Discussion 12

Logging, Recovery & Final Exam Review

EECS 484

Logistics

- **HW 6** due **Today** at 11:45 PM ET
- Final exam: **Dec 12th**, 7:00-9:00 PM ET
  - Practice Final released on Canvas
  - Please bring your Mcard (or a valid government-issued ID)
  - Scantron exam: Bring a #2 pencil + eraser; Multiple-choice questions
  - One 8.5x11 inch (double-sided) cheat sheet is allowed
    - Need to be handwritten
    - Include your name as you will hand it in with your exam
  - A scientific calculator **without** graphing or wi-fi capability is allowed and will be helpful

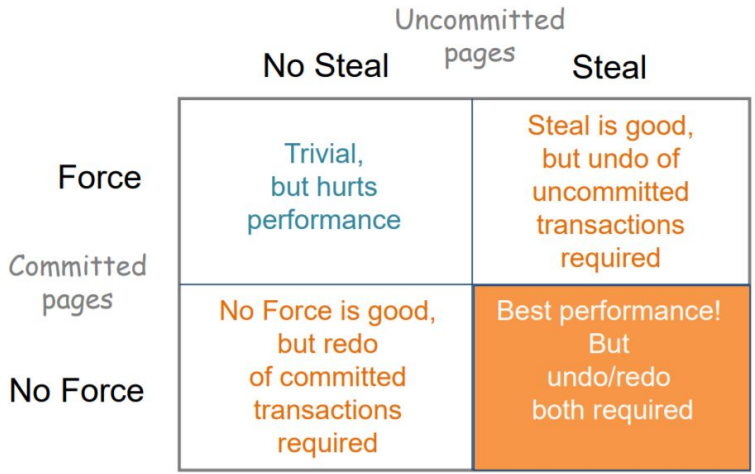
Logging and Recovery

Recovery

- We’ve talked about the ACID properties
  - Need to actually enforce
- Potential problems
  - Committed transaction results not maintained
    - Example: Write result in memory, result does not get stored on disk
    - Power goes off, results lost :(
  - Uncommitted transaction results maintained
    - Example: Temporary results in memory get written to disk (not enough memory space)
    - Power goes off, results maintained :(

Workarounds

- We can workaround this by enforcing constraints on our transaction manager
  - FORCE pages of a transaction to disk upon a commit
    - Don’t leave it in memory so once the commit happens, it’s durable on the disk
    - Added IOs to potentially write back to memory early
  - NO STEAL pages of uncommitted transactions
    - Keep in memory so they don’t get written early
    - Limited caching ability: can’t evict some pages
- Constraints yield a valid database
  - Performance degrades significantly though



Log Entries

- Entry for each update we make to the database
- Contains information we need
  - Log Sequence Number (LSN)
  - Transaction ID (XID)
  - Previous Log Sequence Number (prevLSN)
    - Backward pointer to last log entry for this transaction
  - Operation type:
    - Update, Commit, Abort, End (for Commit or Abort)
    - Compensation Log Record (Undo)
  - Page ID, Offset, Size, Old Data, New Data
    - Information about the data itself

Compensation Log Records (CLRs)

- Needed for undoing actions (ex: after an abort)
- Describes the undo action about to be performed
  - Remember we log before we do the action
  - Record of what the database looked like before the action we're undoing
    - The desired state after undoing the action
  - We don't need to remember the previous state
    - We never undo an undo
  - Points to the next action to undo
    - Quick way to jump to the next action to undo

Cascading chain of CLRs pictured

Write Ahead Logging (WAL)

