Storing Fixed Length Records

(1) Packed — no gaps

page { Pg. Header | Record → | Record | Record | Record → | Record | Record → | Record → }

(2) Unpacked — Bitmap

encodes gaps

Page Header | Record 1 | Record 2 | → Record 3 → | slot 4 | slot 5 | Record 6 | slot 7 |] Bitmap

Variable Length Records

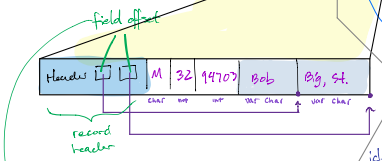
• 2 types of records:

(1) Delimit fields — use special character

F1 | $ | F2 | $ | F3 | $ | F4

downsides): what if records contain $

(2) Array of Field Offsets



• variable length fields placed after fixed field lengths

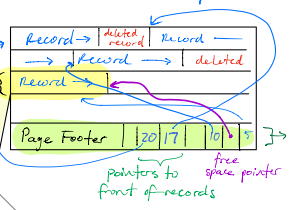
field offset — NOTE: not sufficient for NULL values; • cannot distinguish b/w empty string & NULL

indicate where each variable length field starts

Storing Variable Length Records

• slotted pages

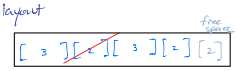
→ use slot directory



• page footer
  • can grow in size
  • store length and pointers to a record
  • store number of slots
  • free space pointer

VLR: Fragmentation

idea): Deleting records causes fragmentation when using unpacked layout



• if 2 byte record is deleted then 4 bytes free will remain

→ problem): cannot insert 4 byte record

Q): How to handle fragmentation?

• always add beginning from free space pointer — deleting doesn't matter for adding new records

→ NULL out field in slot directory of deleted field

• ran out of space for new record → load new page

File Organization

(1) Sorted File

• sorted on a key

(2) Heap File — records will be stored arbitrarily

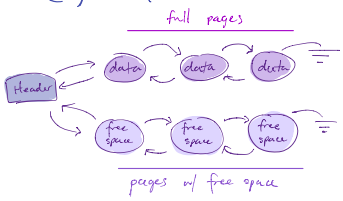
• no order enforced within pages

purpose): keep track of pages

Record ID (RID) : (page ID, slot #)

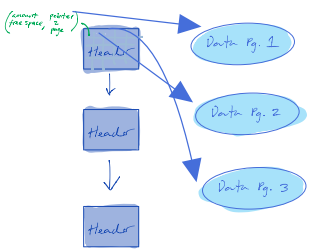
Heap File Implementation

(1) Linked List



(2) Page Directory

idea): linked list of header pages each with a set of pages



Page Replacement Policies

(1) Least recently used (LRU)

• evict page we haven't used in a long time

• on use → pinned — pinned pages cannot be evicted

• track time each frame was last unpinned

→ replace frame that least recently unpinned

pros): • perfect when need repeated access to popular pages | "temporal locality"

cons): • costly, must maintain heap data structure → can use clock method

• repeated sequential scans of large files

(2) most recently used (MRU)

• evict most recently used page

pros): • better performance when conducting sequential scan of large file

B+ Trees

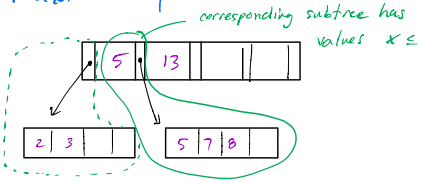
• d: order of the tree

2d: max. values per node

inner nodes

• up to 2d values

• maximum fanout: 2d + 1



reference to search key value less than smallest value (5)

• occupancy invariant — • ensures fanout is high • performance enhancement

• every inner node must be at least half full

• maximum # entries

(2d)(2d+1)^h — current # entries

maximum capacity for order d, height h

Bulk Loading

goal): Construct B+ tree from scratch

algorithm:

1) Sort data on key we are indexing on

2) Fill leaf pages until some fill factor f

3) Add pointer from parent → leaf. If parent overflows:

(a) Keep d entries in L_1, d+1 in L_2

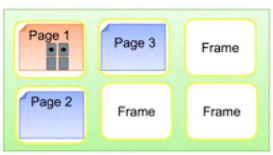
(b) MOVE L_2's first entry into parent

4) Adjust pointers

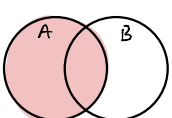
When a page is requested...

