_2 first entry. If not leaf node, MOVE L_2 first entry into parent
3.) If parent overflow, recurse step 2

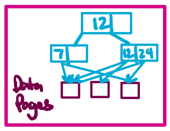
Deletion 1.) Find appropriate leaf and delete, never delete inner nodes

Storing Records

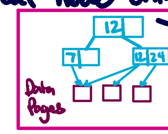
Alt 1) By Value: leaf pages contain records (key, val) ↓ cannot support multiple indexes
Alt 2) By Reference: pointers to coorsponding pages (key, [PageNum, RecordNum]) ↑ multiple indexes
Alt 3) By List of References: list of pointers to corresponding pages (key, List of RecordID) ↑ multiple records w/ same leaf node entry

Clustering

Unclustered (Alt 2,3): read in each data page they point to



Clustered: (Alt 1): data pages sorted on same index as B+ tree



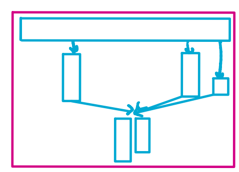
Counting I/Os
1. Read root-to-leaf path
2. Read data pages
3. Write data page if modify
4. Update index page

Bulk Loading construct B+ tree from scratch, better cache use
1. Sort data on key index is built on
2. Fill leaf pages until fill factor f for leaf nodes
3. Pointer from parent to leaf, if overflow
   a. keep d entries in L_1
   b. MOVE L_2 first entry into parent

5) Buffer Management

$$\text{Hit Rate} = \frac{\text{Page Hit}}{\text{Page Hit} + \text{Page Misses}}$$

Buffer Management responsible for eviction policy
Metadata Table: Frame ID (memory addr), Page ID (page on frame), Dirty Bit (whether modified), Pin # (pins)
Least Recently Used (LRU): last used col, lowest value evicted ↓ costly, sequential scanning $|S| >$ buffer pool size
Clock Policy: approx LRU, ref bit, clock hand to track frame, set ref bit to 1
eviction: iterate through frames, if ref = 1, set 0, when found ref = 0, evict, set new page to 1, move clock
if accessing page in buffer pool, set ref bit to 1 w/o moving clock hand
Most Recently Used (MRU): evict most recently used ↑ sequential scanning



6) Sorting

Full External Sort

Conquer first by sorting records on individual pages, merge pages together => Sorted runs
B buffer pages available, can merge together B-1 input buffers

I/Os: $\underbrace{2N}_{R+W}\left(\overbrace{1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil}^{\text{passes}}\right)$

N = 180 pg	→	22 runs	8 pg	→	3 runs	56 pgs	→ 1 run 20x
B = 8		1 run	4 pg		1 run	12 pages	

7) Hashing

grouping like values together, want to build several hash tables and concatenate
First partitioning pass hash into B-1 partitions, recursively hash until partition has less than B

Analysis

depends on partitions,
I/Os: go through passes $\left(\sum_{i=1}^{m} r_i + w_i\right) + 2X$

r_i = # pages to read in for partitioning pass i
m = total partitioning passes req
w_i = # pages to write in partitioning pass i
X = total pages after partition to build hash table

Properties $r_0 = N$ $\quad w_i \ge r_{i+1}$
$r_i \le w_i$ $\quad X \ge N$

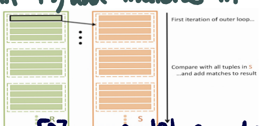
CS 186 Midterm 2 Cheat Sheet — Jeffrey Shen

8 Joins

1) Simple Nested Loop Join (SNLJ)

For every record in R, add matches in S

```
for each record r_i in R:
    for each record s_j in S:
        if θ(r_i,s_j):
            yield <r_i, s_j>
```



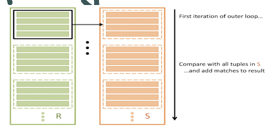
I/O: [R] + |R|[S]

[R] pages, |R| records

2) Page Nested Loop Join (PNLJ)

per page in R, loop through S to match

```
for each page p_r in R:
    for each page p_s in S:
        for each record r_i in p_r:
            for each record s_j in p_s:
                if θ(r_i, s_j):
                    yield <r_i, s_j>
```

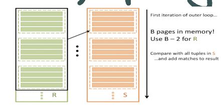


I/O: [R] + [R][S]