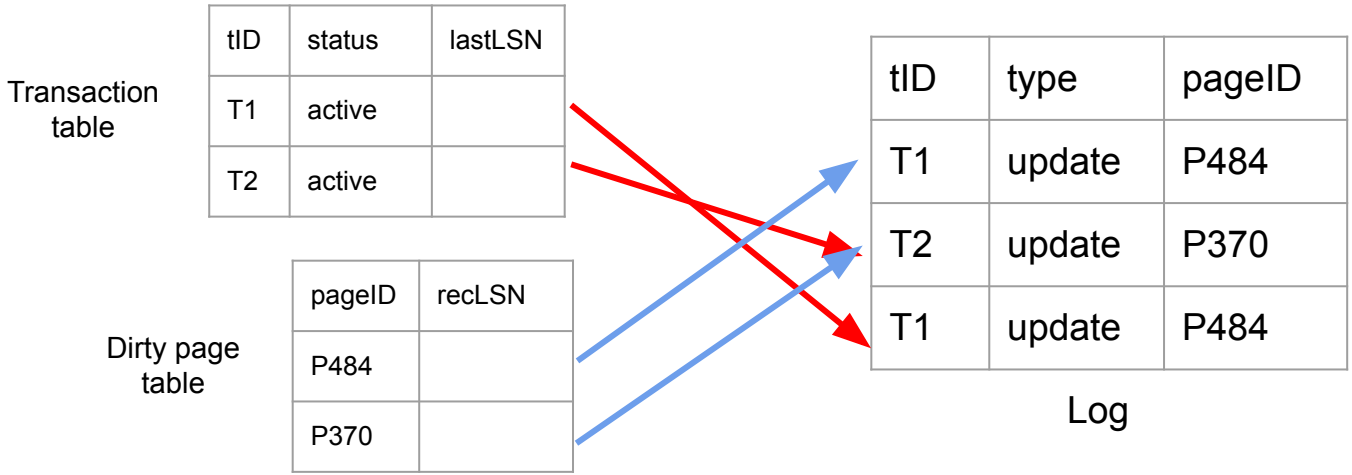
- Write Ahead Logging means we log before we write
- Force the log record for an update before the corresponding data page is written to the disk
  - Guarantees atomicity - we can undo stolen pages of transactions that were aborted
- Write all log records for a transaction before commits
  - Guarantees durability - we can redo committed pages that weren't forced

Checkpointing

- Every so often, the DBMS will save a copy of the transaction table and the dirty page table
  - DBMS will first create a “begin_checkpoint” record and add it to the log
  - DBMS will then create an “end_checkpoint” record that contains copies of the transaction and dirty page tables that are accurate *as of the time of the begin_checkpoint record* and add it to the log
  - Store the LSN of “begin_checkpoint” in the **master** record
- Allows the DBMS to start the analysis phase from the begin_checkpoint record rather than the very beginning of the log

Transaction table and dirty page table

- (Active) Transaction table
  - Contains an entry for every currently active transaction
  - Each entry will have a transaction ID, status, and the LSN of the latest log record for this transaction (lastLSN)
- Dirty page table
  - Contains an entry for every page that has changes that have not yet been written back to disk
  - Each entry will have a page ID and the LSN of the first log record that caused the page to be dirty (recLSN)
  - Updates from which might have to be redone



Checkpointing

- Every so often, the DBMS will save a copy of the transaction table and the dirty page table
  - DBMS will first create a “begin_checkpoint” record and add it to the log
  - DBMS will then create an “end_checkpoint” record that contains copies of the transaction and dirty page tables that are accurate *as of the time of the begin_checkpoint record* and add it to the log
  - Store the LSN of “begin_checkpoint” in the **master** record
- Allows the DBMS to start the analysis phase from the begin_checkpoint record rather than the very beginning of the log

Analysis Phase

- Recovery has three steps: Analysis, Redo, Undo
- Analysis reconstructs transaction and dirty-page tables at time of crash
  - Start at most recent checkpoint (begin_checkpoint)
  - Add all unfinished transactions to the transaction table
  - Add all modified pages to the dirty page table
    - There may be more pages in the dirty page table than were actually dirty at the time
    - We do not log page flushes which means some pages may have been written to disk before the crash
  - No edits are made to the log
- Will use dirty-page table for redo and transaction table for undo phase

Redo Phase

- Recovery has three steps: Analysis, Redo, Undo
- Redo makes sure that all updates made before the crash are redone
  - Does not care if transaction committed before the crash or not
  - Start with the smallest **recLSN** from the dirty page table
  - For each log record (update/CLR) with **LSN**, check
    - If the affected page, say P, is in the dirty page table
    - If **LSN** >= **recLSN** of P
    - If the **pageLSN** of P (on disk) < **LSN**
  - If all checks passed
    - Redo the update; set **pageLSN** of P = **LSN**
  - If any of the criteria are not met, then this action does not need to be reapplied
  - No edits are made to the log

Undo Phase

- Recovery has three steps: Analysis, Redo, Undo
- Start from end of log and work backwards
  - Construct a set **ToUndo** of LSNs that contains the maximum LSN of each uncommitted transaction
    - Can retrieve these from the transaction table
    - Take the log entry corresponding to the maximum LSN from **ToUndo**
      - If it refers to an update, undo the update and add a CLR that points to the prevLSN
        - We chain our CLRs this way!
        - Add the prevLSN to **ToUndo**
      - If it's a CLR, just add the corresponding **undoNextLSN** to **ToUndo**
        - If undoNextLSN is NULL, add an end record to the log.
        - We never undo undos!
    - Remove the currently considered LSN from the set and continue until set is empty
- Only phase that adds to the log (in the form of CLRs)