Buffer pool information "table" contains: <frame#, pageid, pin_count, dirty>

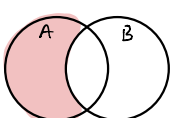
1. If requested page is not in buffer pool:
   a. Choose an **unpinned** frame for *replacement*
   b. If frame "dirty", write current page to disk
   c. Read requested page into frame
2. *Pin* the page (pin_count++, page is in use) and return its address
3. Once we're done using the page, *Unpin* its frame in the buffer pool (pin_count--)



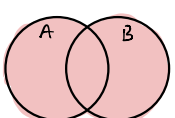
SQL



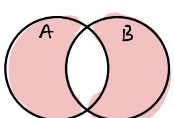
left join



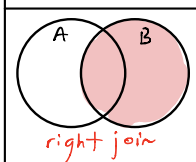
left join where B = null



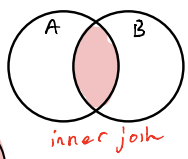
full outer join



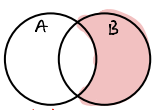
full outer join where B or A = Null



right join



inner join



right join where A = null

Other SQL Keywords To Know

- UNION
- INTERSECT
- IN — specify more than 1 value for WHERE
- EXISTS — test existence of tuple ∈ subquery
- COUNT
- ALL — true if all subquery value meet cond.
- ANY — true if any subquery value meets cond
- NOT

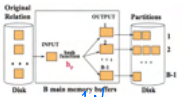
Relational Algebra

* — requires both relations to have same schema

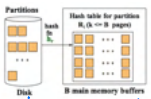
Operator	Type	SQL	Description
π	unary	SELECT	• vertical operator • select attributes (cols) set semantics: • may lose tuples if we remove the only column two tuples differ in
σ	unary	WHERE	• horizontal operator • select subset of tuples satisfying some predicate
ρ	unary	AS	• renames attributes
$\cup$*	binary	UNION	• returns relations containing tuples from both relations • lose tuple if found in both relations
$-$*	binary	EXCEPT	• returns relation containing all tuples from L not found in R
$\times$	binary	SELECT ... from t1, t2	• match each tuple $\in$ L w/ each tuple $\in$ R
$\cap$*	compound	INTERSECT	• intersection of both relations
$\bowtie_\theta$	compound	INNER JOIN	• inner join on θ
$\bowtie$	compound	NATURAL JOIN	• natural join of L & R on every column w/ same name
γ	extension	• GROUP BY • HAVING	• groups by column and aggregates by specified agg.

External Hashing

- use hash function h_p
  - ➡ partition the data
  - stream partitions to disk



- if B $\in$ buffer:
  - split data into B−1 partitions



- if $n_{partitions} \leq B$
- load them in → make in-memory hash table for each partition

→ can apply duplicate removal, agg., etc.

- in-memory hash table must use separate hash h_r

Q): partitions too big after first pass?

A): recursive partitioning

- for each partition, apply new hash h_n
  - → splits partitions into new smaller ones
- repeat until partitions fit in memory
  - → every hash must be new

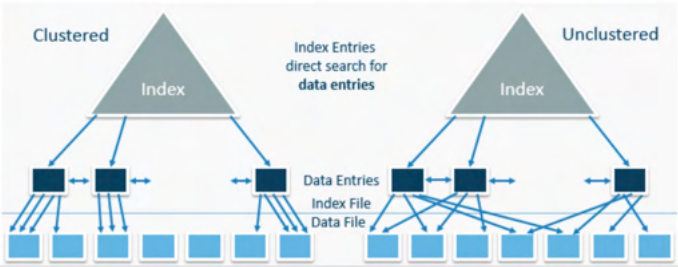
con): not good if more than B pages of duplicates

How data is stored in the index

- **k:** the search key, (a subset of the table's columns)
  - e.g. date of birth, or (lastname, firstname), etc.
- Three alternatives:
  - **Alt 1. By value:** actual data record (with key value **k**)
  - **Alt 2. By reference:** <**k**, rid of matching data record>
  - **Alt 3. By list of refs.:** <**k**, list of rids of *all* matching data records>

Clustering Overview

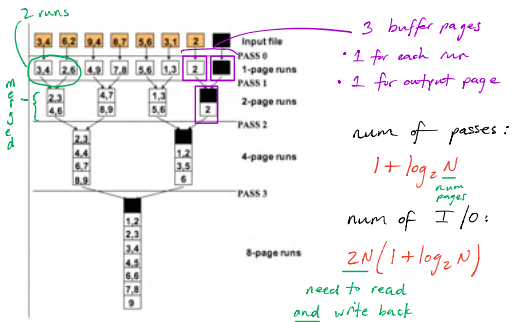
- **Clustered**: data records are stored in approximate order by key
- **Unclustered**: data records are in any order, not sorted by key



- efficient range searches
- potential locality benefits
- expesive to maintain

Sorting

- 2-way external merge
  - sort small amounts of data into runs of sorted tuples
  - given 2 runs of sorted tuples
    - ➡ merge to form 1 larger run



