SQL

(5) SELECT [Distinct] col1, col2, agg(col3)
(1) FROM table1, table2,
(2) WHERE <predicate> AND <predicate> OR
(3) GROUP BY <column list> ← Everything in SELECT must be in GROUP-BY or is an aggregate
(4) HAVING <predicate>
(6) ORDER BY <columns> [DESC] [ASC]
(7) LIMIT <integer>;

Must contain only GROUP-BY columns or aggregate functions

No aggregates!

SQL String Comparison

- Old School SQL
  WHERE S.name LIKE 'B_%' ← returns Bob (starts with B)
  _ = any single char ; % = 0+ chars
- Standard Regular Expressions
  WHERE S.name ~ 'B.*' ← returns Bob, McBob
  . = any char ; * = repeat (0+ instances of previous)

SQL Join Variants

From table1
[INNER | NATURAL | {LEFT | RIGHT | FULL} OUTER] JOIN table2
ON <qualification list>

- INNER: join tables where "ON" qualification
- NATURAL: join tables for pairs of attributes w/ same name

FROM t1, t2 WHERE t1.id = t2.id	=	FROM t1 INNER JOIN t2 ON t1.id = t2.id

= FROM t1 NATURAL JOIN t2
(assume only matching column name = "id")

- LEFT/RIGHT/FULL OUTER JOIN
  FROM t1 [ ] OUTER JOIN
  ON t1.id = t2.id
  - LEFT: if t1.id has no match, t2.id is NULL
    t1  t2
  - Right: if t2.id has no match, t1.id is NULL
    t1  t2
  - FULL: LEFT and RIGHT both apply t1  t2

Relational Table Properties

- Schema is fixed
  - unique attribute names
  - atomic (primitive) types
- Tables are not ordered (sets, multisets)
- Tables are flat (no nested tables)
  - First Normal Form

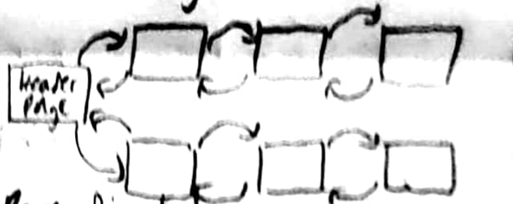
Disk Representation

- DB File: collection of pages
- Page: collection of records ← [pageID, location on page]

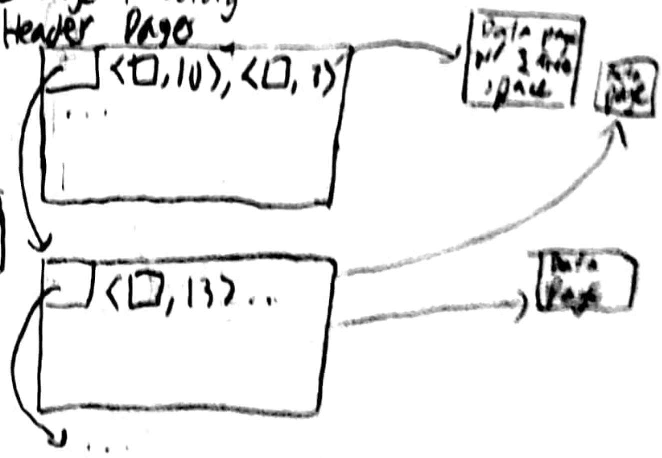
Unordered Heap File

Records placed arbitrarily across pages

1. File as Doubly Linked List



2. Page Directory
Header Pages



Unordered Heap Files vs Sorted File

2,5	1,6	4,7	3,10	8,9	vs.	1,2	3,4	5,6	7,8	9,10

	Heap	Sorted
Scan all records	B · D	B · D
Equality search	0.5 · B · D	(log B) · D
Range Search	B · D	(log B + pages) · D
Insert	2 · D	(log B + B) · D
Delete	(0.5 · B) · D	(log B + B) · D

Assume
- append page to end of heap files
- Sorted files are packed

Page Layout

- FIXED record length, UNPACKED records

Header	1	0	1	0	Rec	X	Rec	X	Rec

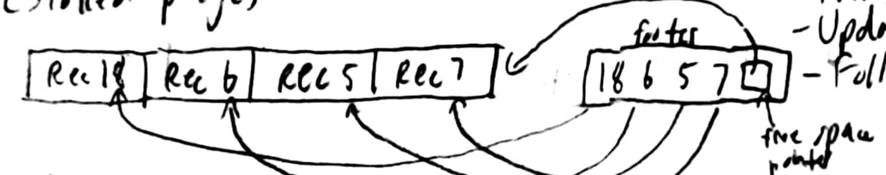
Insert: find first empty slot in bitmap :(
Delete: find record and clear bit in Header :)
- No organization needed

