CS186 MT1

1. SQL

DDL: data definition language
- CREATE TABLE
- data types: text, float, integer, char

DSL: data manipulation language

SELECT [DISTINCT] <column expression list>
  FROM <single table>
  [WHERE <predicate>]
  [GROUP BY <column list>
    [HAVING <predicate>]]
  ]
  [ORDER BY <column list>]
  [LIMIT <integer>]]

- when using group by, must have same regular expression in the select statement, or put it inside aggregate

ORDER of reading SQL Logic:
FROM, WHERE, SELECT, GROUP BY && HAVING, DISTINCT, ORDER BY, LIMIT
- if a WHERE clause evaluated to null, not in output
- aggregates cant be in WHERE clause

CROSS JOINS – can't have aggregation in WHERE clause
From A, B
- cross product. Every row in left with every row on Right

INNER JOIN
same as CROSS JOIN but need ON.

OUTER JOIN
- Even if ON predicate doesnt match, still in output

LEFT means
- still if no match, left columns table are still there with NULL

RIGHT
- vice versa

FULL OUTER JOIN
if either table no match, its still there.

- Duplicate key in Primary key not allowed by definition

Facts
- two same queries not deterministically the same if theres a tie

Relational Algebra
- no duplicates
- all operators take in relation and output different relation

Projection (π)
SELECT: select only columns specified

Selection (σ)
WHERE clause:
$\sigma_{age=12 \wedge name='SAM'}$

Union ($\cup$)
- must be same columns

($-$) set diff

($\cap$) intersection

Joins ($\bowtie$)
no specification = natural join (join on tables w/ same names)

Rename (ρ)
cats $\bowtie_{name=dname}$ $\rho_{name \to dname}$ (dogs)

pattern Matching

LIKE
- if NO % or _, then ' ' acts like equality
- an underscore stands for matches single character
- % matches any sequence of zero or more chars.

standard regex
~
- (.) represents single wildcard character
- (*) represents repetition of prev item zero or more times
- a.c = abc valid

2. Disks, Files, Buffers

Disk vs Flash (SSD)

Disk: Accessing a page
- seek time, rotational delay, transfer time
- Random read much slower than sequential
- random write faster than sequential write

Flash: Random write faster than sequential
- faster than disk for 10-100x random IOs

- Locality matter for both
- Disk 10x/capacity/$ dollar

Files > Pages > Records

FillFactor is for preventing splits on new inserts, not IO.
↗ ONLY FOR LEAF

Bulkload – build from scratch
1. Sort data on key of index
2. Fill leaf pages till ff f.
3. Add ptr from parent to leaf. If parent overflows, follow split parent.
   a) keep d in L1. d+1 in L2
   b) move L_2 first entry up.
4) adjust ptrs

Storing Records

Alt 1: leaf pages are records themselves

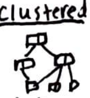
Alt 2: leaf pages are pointers to corresponding record

Alt 3: linked list of pointers to corresponding records

Clustering

1. Unclustered

2. Clustered



- better caching
- when multiple query, we might have that page already since sequential.

File Structures

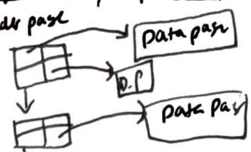
Heap File – no particular ordering

1. Linked List implementation
- Each data page has records, free space tracker, pointers
- one header page as start

I/O
- reading all data pages in one header is only 1 I/O.

2. Page Directory implementation

Header page



- pointer to data page and free space info about that page
- faster insertion in terms of I/Os
  - read header, read data, write data, write header
- fast insertion
- slow search

Sorted File – pages are ordered and records on pages sorted by keys
- implemented w/ Page Directories
- LogN search
- LogN+N insert because shift

Record Types

Fixed Length – only fixed types and all same length records

Variable Length – variable length. Fixed length fields before variable length fields and header has pointers to the end of the variable field

Record ID: [page #, record # on page]

Page Formats

1. Pages w/ FLR: page header to store # of records
   - Packed:
   - Unpacked: bitmap in header

2. Pages w/ VLR:
   - page footer that maintains slot directory tracking
     - slot count, free space pointer, entries
     - 4 bytes + 4 bytes + 8 * # records
     - ↳ [record pointer, record length]

Heap vs Sorted

Heap: good for frequent deletes, inserts, updates. fine for frequent full scan

Sorted: good for range searches, frequent lookup

3. B+ Trees/Indexes

Index: data structure that allows fast lookup, to certain key

B+ Tree Properties
- d: order of tree. Each node except Root must have $d \le x \le 2d$ entries assuming no delete. Sorted
- Inner Node: 2d entries, 2d+1 children ptr. (tree fanout)
- The keys in the children to the left of an entry must be <. To Right ≥ than.