3 buffer pages
- 1 for each run
- 1 for output page

num of passes: $1 + \log_2 \frac{N}{\text{num pages}}$

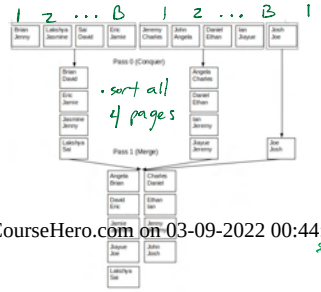
num of I/O: $2N(1 + \log_2 N)$

need to read and write back every page every pass

- full external merge sort
  - assume B buffer pages
  - load B pages, sort all at once
  - merge B − 1 runs at once (min PQ DS)

look at first tuple of each run, output tuple w/ lowest value

B = 4



- sort all 4 pages

num of passes: $1 + \log_{B-1}\left(\frac{N}{B}\right)$

num runs after pass 0

num of I/Os: $2N\left(1 + \log_{B-1}\left(\frac{N}{B}\right)\right)$

same principle as above

SNLJ $[R] + p_R[R][S]$

PNLJ $[R] + [R][S]$

BNLJ $[R] + \left\lceil \frac{[R]}{B-2} \right\rceil [S]$

INLJ $[R] + |R| \cdot$ (cost to find matching tuples $\in S$)

differs based on alt.

alt 1 cost: cost to traverse from root → leaf + read all leaves w/ matching tuples

alt 2/3 cost: cost of retrieving RIDs + cost of fetching actual records

- clustered: 1 IO per page
- unclustered: 1 IO per record

SMJ $cost(sort(R)) + cost(sort(S)) + [R] + [S]$

merge step

$cost(sort(R)) = 2 \cdot [R] \cdot$ # passes

possible optimization iff (must satisfy cond. for optimization): (# runs in last pass for R + # runs in last pass for S) $\leq B - 1$:

▶ reduces I/o cost by $2([R]+[S])$

GHJ $cost(hashing) + cost($Naive Hash Join on subsections$)$

cost of reading in each page in both partitions

$cost(hash(R))$: IOs needed to read all pgs in R + IOs needed to write all resulting partitions

Selectivity Estimation - Connectives

Predicate	Selectivity	Assumption
p1 **AND** p2	S(p1)*S(p2)	Independent predicates
p1 **OR** p2	S(p1) + S(p2) - S(p1)*S(p2)	Independent predicates
NOT p	1 - S(p)	

Selectivity Estimation - Inequalities on Floats

Predicate	Selectivity	Assumption
c >= v	(max(c) - v) / (max(c) - min(c))	We know max(c) and min(c). c is a float.
c > v	1 / 10	We don't know max(c) and min(c). c is a float.
c <= v	(v - min(c)) / (max(c) - min(c))	We know max(c) and min(c). c is a float.
c < v	1 / 10	We don't know max(c) and min(c). c is a float.

* We don't add 1 to the denominator. floats are continuous, integers are discrete)
E.g. range [2.0, 4.0] = 2.0, 2.1, ..., 3.9, 4.0 → 4.0 - 2.0 = 2.0

Selectivity Estimation - Inequalities on Integers

Predicate	Selectivity	Assumption
c < v c > v	(v - min(c)) / (max(c) - min(c) + 1) (max(c) - v) / (max(c) - min(c) + 1)	We know max(c) and min(c). c is an integer.
c < v c > v	1 / 10	We don't know max(c) and min(c). c is an integer.
c <= v c >= v	(v - min(c)) / (max(c) - min(c) + 1) + (1 / |c|) (max(c) - v) / (max(c) - min(c) + 1) + (1 / |c|)	We know max(c) and min(c). c is an integer.
c <= v c >= v	1 / 10	We don't know max(c) and min(c). c is an integer.

* We add 1 to the denominator in order for our [low, high] range to be inclusive.
E.g. range [2, 4] = 2, 3, 4 → (4 - 2) + 1 = 3

Selectivity Estimation - Equalities

Predicate	Selectivity	Assumption
c = v	1 / (number of distinct values of c in index)	We know |c|.
c = v	1 / 10	We don't know |c|.
c1 = c2	1 / MAX(number of distinct values of c1, number of distinct values of c2)	We know |c1| and |c2|.
c1 = c2	1 / (number of distinct values of ci)	We know |ci| but not |other column|.
c1 = c2	1 / 10	We don't know |c1| or |c2|.

- **|column|** = the number of distinct values for the column
- If you have an index on the column, you can assume you know |column|, max(c), and min(c)
- When applying selectivity to # of records, take the floor of the result. (e.g. 256.3 → 256 records)

Transactions

- collections of operations
  - a single logical unit
  - would like all components to happen at once

- commit
  - save changes
  - successful tx
- abort
  - reverts changes
  - unsuccessful tx

- should follow ACID properties

property	description	associated topic
Atomicity	- all operations ∈ tx happen - all or none	• recovery
Consistency	• database constraints (i.e: unique values) are maintained	
Isolation	• should look like we only run one transaction at a time	• concurrency • locking
Durability	• once a transaction commits, it should persist	• recovery

Concurrency

at same time

- benefits of running many txs concurrently:

(a) throughput argument

- increases processor/disk utilization: work on another tx while one is busy waiting for data from disk

▶ more transactions per second

(b) latency argument

- run a lot of txs at once then a long running tx doesn't cause short tx to wait long time to start

▶ transactions may finish faster

note): still must maintain ISOLATION

Conflict Serializability

Q): How to check if schedule is serializable?