3) Block Nested Loop Join (BNLJ)

use B-2 buffer to store R, loop through S

```
for each block of B−2 pages B_r in R:
    for each page p_s in S:
        for each record r_i in B_r:
            for each record s_j in p_s:
                if θ(r_i,s_j):
                    yield <r_i, s_j>
```



I/O: $[R] + \left\lceil \frac{[R]}{B-2} \right\rceil [S]$

4) Index Nested Loop Join

create indexing tree, lookup each r, find s

```
for each record r_i in R:
    for each record s_j in S where θ(r_i,s_j)==true:
        yield <r_i, s_j>
```

I/O: [R] + |R| * (cost to look up records in S)

5) Naive/Grace Hash Join

create hash table B-2 pages big for R

for Grace Hash, keep hashing until B-2

I/O: [R] + [S]

6) Sort-Merge Join

sort R and S, advance until match; mark in S, look in S until no match; go next in R marked in S

```
do {
    if (!mark) {
        while (r < s) { advance r }
        while (r > s) { advance s }
        // mark start of \block" of S
        mark = s
    }
    if (r == s) {
        result = <r, s>
        advance s
        yield result
    }
    else {
        reset s to mark
        advance r
        mark = NULL
    }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

I/O: [R] + [S] + cost sort R + cost sort S

9 Relational Algebra

procedural programming language

- **Projection** π [SELECT] takes in relation, selects column
- **Selection** σ [WHERE] takes in relation, filters rows
- **Union** ∪ combine different relations, remove duplicates
- **Set Diff** − returns every row in table 1, not in table 2
- **Group By** $\lambda_{col, group}$ [GROUP BY/HAVING] group by features
- **Intersection** ∩ rows in both tables
- **Cross Product** × one tuple for every possible pair btwn relations
- **Join** ⋈ [JOIN] default natural join
- **Rename** ρ [AS] aliasing

10 Query Optimization

Find the query plan to minimize the I/Os to execute the query

Use iterators for streaming or blocking (need entire input)

Selectivity Estimation: approx for percentage of pages making it through operator

- $X = a$: $1/$(unique vals in X)
- $X = Y$: $1/\max$(unique vals in X, unique vals in Y)
- $X > a$: $(\max(X) - a)/(\max(X) - \min(X) + 1)$
- cond 1 AND cond2: Selectivity(cond1) * Selectivity(cond2)

1) Join selectivity: $\dfrac{|A| * |B|}{\max(\text{unique vals } A.id, \text{unique vals for } B.id)}$
2) Est. no of joined tuples by multiplying selectivity of join w/ joined tuples in Cartesian Product
3) Est. no. of pages by dividing by tuples per page

Heuristics to find best query plans

1) Push down projects (π) and selects (σ)
2) Only consider left deep plans, can be piplined
3) Do not consider cross joins unless only option

System R (Selinger Optimization)

1st Pass: full scan [P] or index scan
- Alt 1: cost to level above leaf + num leaves read
- Alt 2/3: Alt 1 cost + data pages read

advance optimal access plan and an optimal interesting order col used in ORDER BY, GROUP BY, or join

GHJ, PNLJ, BNLJ never have interesting order

SNLJ, INLJ can preserve sorted ordering on left ordering

Assume we never materialize operators

11 Transactions & Concurrency

1) Inconsistent Reads: user reads part of what was updated
2) Lost Update: two users try to update at same time and one gets lost
3) Dirty Read: one user reads update not committed
4) Unrepeatable Reads: reads two values for same record bc another user updated in between

Transactions are sequence of multiple actions executed as single, logical, atomic unit

1) Atomicity: commits or aborts, all happen or none
2) Consistency: starts and ends consistent
3) Isolation: isolated from other transactions
4) Durability: if transaction commits, effects persist

Concurrent Execution ↑ throughput, ↓ latency