- FIXED record length, PACKED RECORDS

Header	Rec	Rec	Rec	Rec

Insert: just append to end :)
Delete: Scan for record, delete, reorganize all records (b/c pack) :(

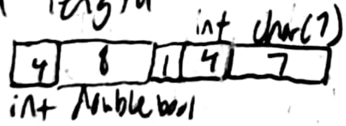
- VARIABLE record lengths, UNPACKED records (slotted page)



Delete: remove footer pointer (set record slot to null)
Insert: First empty pointer slot of free space pointer
NOTE: footer stores one int and one pointer for EACH page

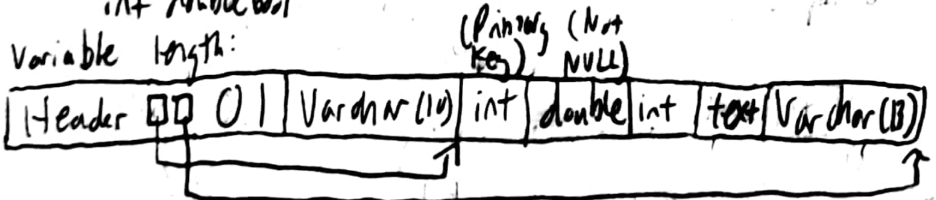
Record Length

- Fixed length:



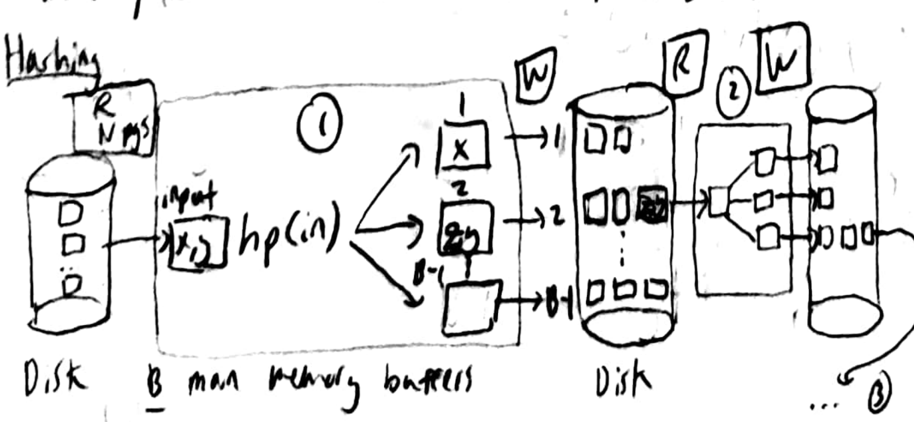
→ store 24 bytes

- Variable length:



- 2 pointers to variable length attributes
- bitmap (2 bits) where int and text are NULL

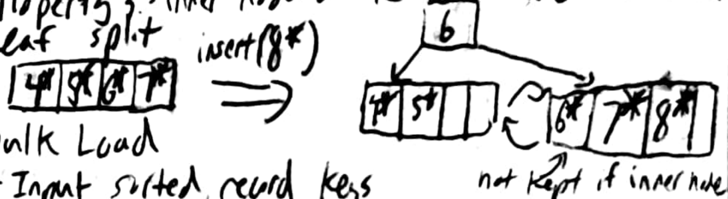
Hashing



Index Files (B+ trees) - acts similar to B-tree

- Property: # entries per node, d = order ≤ entries ≤ 2d
  # leaves = $(2d+1)^h$
- Alternative 1: record contents stored in the leaf node itself
- Alternative 2: Leaf nodes: ⟨K, rid of matching record⟩
- Alternative 3: Leaf nodes: ⟨K, list of rids of records⟩