A): you don't

▶ instead, check if schedules are conflict serializable

- stronger condition than serializability

- two operations ∈ schedule conflict if (R(A) or W(A)):
  - at least one operation is a WRITE
  - they are on diff. txs
  - they work on same resource

Q): Check for conflict equivalence/serializability?

A): Dependency graph (precedence graph)

- if all arrows are in same direction in dependency graph for e/a schedule

▶ two schedules are conflict equivalent

theorem): All conflict serializable schedules have an acyclic dependency graph

▶ check for cycle in dependency graph

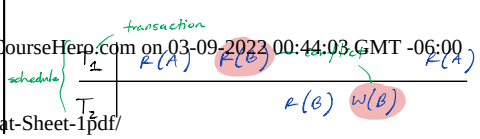
- if cycle ▶ not conflict serializable
- else ▶ conflict serializable

- two schedules are conflict equivalent if
  - every conflict is ordered the same way

- a schedule is conflict serializable if:
  - it is conflict equivalent to a serial schedule

Serializability Hierarchy

Dependency Graphs (precedence)



R(B) ∈ T_1 comes first so T_1 has edge to T_2

one node per transaction

Query Optimizer: Selinger

plan space

- avoid cartesian products at all times
- only left-deep trees: represent a plan where all new tables are joined one at a time from the right

cost estimation

- only incorporate I/O cost

search algorithm

- dynamic programming

interesting orderings

- orderings on intermediate relations that may help reduce cost of later joins
  - GROUP BY attributes
  - ORDER BY attributes
  - downstream join attributes

Deadlocks

method	description	approaches
Deadlock avoidance	avoid them at all costs	• wait-die • wound-wait
Deadlock detection	catch them and abort tx in deadlock	• "waits for" graph • kill inactive txs

Deadlock Avoidance

scenario): T_i wants lock but T_j has conflicting lock

1. Wait-Die
   - T_i has higher priority → waits for T_j to release lock
   - T_i has lower priority → abort T_i (dies)
   - txs can only wait on lower priority txs
     → lowest priority transactions cannot wait
2. Wound-Wait
   - T_i has higher priority → cause T_j to abort (wound)
   - T_i has lower priority → waits for T_j to finish
   - txs can only wait on higher priority txs
     → highest priority txs cannot wait

2-Phase Locking (2PL)

idea): enforces conflict serializability

- method
  - tx may not acquire a lock after it has released any locks
  - two phases
    - Phase 1: start until some lock is released; tx is only acquiring locks
    - Phase 2: until end of tx; tx is only releasing locks

problem): doesn't protect from cascading aborts

- if some earlier tx T_1 aborts in the end, it will affect all other txs
  → all txs have to be rolled back

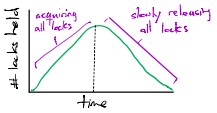
why? it allows a new tx to read new values before the transaction commits

Strict 2PL

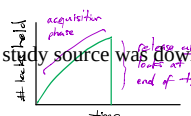
idea): avoid cascading aborts

- method
  - same as 2PL except all locks get released together when a tx completes

2PL

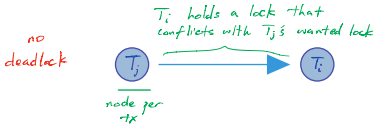


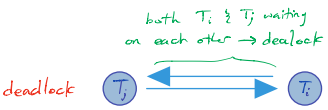
Strict 2PL



Deadlock Detection

- "waits-for" graph





- cycle ∈ graph → deadlock
- abort one of T_i, T_j to end deadlock

Undo Logging

- Write log records to ensure **Atomicity** after a system crash:
  - **<START T>**: transaction T has begun
  - **<COMMIT T>**: T has committed
  - **<ABORT T>**: T has aborted
  - **<T,X,v>**: T has updated element X, and its old value was v
- If T commits, then **FLUSH(X)** must be written to disk before **<COMMIT T>**
  - **Force** – we can UNDO any modifications if a Xact crashes before COMMIT
- If T modifies X, then **<T,X,v>** log entry must be written to disk before **FLUSH(X)**
  - **Steal** – we can UNDO any modifications if a Xact crashes before FLUSH

Undo/Redo Logging

- UNDO logging provides **atomicity**
  - Undoes all updates for ***running*** transactions
- REDO logging provides **durability**
  - Redoes all updates for ***committed*** transactions
- Requires us to recover from the beginning of the log!
  - Computationally expensive
- What if we could start mid-way through the log?
- What if we could get the best of both? **Steal / No-Force**

Write-Ahead Logging

- Two Rules:
  - A transaction is not committed until the log records for all its changes have been written to disk
  - Changes must be logged before the actual data is modified on disk
- With these two rules we can develop a system that recovers properly from failures