Transaction Schedule: Begin, Read, Write, Commit, Abort

Want to find schedules serializable so same as in serial

Check serializability by building dependency graph:
- one node per transaction
- edge from T_i to T_j if operation O_i of T_i conflicts w O_j of T_j or O_j appears earlier than O_j

- Conflict serializable iff dependency graph acyclic
- View serializability finds schedules conflict serializable
- Blind writes are sequential writes w/ no interleaving reads

12 Transactions & Concurrency II

Two Phase locking (2PL): ensure conflict serializable schedules 1) transactions need shared lock (S) before reading, exclusive lock (X) before writing 2) cannot acquire new locks after releasing any locks

Does not prevent cascading aborts
↳ Strict 2PL: all locks released together when transaction done

Lock Manager: hash table of resources, granted set, lock type, wait queue, either granted or put in queue

	Granted Set	Mode	Wait Queue
A	{T1, T2}	S	T3(X) -> T4(X)
B	{T6}	X	T5(X) -> T7(S)

Deadlock: cycle of Xacts waiting for locks to be released
↳ Avoidance: avoid deadlocks

wait die: if T_i higher priority, T_i waits for T_j else T_i aborts
wound wait: if T_i higher priority, T_j aborts, else T_i waits
↳ Detection: maintain "waits-for" graph: edge if hold lock or attempt to acquire lock

Lock Granularity: want to allow more granularity
IS, IX lock has more granularity
- must hold IS/IX of parent node

(13) Recovery

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

Force Policy: when transaction finishes, force pages to disk
No Force: only write back when evicted from buffer pool
No-Steal Policy: pages cannot be evicted until transaction commits
Steal Policy: allow modified pages to be written to disk before transaction finishes
Steal, No Force
↳ Write-Ahead logging <XID, pageID, offset, length, old_data, new_data>
- log records written to disk before data page to disk
- all log records written to disk when transaction commits

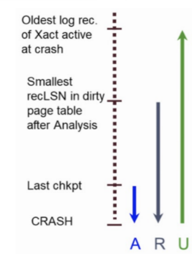
	No Steal	Steal
No Force		Fastest
Force	Slowest	

Performance Implications

	No Steal	Steal
No Force	No UNDO REDO	UNDO REDO
Force	No UNDO No REDO	UNDO No REDO

Logging/Recovery Implications

Log Sequence Number (LSN) to track order of operations,
prevLSN: last operation from same transaction
flushedLSN: to track last LSN
Abort: write ABORT, undo each operation from bottom-up, write CLR (Compensation Log Record)
Recovery: recover from crash
Transaction Table: XID: transaction ID, Status, lastLSN
Dirty Page Table (DPT): page ID, recLSN (first op to dirty table)

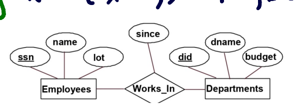


Undo logging: want to undo if not committed
4 types: Start, Commit, Abort, Update
1) For transaction modifying data element, update log record written to disk before dirty page
2) if committed, write to disk before commit record
scan log from end to find transaction completed or not, if T not completed, write X=v to disk
Redo Logging: no force, no-steal, redo all transactions, both update record & commit record written before dirty pg
ARIES Recovery Algo: Analysis, Redo, Undo
Analysis: rebuild the Transaction table, DPT
if not END, add to transaction table, set LastLSN
if COMMIT or ABORT, transaction status change
if UPDATE, not in DPT, add to DPT, recLSN set to LSN
if END, remove from Transaction Table

Checkpointing: writes Transaction Table and DPT to log
redo from smallest recLSN in DPT unless
1) page not in DPT, recLSN > LSN, pageLSN(disk) ≥ LSN
Undo: Start from end of log to start undoing updates

(14) DB Design

Entity-Relationship Model: entity object □ w set of attribute values ○
relationship: association among 2+ entities ◇, many-to-many
Use key constraint to denote 1-to-many relationship, 0 or more,