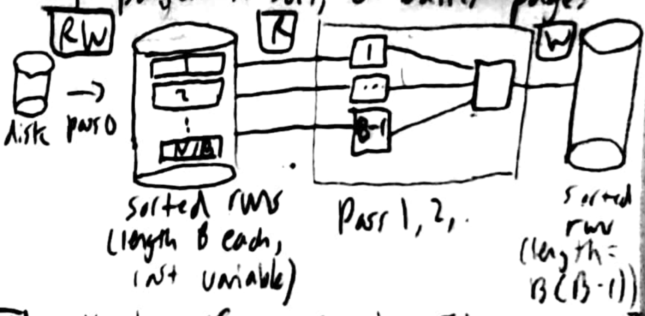
- Property 1: leaf node entries: d = order ≤ entries ≤ 2d
- Property 2: all leaves same dist. from root
- Property 3: inner node w/ K ~~items~~ have K+1 children
- Leaf split



- Bulk Load
  - Input sorted record keys (1*, 2*, 3*, ...)
  - Fill leaf page to fill factor > d
  - Update inner pages to full
  - Follow leaf and inner page split rules.

Sorting

- General External Merge Sort
  - N pages to sort, B buffer pages



- Number of passes: $1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$
- Total I/Os = (I/Os per pass) * (# passes)
  $= 2N * (1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$
- Memory: B(B-1) after 2 passes

(1) Divide: split pages into B-1 partitions using hp hash function

(2) Conquer: Repeat for big partitions (> B pages), rehash w/ new hash function h_{p_1}

(3) When partition fits in memory (<= B pages), read + write to check (Build Pass)

JOINS $[R]$ = # pages in R, p_R = # records per page in R, $|R|$ = # records ~~break~~

• **Simple Nested Loop join on θ:**

```
for record r in R:    } cost = scan R pages +
  for record s in S:  } scan S pages per record
    if θ(r,s):          = [R] + |R|[S]
      add <r,s> to result buffer
```

• **Pages Nested Loop Join**

```
for rpage in R:        } cost = [R] + [R][S]
  for spage in S:      }
    for rtuple in rpage
      for stuple in spage.
        if θ(r,s): add <r,s> result buffer
```

• **Block Nested Loop Join**

```
for rchunk of B-2 pages of R:  } cost =
  for spage of S:              } [R] + ⌈[R]/(B-2)⌉[S]
    add matching tuples to res buffer
```

• **Index Nested Loops Join**

```
for each tuple r in R:
  for each tuple s in S, r==s:
    add <r,s> to res buffer
```

cost $= [R] + |R|$ (cost to find matching tuples in S)

• **Sort-Merge Join**

① Sort R and S by join key

② Join: merge-scan sorted partitions, emit matching tuples

```
mark = NULL
while r in range:
  if not mark:
    while r<s, advance r
    while r>s, advance s
    mark = s # start of block of s
  if r==s:
    result = <r,s>
    advance s
    return result
  reset s to mark
  advance r
  mark = NULL
```

r →

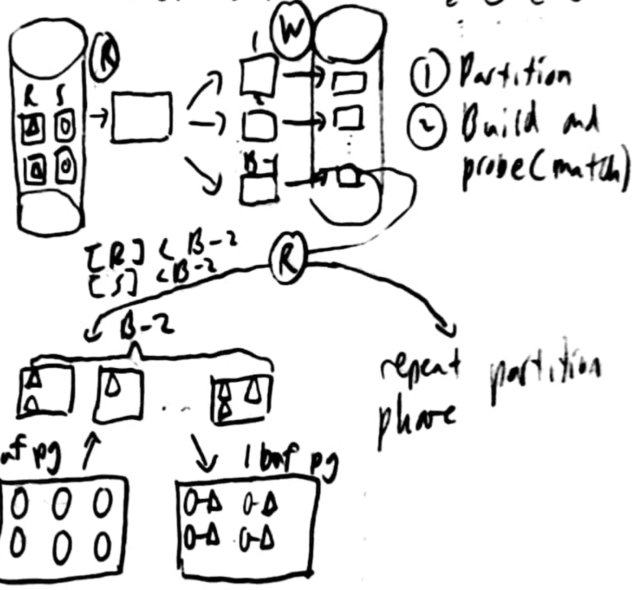
sid	sname
22	dustin
28	allen
31	john
31	joe
44	mama
58	rusty

s →

sid	bid
28	100001
28	1010
31	11010
31	10001
42	0101
58	00110

cost:
best: sort R + sort S + $[R]$ + $[S]$
worst: sort R + sort S + $|R|[S]$

• **Grace Hash Join**



Relational Algebra

Projection (π) ≡ SELECT

Selection (σ_{cond}) ≡ WHERE

Renaming ($\rho_{\text{Temp1}(R1.sid \to sid1,\ S1.sid \to sid2)}(R1 \times S1)$) — output table name; old col → new col; old table