Redo Logging

- Write log records to ensure **Durability** after a system crash:
  - **<START T>**: transaction T has begun
  - **<COMMIT T>**: T has committed
  - **<ABORT T>**: T has aborted
  - **<T,X,v>**: *T has updated element X, and its new value was v*

  } Same as in Undo Logging
- If T modifies X, then *both* **<T,X,v>** and **<COMMIT T>** must be written to disk before **FLUSH(X)**
  - **No-Steal, No-Force** – we can REDO any modifications if a Xact

Force/No-force

Whether or not modified pages from a transaction are forced to disk on commit

- **Force** = Enforce
  - Easy to maintain durability
  - If we crash after committing, everything was already written to disk!
- **No-Force** = Don't Enforce
  - Faster, more difficult to maintain durability
  - Don't have to do unnecessary writes
  - If DB crashes mid-transaction, we'll need to redo any changes that didn't make it to disk

No-Steal, Force Example

T_1	Write(A)	Write(B)			*CRASH*	Commit
T_2			Write(B)	Commit		

- **No-Steal** = Don't Allow modified pages to be flushed to disk before Commit
  Force = Force modified pages to be flushed to disk on Commit
  - T_1: page A will not be modified on disk. T_1 crashed before Commit (*No-steal*)
  - T_2: page B is modified on disk. (*Force*)
- Durability of database maintained. Committed Transactions have their modifications flushed to disk.
- Atomicity is not maintained. T_1 didn't commit, but it's changes to page B still made it to disk!

Steal, No-Force Example

T_1	Write(A)	Write(C)	*Flush to disk*		*CRASH*	Commit
T_2			Write(B)	Commit		

- **Steal** = Allow modified pages to be flushed to disk before Commit,
  No-Force = Do not force modified pages to be flushed to disk on Commit
  - T_1: Pages A, C are modified on disk. (*Steal*)
  - T_2: page B not modified on disk (*No-force*)
- Without additional policies in place, Atomicity and Durability are *not* maintained:
  T_1's changes are flushed to disk despite not committing,
  T_2's changes are not flushed to disk despite committing.

Multigranularity Lock Compatibility Matrix

	NL	IS	IX	SIX	S	X
NL	•	•	•	•	•	•
IS	•	•	•	•	•	no
IX	•	•	•	no	no	no
SIX	•	•	no	no	no	no
S	•	•	no	no	•	no
X	•	no	no	no	no	no

Steal/No-steal

Whether or not modified pages from an uncommitted xact can be flushed to disk

- **No-Steal** = Don't Allow
  - Easy to maintain atomicity
  - If we crash mid-transaction, none of the changes were written to disk!
- **Steal** = Allow
  - Avoids some problems
  - Allows buffer manager to use frames optimally
  - Allows uncommitted changes to make it to disk so we may need to undo these

Parallel Query Processing

Types of Query Parallelism

- inter-query parallelism
  - works using queries as the unit of parallelism
- intra-query parallelism
  - works within a single query

Types of Intra-Query Parallelism

- inter-operator
  - works using operators as units of parallelism
- intra-operator
  - works within a single operator

Inter-Operator Parallelism

- pipeline parallelism
  - inter-operator parallelism on operators in a pipeline
- bushy tree parallelism
  - inter-operator parallelism on operators that do not depend on each other

Intra-Operator Parallelism

- partition parallelism
  - partitions the data
  - operates on partitions in parallel

Parallel Sorting

1. range partitioning
   - distribute data across machines

Parallel Sort Merge Join

1. range partitioning of both relations
   - use same range for both relations
2. sort-merge join on e/a machine independently
2. external sorting on e/a machine independently
   - each machine holds diff. range of data

Partoning Data

- range partitioning
  - on a key
  - divides data based on which range the key falls into
- hash partitioning
  - uses hash function
- round-robin partitioning
  - cycles through partition in order as the data comes in
  - not ordered by key

Parallel Hashing

1. use hash partitioning
2. run external hashing on each machine individually and independently
   → similar to recursive hashing

Parallel Hash Joins

1. use hash partitioning on both relations
   ↳ then perform normal hash join on e/a machine independently

Phase 1
- partition R & S across diff. machines

Phase 2
- within each machine, perform local grace hash joins

Symmetric Hash Joins

- streaming hash join algorithm
  - does not require all tuples to be available before starting
    - inefficient to wait all available
    - partitions may come from multiple machines
  - not writing to disk
  - requires R & S partitions to fit in memory
    - get more machines to make partitions small

idea): Build two hash tables at same time

- tuple from R arrives
  - add to R's hash table
  - probe hash table for S
- tuple from S arrives
  - add to S's hash table
  - probe R's hash table

→ generates each joined tuple once output generated when both parts arrive

Parallel Aggregation

goal): calculate SUM, COUNT, other aggregation fncs.