INSERTION
1. Find leaf node to insert. Add key, record to leaf
2. if overflow ($L > 2d$):
   a) split into L_1, L_2 – D in L1 D+1 in L2
   b) if L is leaf, COPY L2's first entry into parent. else, move
   c) adjust pointer
3. if parent overflow, recurse on it w/ step 2.

DELETE
- just delete in leaf

Total Capacity: $(2d)(2d+1)^h$ where h = # edges from R to L

Counting I/Os
1. Read appropriate root to leaf. one I/O per node
2. Read appropriate data pages. IO per page. (account for clustering)
3. Write data page, if modifying. if we want to write that spans multiple pages, IO per page
4. update index page (Node)

1 I/O per linkedlist data page

- when searching key, theres not always a datapage associated so BEST CASE is NOT bringing I/O for data.

5. Sorting

Goal: SORT pages in Disk

External Merge SORT

Assume N data pages in Disk, B buffer pages available

1. Load B buffer pages fully and sort that group. do that for all $\lceil N/B \rceil$ sorted RUNS.
2. Merge B-1 pages recursively

$O(2N \cdot \underbrace{(1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)}_{passes})$

Two Way: $2N \cdot (1 + \lceil \log_2 N \rceil)$

5. Hashing

Goal: Group same values pages in Disk

External Hashing

Assume B Buffer frames available

Divide Phase

- 1 input buffer
- B-1 output buffers / Partition
- group B-1 partitions into disk

Conquer Phase

- Start when all partitions fit in B pages
- use fine grained hash function

Fact: 2 pass if B(B-1) pages

If some of the pages/partitions are ≥B, first finish the pass and in the next pass recursively partition just that partition. So split that into B-1 as usual.

- Usually write more pages after reading *. Round up. So, 12 data pages, 6 buffer pages, each of 5 frames get 12/5 = 3 each. So write 15. and read 12
- Dont forget last pass of Reading / writing all the pages after fitting.

4. Buffer Management

- buffer manager works with disk space manager and RAM.
- between RAM and DISK
- B+ code makes read requests from Buffer Manager
- Metadata: FrameID, PageID, DirtyBit, PinCount

Buffer Pool

Memory converted to buffer pool by partitioning into frames for pages.

1. FrameID: uniquely associated w/ memory address
2. PageID: determining which page a frame contains
3. Dirty Bit: verifying if modified
4. PinCount: tracking number of requestors using a page

Handling Page Requests

```
upon page request:
    if page exists in memory:
        1. pin count ++
        2. return address of page
    else:
        if space:
            1. read page into it
            2. pincount = 1
            3. return page
        else:
            REPLACEMENT POLICY!
```

LRU

pros: good for repeated access
wins for random access / popular access

Cons: LRU page finding requires priority Queue $O(\log_2 N)$
sequential flooding

Clock (LRU)

- extra metadata: Ref Bit

Adding Page

1. if page exists: just return it. Dont move any clock pointers. set the reference bit
2. if doesnt exist:
   1. Move clock pointer around.
      1. if not reference bit, evict, put in, set ref bit, move clock
      2. if ref bit, unref bit, move clock

MRU

- evict most recently used unpinned page
- Pincount reduced by whoever uses the page, NOT Always BUFFERMANAGER
- file/index management code / page requestor sets dirty bit.

Conceptual Ideas

- B+ Tree:
  API: get(key, recordID)
- Buffer Manager:
  Stores pages
- Remember a B+ Tree is just defined by a "page" Root, so this is something we need to fetch from disk, and keep fetching the other pages that pointers to into memory.

CS186 MT2

Note 10. Parallel Query Processing

def: shared nothing - every CPU has own disk + memory
def: intra-operator parallelism - make one operator run as fast as possible
def: inter-operator parallelism - make query run as fast as possible by running operators in parallel.