Union ($\cup$) ≡ tuples in r1 OR r2

Set-difference ($-$) ≡ tuples in r1 but not in r2

Cross-Product ($\times$) ≡ each r1 row paired w/ each r2 row

Intersection ($\cap$) ≡ $S1 - (S1 - S2)$

Theta Join ($\bowtie_\theta$): join on logical expression θ

Natural Join ($\bowtie$) = equi-join on matching col names

Left Outer Join ($⟕$)

Right Outer Join ($⟖$)

Full Outer Join ($⟗$)

Group By ($\gamma_{age,\ count(*)>2}$) ≡ group by age, having count > 2

Iterators

• Materializing: save iterator as list on disk (costs I/O)

• Streaming: don't need to save on disk (SELECT, π)

Push down: equivalent statements

• Selection (σ)

cascade: $\sigma_{c_1 \wedge c_2 \wedge \ldots \wedge c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R))\ldots))$

commute: $\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$

• Projections

cascade: $\pi_{a_1}(R) \equiv \pi_{a_1}(\pi_{a_2}(\pi_{a_3, a_4 \ldots}(R)))$

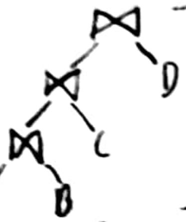
• Cartesian Product

associative: $R \times (S \times T) \equiv (R \times S) \times T$

commutative: $R \times S \equiv S \times R$

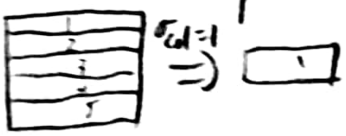
Query Optimization

① Plan Space
- Try diff queries (equivalent relational algebra)
- Try different types of joins (brace Hash, Block)
- Heuristics
  - Left-deep trees only
  - Avoid Cartesian products
  - Select and projection pushdown



$((A \bowtie B) \bowtie C) \bowtie D$

② Cost Estimation (cost = # I/O + CPU factor * # tuples)
- Catalog: containing table statistics (updated periodically)
- Selectivity (sel) = |output| / |input| of σ term
  - σ term1 $\wedge$ term2 $\wedge$ term3
  - reflects impact of σ term



, $sel = \frac{1}{5}$

Predicate	Sel	Assume	
C=V	1/ distinct C	|C|	
C=V	1/10		
C1=C2	1/MAX(distinct C1, distinct C2)	|C1|,|C2|	
C1=C2	1/distinct Ci	|Ci| only	
C1=C2	1/10		
C<V	(V-min(C))/(max(C)-min(C)+1)	max(C), min(C)	C= int
C>V	(max(C)-V)/(max(C)-min(C)+1)	max(C), min(C)	C= int
C<V	1/10		C= int
C>V	1/10		C= int
C<=V	(V-min(C))/(max(C)-min(C)+1) + $\frac{1}{\|C\|}$	max(C), min(C)	C= int
C>=V	(max(C)-V)/(max(C)-min(C)+1) + $\frac{1}{\|C\|}$	max(C), min(C)	C= int
C<=V	1/10		C= float
C>=V	1/10		C= float
C>=V	(max(C)-V)/(max(C)-min(C))	max(C), min(C)	C= float
C>=V	1/10		C= float
C<=V	(V-min(C))/(max(C)-min(C))	max(C), min(C)	C= float
C<=V	1/10		C= float
$P_1 \cap P_2$	$S(P_1) * S(P_2)$	independent terms	
$P_1 \cup P_2$	$S(P_1) + S(P_2) - S(P_1) * S(P_2)$	independent terms	
NOT P	1 - S(P)		

Lossless: can we construct original relation?
- Decompose R into A and B
  Decomp is lossless iff F^+ contains:
  $X \cap Y \rightarrow X$
  $X \cap Y \rightarrow Y$
  BCNF = always lossless

③ Search Algorithm (best joins)

pass 1: find minimum cost access method (index scan, full scan, etc.) for each table, and interesting order on interesting index

pass i:
- consider only: left deep and not Cartesian product (unless all are cartesian)
- advance only: cheapest cost plan every subset group of relations and interesting order

Interesting Orders
- ORDER BY attributes
- GROUP BY attributes
- downstream join attributes