method): Hierarchical aggregation

- decompose aggregate into two parts

(1) Global
- apply on local agg. values to get one final agg. value

(2) Local
- apply on each machine independently

EXAMPLE: AVERAGE(col)

(1) Global
- get e/a machine sums and counts
- combine all of them, compute final product

$$\Sigma\left(\frac{\Sigma(col)}{\Sigma(count(col))}\right)$$

(2) Local
- return sum $\Sigma(col)$
- return count $Count(col)$

Assymetric Shuffles

- when data is already partitioned the desired way

Broadcast Joins

- sometimes one relation ⇒ HUGE other relation ⇒ small
- much cheaper to send all of tiny tables multiple times than partitioning huge table

General Formula

best case for external sort

passes to sort e/a partition

$$1 + \left\lceil 1 + \log_{B-1}\left(\frac{N}{MB}\right)\right\rceil$$

↑ partition data

Constraints

- key constraint
  - at most one
- participation constraint
  - at least one
- key constraint w/ total participation
  - exactly one

Functional Dependencies

- functional dependency
- set of attributes X → Y
- if attributes in X match, then attributes in Y
- can not be inferred from the data

Superkey

- X is a superkey of R if:

X → [all attributes of R]

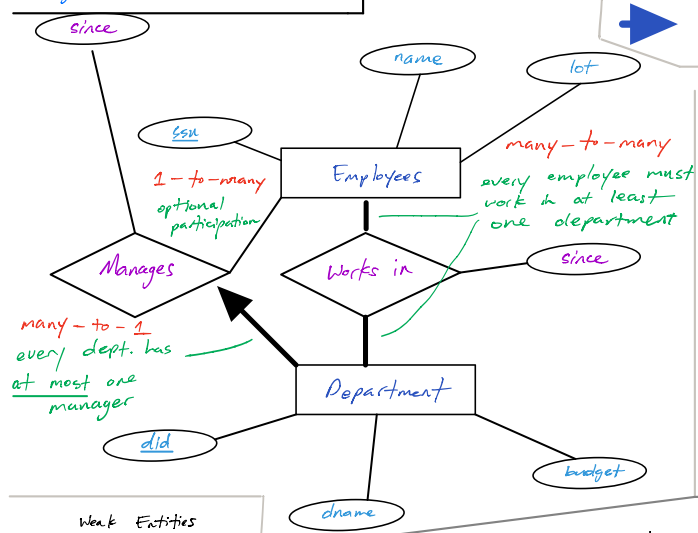
idea): can use other columns to infer other columns

→ eliminate redundancy

- candidate key
  - a minimal superkey
  - has no columns that can be removed & still be a superkey

Closure

- the closure of a set of FDs F is F^+
  - set of all FDs implied by F
  - hard to find
  - exponential in # of attributes
  - ▶ use attribute closure
- the attribute closure of X given a set of FDs is X^+
  - set of all attributes A such that $(X \to A) \in F^+$
  - set of all attributes that can be determined by X



Anomaly	Description
update	• if data changed for one person, must change for ALL
insert	• if want to insert data point w/ rating = 10, must figure out wage associated with it
delete	• if delete all employees with rating = 8, will not know wage value associated with it

Normalization

Boyce-Codd Normal Form (BCNF)

- R is in BCNF if:
  - for every FD $X \to A$ that holds over R, either $A \subseteq X$ or X is a superkey
- no redundancy in BCNF
  - every field of every tuple contains some information that cannot be inferred from FDs

Decomposition: Problems

- lossless
  - may not be able to reconstruct original relation
    - generates bad data
    - data not actually lost
- decompose R into X & Y
  - lossless iff F^+ contains:

    X intersect Y → X or
    X intersect Y → Y
- BCNF is always lossless

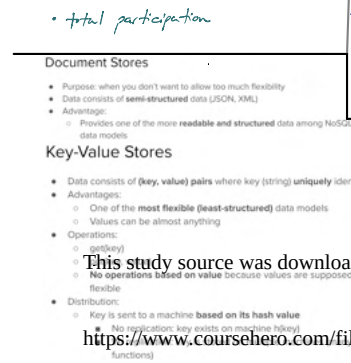
Weak Entities

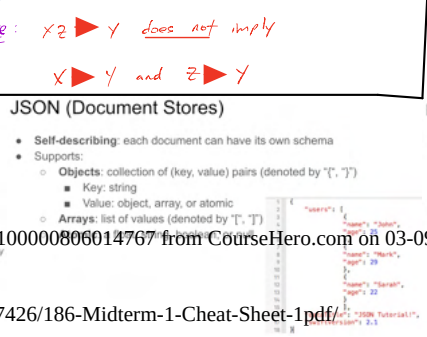
- weak entity
  - entity that can be identified uniquely only with the key of another entity
  - have a partial key (will be a dashed underline)
    - identifies the weak entity when combined w/ owner entity's key
  - must be a 1-to-many relationship
    - 1 owner entity → many weak entities
  - total participation