Partitioning

1. Range Partitioning
   - Each machine gets a certain range of values that it will store
   - PRO: good for queries that lookup on a specific key
   - PRO: used in parallel sorting and parallel sort merge join
2. Hash Partitioning
   - Each record is hashed and is sent to a machine matches that hash value.
3. Round Robin Partitioning
   - Distribute records evenly, one by one

def: Network cost - how much data we need to send over the networks

Parallel Sorting
1. Range partition the table
2. Perform local sort on each machine

Parallel Hashing
1. Hash partition the table
2. Perform local hashing on each machine

Parallel Sort Merge Join
1. Range partition each table using the same ranges on the join column
2. Perform local sort merge join on each machine

Parallel Grace Hash Join
1. Hash partition each table using the same hash function on the join column
2. Perform local grace hash join on each machine

Pipeline Breakers

Broadcast Join
When one big table and one small table, just send the small table to each machine and each machine does a local join.

Symmetric Hash Join (pipeline friendly)
1. Build two hash tables, one for each table in the join
2. When a record from R arrives, probe the hash table for S for all of the matches. When a record from S arrives, probe the hash table for R for all of the matches
3. whenever a record arrives add it to its corresponding hash table after probing the other hash table for matches

Note 11. Transactions

def: inconsistent reads - a user reads only part of what was updated
def: lost update - two users try to update the same record so one of the updates gets lost.
def: dirty reads - one user reads an update that was never committed, aborted
def: transactions - sequence of multiple actions that should be executed as a single, logical, atomic unit. Guarantee ACID properties

Atomicity - transaction ends in two ways: either commits or aborts. Atomicity means that either all actions in the Xact happen, or none
Consistency - if DB starts out consistent, ends consistent after Xact
Isolation - Execution of each Xact is isolated from that of others
Durability - if Xact commits, effect persists, must survive failures

Ensuring Isolation Property of Transactions

Serial schedule - run all operations of one transaction to complete before beginning the next transaction
↳ Not efficient, want to interleave transaction actions.

Equivalence - 1. involve same transactions
2. operations are ordered same way with same transactions
3. Each leave the database in the same state

↳ Serializable
we can ensure this #3 constraint by looking for conflicting operations
1. operations are from different ~~operations~~ transactions
2. both operate on same resource
3. atleast one is a write

- if two schedules order every pair of conflicting operations the same way, then they are output equivalent (conflict equivalent)
- conflict serializable if conflict equivalent to a serial schedule

Dependency Graph
- one Node per Xact
- Edge from T_i to T_j if:
  - an operation O_i of T_i conflicts w/ an operation O_j of T_j
  - O_i appears earlier than O_j in the schedule
- Serializable if acyclic

↗ Topological Sort to get equivalent serial schedule

Deadlock
- Waiting for each other to release

Avoidance
Set priority by its age: now - start time

Wait-Die: if T_i has higher priority, T_i waits for T_j; else T_i aborts
Wound-Wait: if T_i has higher priority, T_j aborts; else T_i waits

Detection of Deadlock
"Waits for Graph": one node per XACT and an edge from T_i to T_j if
- T_j holds a lock on resource X
- T_i tries to acquire a lock on resource X but T_j must release its lock on resource X before T_i can acquire its desired lock

- if cycle, shoot a XACT in cycle
- Start backwards, and only draw arrow if they actually ~~s~~ it

2PL
can't get locks after unlocking

S2PL: unlock after transaction

CS186 Iterators and Joins (Module 2)

mainly equijoin and natural join

Notation

[R] = # pages in R

[S] = # pages in S

|R| = # Records in R

Simple Nested Loop Join

- each Record R, get each page of S
- O(join(R,S)) where R is the outer loop: [R] + |R|[S]

Page Oriented Nested Loop Join

- for every page in R, bring in one page S
  [R] + [R][S]

Block Nested Loop Join

- B−2 pages as one block of R
  $[R] + \lceil \frac{[R]}{B-2} \rceil [S]$

Index Nested Loop Join

- If we have an index on S, just look it up
  [R] + |R| * (cost to look it up matching records in S)

Hash Join

Naiive Hash Join

Fit R into B−2 pages memory and then read in each Record of S and look it up in the hashtable. [R] + [I] I/Os

Grace Hash Join

1. Repeatedly hash R and S into B−1 buffers so that we can get partitions of ≤ B−2 pages
2. If both R and S > B−2 pages keep partitioning
3. when either R or S is small enough i.e ≤ B−2 pages, then load smaller one into mem and create hashtable, matching against other one. - dont care about final write

cost
partition: 2([R] + [S]) I/O or the partitioning cost
matching: [R] + [S]

Memory Requirements
- partitioning phase divides R into (B−1) runs of size $\lceil\frac{[R]}{B-1}\rceil$
- matching phase requires each [R]/B−1 ≤ B−2
- R < (B−1)(B−2)
  no S constraint
- Naive Join better for R < memory
- GJ better for R^2 > mem > R