participation constraint: at least one, thick line
weak entity: identified uniquely w/ primary key of another entity
Avoid redundancies: functional dependencies $X \to Y$ X determines Y
Superkey: set of columns that determine all columns
Candidate key: set of columns that determine all columns
decomposition is lossy if can't reconstruct, R into X, Y, $X \bowtie Y = R$
$X \cap Y \to X$ ($X \cap Y$ is superkey of X)
$X \cap Y \to X$ ($X \cap Y$ is superkey of Y)
Dependency Preserving if $(F_X \cup F_Y)^+ = F^+$, BCNF not necessarily lossless



BCNF Decomp: R w FD's F in BCNF for all $X \to A$ in F^+ if $A \subseteq X$, X superkey for R, lossless
algorithm:
Input: R, F
$R = \{R\}$
if relation $r \in R$ not BCNF
a) Pick violating FD f: $X \to A$ st $X, A \in$ attributes of r
b) Compute X^+
c) let $R_1 = X^+$, $R_2 = X \cup (r - X^+)$
d) remove r from R
e) Insert R_1 and R_2 into R
f) Recompute F as FDs over all relations $r \in R$

(15) Parallel Query Processing

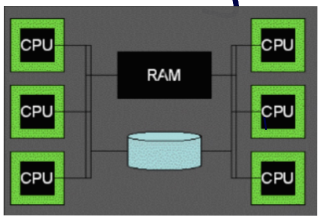
query run on multiple machines in parallel

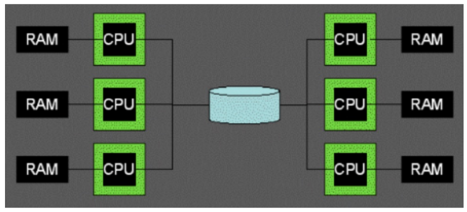
Parallel architectures

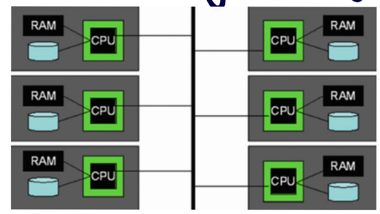
Shared memory: every CPU share memory and disk

Shared disk: CPU has own memory but share disk

Shared nothing: machines communicate through messages







Intraquery parallelism: spread work of one query over multiple machines

↳ Intra-operator: make one operator run as quickly as possible (ex. sorting on multiple)

↳ Inter-operator: running operators in parallel (ex. sort S, sort R on another)

↳ Pipeline Parallelism: records passed to parent as soon as done

↳ Bushy Tree Parallelism: different branches of operators of tree run in parallel

Interquery parallelism: gives each machine different queries for higher throughput & finish more queries

Sharding: each data page stored only on one machine

Replication: each data page on multiple machines

Partitioning scheme to find which machine a certain record is on

Range Partitioning: each machine stores certain range
↑ key lookup, range query

Hash Partitioning: each record hashed sent to machine
↑ key lookup, ↓ range query

Round Robin: assign each record to next machine
↓ every machine activated for every query
↑ every machine has same data

Network Cost: how much data to send over network to do operation

Parallel Sorting/hashing: range partition table, local sort/hash on each machine

Passes: 1 (partition across machines) + $\lceil 1 + \log_{B-1} \lceil N/mB \rceil \rceil$ (number of passes needed to sort table)

SMJ Passes: 2 (1 Pass/table to partition across machines) + $\lceil 1 + \log_{B-1} \lceil \overset{S}{R}/mB \rceil \rceil$ (passes to sort R+S) + 2 (final merge sort pass)

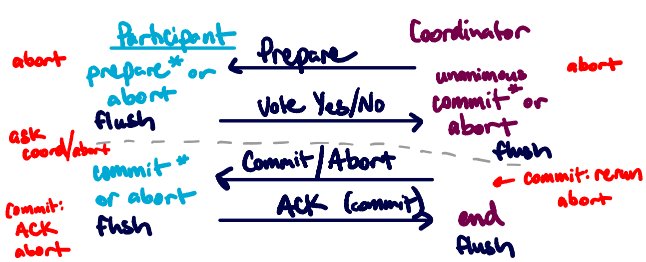
Non-pipeline breaker: Symmetric Hash Join: two hash tables, R record probe in S for matches, vv

For Hierarchical Aggregation: send data to coordinator nodes for COUNT, AVG

(16) Distributed Transactions

Every table has own local lock table, union waits-for graphs for deadlocks

2 Phase Commit: ensure all nodes reach consensus for a transaction



T/F

- I/O cost of performing full scan on a sorted file is same as scanning a heap file, assuming both are packed

- Page directory keeps track of amount of free space on data pages

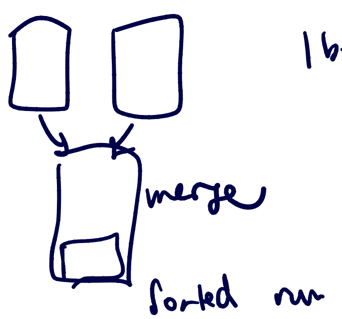
- Deadlocks: wait-die will avoid all scenarios of deadlock
- deadlock avoidance aborts many

- leave aborting transactions on txn table, last operation by T2 for last LSN
- recLSN, last operation to dirty P1 at LSN

Tips
- Don't join on wrong col,
- Don't cross join, pay attention to col that join is on

Sorting

sort records on individual page



1 buffer page to sort

Hashing