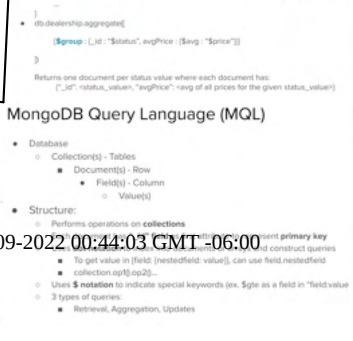
Inference Rules

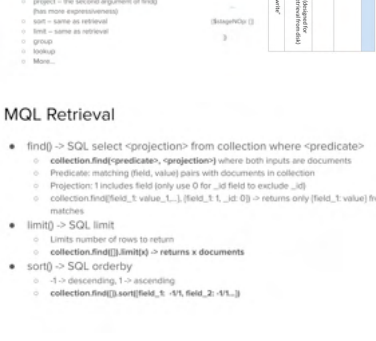
- Armstrong's Axioms:
  - reflexivity: if $Y \subseteq X$, then $X \to Y$
  - augmentation: if $X \to Y$, then $XZ \to YZ$
  - transitivity: if $X \to Y$ and $Y \to Z$ then $X \to Z$
- union: if $X \to Y$ and $X \to Z$, then $X \to YZ$
- decomposition: if $X \to YZ$, then $X \to Y$ and $X \to Z$

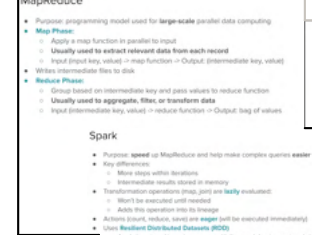
note: $XZ \to Y$ does not imply $X \to Y$ and $Z \to Y$

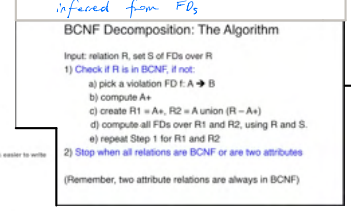












Two-Phase Commit Logging

(1) Voting Phase
- coordinator → participants: prepare message
- participants
  - log & flush either a PREPARE or ABORT log record
    - log entry keeps track of coordinator ID
    - if participant votes NO
      - > transaction will abort
      - > begin cleaning up transaction
- participants → coordinator: vote YES or NO
- coordinator
  - log & flush either a COMMIT or ABORT log record
    - commit log entry contains all participant IDs

(2) Results Phase
- coordinator → participants: COMMIT or ABORT message
- participants
  - log & flush COMMIT or ABORT record to log
- participants → coordinator: ACK message
- coordinator
  - add END record to log
  - remove txn from TxnTbl
  - no flush required

2PC

* = flush log before sending message

Participant		Coordinator
• PREPARE or ABORT • coord_id	Prepare →	
	← Vote YES/NO	• commit* or ABORT • all participant IDs
• commit*/ abort*	commit/abort →	
	← ack(commit)	• end(commit)

2PC: Handling Failures

- if coordinator thinks a participant went down:
  - abort txn if participant has not voted
  - if waiting for ACK
    - → perform recovery
    - must finish committing what we started
- if participant thinks coordinator is down:
  - if no PREPARE record in log
    - → abort unilaterally (vote no)
  - if PREPARE record in log
    - all part. voted YES already
    - cannot proceed unilaterally

2PC: Optimization

- ACK messages
  - used by coordinator
    - tells when we can forget a txn / delete txn from TxnTbl
  - must keep txn until receive all ACKs
- if coordinator crashes after sending PREPARE messages:
  - unilaterally abort the txn
    - → coordinator can vote NO on it
  - tell anyone that asks that the txn is aborted
    - can abort in absence of information

2PC: Recovery

- if COMMIT/ABORT log record
  - do what usually do
  - if coordinator record (contains all participant IDs)
    - periodically send COMMIT/ABORT messages until we get all ACKs back
- if PREPARE log record:
  - this is a participant node
  - participant → coordinator (ID obtained from log record)
    - send message inquiring about txn status
- else
  - did not vote YES → abort the txn
  - if we receive VOTE YES/NO messages
    - this is the coordinator
    - → respond with ABORT message

optimization): presumed abort
- assume a txn aborts if we have no log records
  - → reduces messages sent/logging
- when a txn aborts:
  - coordinator cleans up locally
    - → remove txn from TxnTbl without ACKs
  - part. IDs do not need to be stored in ABORT records
    - bc not sending ACKs
  - part. that receive ABORT msgs do not send ACKs
  - if part. doesn't hear from coordinator about txn status:
    - → send inquiry
    - if txn not in coordinators TxnTbl
      - → return ABORT
  - ABORT records do not need to be flushed

2PC: Blocking

Q): node goes down during 1st phase (voting)?
- all part. that voted YES
  - keep locks
  - wait for COMMIT/ABORT message

Q): a participant never recovers?
- coordinator can
  - respawn part.
  - recover from log
  - → continue
- if participant comes up again → tell it to recycle itself

Q): what if coordinator does not recover?
- checkmate!