Sort Merge Join

1. Sort R and sort S and then iteratively check. [R] + [S] at the last step.

Query Optimization Module 7

Selectivity Estimation - how much a query plan costs

Rules

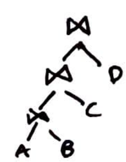
capital letter = columns

lowercase = constants

- 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)
- cond1 AND cond2: selectivity (cond 1) * selectivity (cond2)
- 1/10 default if dont have INFO
- cond1 OR cond2 = (cond1) + (cond2) − (cond1 · cond2)

Selectivity of Joins

join A and B on condition A.id = B.id
[A][B]/max(unique vals for A.id, unique vals for B.id)



Common Heuristics
1. push down projects (π) and selects (σ) as far as they go
2. only consider left deep plans
3. Dont consider cross joins unless they are the only option

Query Optimization Contd.

Pass 1 of System R

- Single table/condition operations
  - consider Fullscan − O[P] I/Os
  - consider Index Scan
    - alt 1 : (cost reach level above leaf) + num leaves read
      - can stop at condition though, so find the leaf to start at, and keep going right at leaf level because its sorted
    - alt 2/3 indices:
      (cost to reach level above leaf) + (# leaf nodes read) + (num data pages read)
      - clustered index: # data pages read is the selectivity multiplied by total # data pages
      - unclustered index: I/O per each Record so selectivity * # records

      example
      clustered index = 2 + 0.5[L] + 0.5[B]
      unclustered = 2 + 0.5[L] + 0.5|B|

Evaluating Query

- Either optimal I/O query on interesting Queries
- Interesting = sorted on a column used by GROUPBY or ORDERBY or used in a downstream join

Pass 2…n

1. At each pass i, attempt to join i tables together, each from pass i−1 and pass 1

~~Advance optimal plan for each set~~ and also the optimal plan for each interesting order for each set

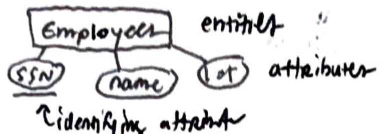
* Joined Tables must be on left

* No cross Joins are considered

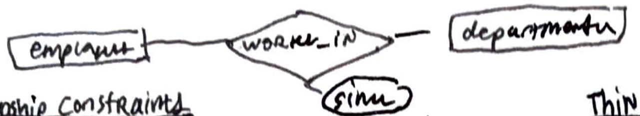
Note 12: DB Design

· when designing a database, we often use Entity-Relational Models.

def: **entity**: a real world object described by a set of attribute values



def: **Relationship**: association among two or more entities



Relationship Constraints

Thin black line - many-to-many Relationship

ex) many employees can work in many departments
many departments can have many employees

key constraint: each department has at most one manager but employee can be manager for 0 or more dept

Thin Arrow - 1 to many - at most 1, or 0, and 0 or more

Thick Line - participation: at least one!

Thick Arrow: key + participation: at least and at most one

Department ➡ Manager; each dep, one manager exactly one

Functional Dependencies and Normalization

- $X \to Y$ means X column determines Y column in a table R.
  i.e given any two tuples in table R, if their X values are the same, then their Y values must be the same
- **superkey**: set of columns that determine all the columns in the table
- **candidate key**: minimal set of columns that determine all the columns in the table
- **closure of F**: $F+$: set of all FDs that are implied by F

weak entity - entity that can be identified uniquely only w/ key of another entity (owner entity)

primary key

$X+$: set of attributes closures, set of attributes implied by X

Relation R w/ FD F

Decomposing a Relation - Boyce Codd Normal Form

· Relation R with FDs F is in BNCF if for all $X \to A$ in $F+$, $A \subseteq X$ (called trivial form) or X is a superkey for R.

The procedure:

if $X \to Y$ violates BCNF, R becomes $R-Y$ and XY

ex) Relation $R = \{C, S, J, D, P, Q, V\}$ key C and $F = \{JP \to C, SD \to P, J \to S\}$
- to deal w/ $SD \to P$, decompose into SDP, CSJDQV
- to deal w/ $J \to S$, decompose CSJDQV into JS and CJDQV

end up w/ SDP, JS, and CJDQV

Attribute Closures

A+ ~~Suppose~~ if you have $F = A \to B,\ AB \to AC,\ BC \to BD,\ DA \to C$

then A+ closure = {AB, ABC, ABCD,

Decomposition Preservation

IFF a dependency can be applied, i.e we have a set containing all the variables of a dependency