Functional Dependencies
- FD: $X \rightarrow Y$ (X determines Y)
- Superkey: X is a superkey of R if $X \rightarrow$ [all attributes of R]
- Candidate key: minimal super key of R (no subset is a superkey)
- Armstrong's Axioms (X,Y,Z are attribute sets)
  - Reflexivity: if $Y \subseteq X$, $X \rightarrow Y$
  - Augmentation: if $X \rightarrow Y$, $XZ \rightarrow YZ$ (not other way)
  - Transitivity: if $X \rightarrow Y$, and $Y \rightarrow Z$, then $X \rightarrow Z$
  - Union: if $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
  - Decomposition: if $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$
- Closure: set of FDs F^+ s.t. $F \rightarrow F^+$
  - ex. $\{A \rightarrow B, A \rightarrow C\} \rightarrow \{A \rightarrow B, A \rightarrow C, A \rightarrow CB\}$
- Attribute closure: set of attributes X^+ s.t. $X \rightarrow X^+$ is in F^+
- Closure algorithm: To find X^+

```
closure = X
repeat until closure does not change:
    for U->V in F:
        if U ⊆ closure: closure = closure ∪ V
```

Normalization: BCNF Decomposition
- R is in BCNF if: for every FD $X \rightarrow A$ in R, $A \subseteq X$ or X is superkey
- Decompose relation R not in BCNF into multiple relations in BCNF:

```
for each X->Y in F+:
    if X->Y violates BCNF:
        Decompose R into (R-X+) ∪ X, X+
```

Transactions and Concurrency

Transactions follow rules

- **Atomicity**: All operations in a transaction happens or no ops happen
- **Consistency**: Data starts and ends consistent
- **Isolation**: Execution of txns looks like running one at a time
- **Durability**: if txn commits, effects persist

Non-conflict swapping:

if two consecutive operations from two txns can be swapped if non-conflicting.

T1: R(A) ... W(A)} reversed too OR W(A) ... W(A)

T2: ... W(A)

conflicting

Conflict equivalent: Schedules w/ same txn operations, different orderings (non-conflict swaps)

Serializable: a schedule S is equivalent to a serial schedule

Conflict Serializable: schedule S is conflict serializable to some other serial schedule (CS → Serial)

Dependency graph:

- One node per transaction
- If an operation in T_i conflicts w/ an operation in T_j and T_i comes first, add edge from T_i to T_j.
- Schedule is conflict serializable IFF dependency graph is acyclic

T_1	R(A) R(B)*		R(A)
T_2		R(B) W(B)*	

$T_1 \rightarrow T_2$

View Serializable: same as conflict serializable but calls W(A) ... W(A) R(A) ... W(A) non-conflicting, only is conflicting

- CS → VS

Simple Locking

	S	X
S	✓	X
X	X	X

on a resource

- An S lock lets a txn read a resource
  - many txns can hold S locks on a resource at once
- An X lock lets a txn modify a resource
  - No other txn can have any type of lock while a txn has X lock
- If a txn can't get a lock it wants, it blocks and waits until another txn releases the conflicting lock

Deadlocks

T_1: R(A), W(B) } T_1 waits for T_2's lock on B, while T_2 waits for T_1's lock on A

T_2: R(B), W(A) }

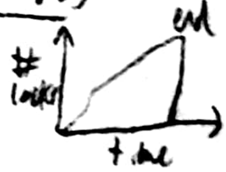
- Priority: older txns = higher priority
- **wait-die**: if T_i wants T_j's lock
  - if T_i higher priority, wait for T_j to release
  - if T_i lower priority, T_i aborts
- **wound-wait**: if T_i wants T_j's lock
  - if T_i higher priority, T_j aborts ("wound")
  - if T_i lower priority, T_j waits
- "waits-for" graph
  - One node for each txn
  - If T_j waits for T_i edge from T_j to T_i
  - Cycle means deadlock

2 Phase Locking (2PL)

- A txn may not acquire a lock after releasing any lock
- Conflict Serializable guaranteed

Strict 2-Phase Locking (Strict 2PL)

- Same as 2PL except all locks released at end of txn
- Avoids cascading aborts



Multi-granularity Locking

	NL	IS	IX	SIX	S	X
NL	✓	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	✓	X
IX	✓	✓	✓	X	X	X
SIX	✓	✓	X	X	X	X
S	✓	✓	X	X	X	X
X	✓	X	X	X	X	X

- To get S or IS lock on a node, need IS or IX on parent node
- To get X, IX, or SIX on a node, need IX or SIX on parent node
- Assume Strict 2PL unless interfere w/ txn's new operation