Keeping Track of Page Updates

- pageLSN → LSN of last record that modified the pg
  - stored in the page
  - updated in-memory when page is modified
  - flushed to disk w/ pg
    - → indicates how up-to-date the page on disk actually is
  - in-memory pageLSN can differ from disk pageLSN

Analysis Phase

Goal: Rebuild
- TransactionTbl at time of crash
- Dirty Pg Tbl

1) Start from checkpoint → LSN after BEGIN CHECKPOINT record
- if record not END
  - add Transaction to TransactionTbl (→ data structures) if not there
  - last LSN_transaction = record.LSN
  - if record = COMMIT or ABORT
    - change status in TransactionTable
  - if record = UPDATE
    - add record to Dirty Page Table
    - recLSN = record.LSN ∈ DPT
  - if record = END
    - remove transaction from Transaction Tbl
- (2) End of analysis phase
  - if transaction is COMMITTING
    - write END record
    - remove transaction from TxnTbl
  - if transaction is RUNNING
    - write ABORT record
    - change status to ABORTING

Memory / Log / Disk

Memory
- TxnTable
  - lastLSN
  - status
- DirtyPageTbl
  - recLSN
- flushedLSN
- BufferPool
  - ala pg hns an in-memory pageLSN
- LogTail

Log
- Log Records
  - LSN
  - prevLSN
  - txn number
  - type
  - page ID
  - length (bytes)
  - offset (bytes)
  - before-image
  - after-image

Disk
- Data Pgs
  - ala pg hns a diskpageLSN that may differ from in-memory pLSN
- Flushed Log Records

Commit protocol

(1) Write COMMIT record to log manager
(2) Flush log up to COMMIT record
(3) Write END record

ABORT Protocol

(1) Write ABORT record to log manager
(2) Undo all changes made by transaction.
- lastLSN (start) → prevLSN (backwards via prevLSN of each record)
- undo all log records that can be undone
  - → adding CLRs to log manager
(3) Write END record.

Log Records

	LSN	prevLSN	txn Num	Log Payload
log record	10	—	1	START
	20	10	1	UPDATE
	30	—	2	START
	40	20	1	COMMIT

LSN of last log record for same transaction

Data Structures: Dirty Page Table

- tracks dirty pages → modified in memory but not flushed to disk yet

Page num.	recLSN
46	11
63	24

recovery LSN indicating the first log record that caused the page to get dirty

Checkpoints

idea): do not want to start from beginning each crash!
- write BEGIN CHECKPOINT record
- latest record to start analysis at
  - → records later than this may have changes that may or may not be reflected in checkpoint
- write END CHECKPOINT record (sometime later)
  - contains: DPT, TransactionTable
  - after BEGIN checkpoint
  - do not know exactly when!

Redo Phase

goal): redo everything from earliest recLSN ∈ DPT
- get back unflushed changes before crash

(a) pg not in DPT
- pg on disk is up to date. no changes needed

(b) recLSN_pg > LSN
- no need to undo
- recLSN_pg is 1st record to dirty pg
  - → change is already flushed

(c) pgLSN ≥ LSN
- determines which changes have already been applied

Compensation Log Record (CLR)

idea): write a CLR for each undone operation
- write before actually undoing data
- undoNextLSN → next LSN to undo
- determined by prevLSN of current log record that is being undone

Data Structures: Transaction Table

- tracks active transactions

(running, aborting, committing) status of transaction

Transaction	lastLSN	status
1	A	33
2	C	42

last LSN/latest log record for transaction

Flushed LSN

- max LSN that was succesfully flushed to disk

$pageLSN_A \le flushedLSN$

- before A can be flushed to disk, need to ensure that corresponding log records are on disk
- ensures ATOMICITY

UNDO

TransactionTbl

- Any remaining transactions in the ATT didn't make it to completion, but may have some updates on disk (due to STEAL)
- We can't let their effects persist because of atomicity
- Undo all the "loser" transactions' UPDATES and write CLRs for each UPDATE

LSN	txn id	page	prevLSN	type
10	T1	A		Update
20	T1	B	10	Update
30	T2	C		Update
35				Begin checkpoint
40	T3	E		Update
50	T2		30	Commit
60	T3	D	40	Update
70	T2		50	End
80	T1	C	20	Update
90	T1		80	Commit
100	T3		60	Update
105				End Checkpoint
110	T1		90	End
115	T3		100	Abort

LSN	txn id	page	prevLSN	type	undoNextLSN
120	T3	E	115	CLR: Undo LSN 100	60
130	T3	D	120	CLR: Undo LSN 60	40
140	T3	E	130	CLR: Undo LSN 40	
150	T3		140	End	

ToUndo = { ~~100~~ ~~60~~ ~~40~~ }