Recovery

STEAL: txns can flush pages w/ uncommitted updates
- Pro: can maximize use of buffer pages
- Con: Must undo if program crashes before commit

FORCE: force dirty pages to disk before commit

NO-STEAL: txns cannot flush pages w/ uncommitted changes

NO-FORCE: dirty pages allowed to not be forced on disk upon commit

Write ahead logging
- txn not committed until log records for changes written to disk
- Changes must be logged before data modified on disk

Undo Logging (Atomicity)
⟨Start T⟩ ⟨Commit T⟩ ⟨Abort T⟩ ⟨T, X, v⟩ (X: updated element (dirty page), v: old value)
- If T commits, then FLUSH(X) must be written to disk before ⟨Commit T⟩
- If T modifies X, then ⟨T, X, v⟩ written to log before FLUSH(X)

Redo Logging (Durability)
⟨Start T⟩ ⟨Commit T⟩ ⟨Abort T⟩ ⟨T, X, v⟩ (X: updated element (dirty page), v: new value)
- If T modifies X, both ⟨T, X, v⟩ and ⟨Commit T⟩ must be logged before FLUSH(X)

Undo	Redo
Flush(X)	Commit(txn)
Flush(Y)	Flush(X)
Commit(txn)	Flush(Y)

ARIES (STEAL, NO FORCE)
- Writes
  - UPDATE log record for every write
- Commit:
  - Write COMMIT log record, flush log to disk, "committed" is status, release txn locks, write END log record, remove from txn table
- Abort (see UNDO)
  - write ABORT log record, write CLR record for each UPDATE the txn logged in reverse order, release txn locks, write END log record, remove from txn table
- Assume strict 2-phase locking and write-ahead logging
- Data structures
  - Transaction Table: information on currently active transactions (running, committing, aborting)
  - Dirty page table: dirty pages in memory not yet written to memory

ATT

Txn	LastLSN	Status
T1	10	Running
T2	20	Committing
T3	40	Running
T4	50	Aborting

DPT

Page	recLSN
P1	0
P3	10

(1) Analysis: identify changes not written to disk and active txns at time of crash
- Start at last BEGIN-CHECKPOINT and scan all following log records
- For each record:
  - if record is UPDATE/CLR:
    - if txn not in ATT: add it (Txn ID, Status, lastLSN)
    - if txn in ATT: update lastLSN
  - if record is ABORT/COMMIT: update ATT (Status, lastLSN)
  - if record is END: remove transaction from ATT

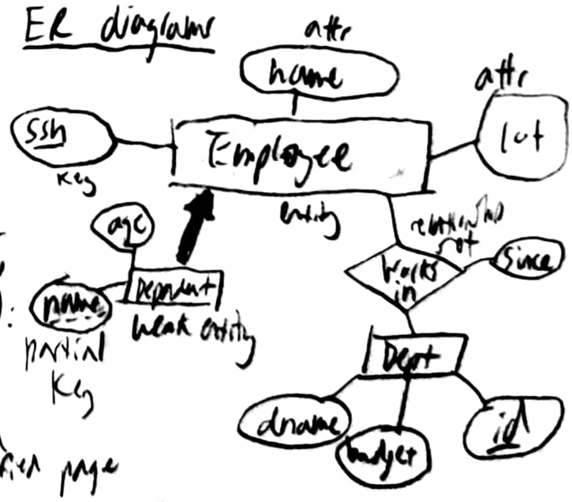
(2) REDO: Repeat all actions to restore database state to what it was at the time of the crash from the smallest recLSN
- For each UPDATE/CLR beginning from min(all recLSN in DPT): REDO operation UNLESS any of the following is true:
  - affected page not in DPT
  - recLSN of page in DPT > LSN
  - pageLSN of page (in DB disk) ≥ LSN (pageLSN = LSN of last record that modified page)

(3) UNDO: undo the actions of transactions that did not commit
- For all "loser" (active) transactions:

```
undoLSN = lastLSN
while undoLSN != NULL:
    if record at undoLSN is UPDATE:
        undo record w/ undoNextLSN = record's prevLSN
    undoLSN = prevLSN
add END record for this txn to CLR
```

(abort) undo = add record to CLR

ER diagrams



- Key constraint → participates in at most one
- Participation constraint participates in at least one
- Key constraint with total participation exactly one
- Non-key partial participation 0 or more