Recovery Example Question

LSN	LOG
1	update: T1 writes P102
2	update: T2 writes P102
3	update: T2 writes P103
4	begin_checkpoint
5	update: T1 writes P101
6	commit: T1
7	end_checkpoint
8	end: T1
9	abort: T2
10	CLR: Undo T2 LSN 3; undoNextLSN = ?
11	update: T3 writes P104
12	update: T3 writes P105
13	update: T4 writes P104
14	update: T5 writes P101
15	commit: T4
16	end: T4
CRASH, RESTART (Not a log record)	

Besides, a table showing the page information on the disk is found.

PageID	pageLSN
P101	5
P102	NULL
P103	NULL
P104	NULL
P105	12

LSN	LOG
1	update: T1 writes P102
2	update: T2 writes P102
3	update: T2 writes P103
4	begin_checkpoint
5	update: T1 writes P101
6	commit: T1
7	end_checkpoint
8	end: T1
9	abort: T2
10	CLR: Undo T2 LSN 3; undoNextLSN = ?
11	update: T3 writes P104
12	update: T3 writes P105
13	update: T4 writes P104
14	update: T5 writes P101
15	commit: T4
16	end: T4
CRASH, RESTART (Not a log record)	

Q: What is the undoNextLSN value of the log record with LSN = 10?

LSN	LOG
1	update: T1 writes P102
2	update: T2 writes P102
3	update: T2 writes P103
4	begin_checkpoint
5	update: T1 writes P101
6	commit: T1
7	end_checkpoint
8	end: T1
9	abort: T2
10	CLR: Undo T2 LSN 3; undoNextLSN = ?
11	update: T3 writes P104
12	update: T3 writes P105
13	update: T4 writes P104
14	update: T5 writes P101
15	commit: T4
16	end: T4
CRASH, RESTART (Not a log record)	

Q: What is the undoNextLSN value of the log record with LSN = 10?

A: **2**

LSN	LOG
1	update: T1 writes P102
2	update: T2 writes P102
3	update: T2 writes P103
4	begin_checkpoint
5	update: T1 writes P101
6	commit: T1
7	end_checkpoint
8	end: T1
9	abort: T2
10	CLR: Undo T2 LSN 3; undoNextLSN = ?
11	update: T3 writes P104
12	update: T3 writes P105
13	update: T4 writes P104
14	update: T5 writes P101
15	commit: T4
16	end: T4
CRASH, RESTART (Not a log record)	

Besides, a table showing the page information on the disk is found.

PageID	pageLSN
P101	5
P102	NULL
P103	NULL
P104	NULL
P105	12

Q: Give the Transaction Table and Dirty Page Table stored within the checkpoint. (***For DPT, assume no flushes before begin_checkpoint.)***

A:

LSN	LOG
1	update: T1 writes P102
2	update: T2 writes P102
3	update: T2 writes P103
4	begin_checkpoint
5	update: T1 writes P101
6	commit: T1
7	end_checkpoint
8	end: T1
9	abort: T2
10	CLR: Undo T2 LSN 3; undoNextLSN = ?
11	update: T3 writes P104
12	update: T3 writes P105
13	update: T4 writes P104
14	update: T5 writes P101
15	commit: T4
16	end: T4
CRASH, RESTART (Not a log record)	

Besides, a table showing the page information on the disk is found.

PageID	pageLSN
P101	5
P102	NULL
P103	NULL
P104	NULL
P105	12

Q: Give the Transaction Table and Dirty Page Table stored within the checkpoint. (***For DPT, assume no flushes before begin_checkpoint.)***

A:

Transaction Table	
TxID	lastLSN
T1	1
T2	3

Dirty Page Table	
PgID	recLSN
P102	1
P103	3

LSN	LOG
1	update: T1 writes P102
2	update: T2 writes P102
3	update: T2 writes P103
4	begin_checkpoint
5	update: T1 writes P101
6	commit: T1
7	end_checkpoint
8	end: T1
9	abort: T2
10	CLR: Undo T2 LSN 3; undoNextLSN = ?
11	update: T3 writes P104
12	update: T3 writes P105
13	update: T4 writes P104
14	update: T5 writes P101
15	commit: T4
16	end: T4
CRASH, RESTART (Not a log record)	

Q: Which LSN is stored in the master record?

A:

LSN	LOG
1	update: T1 writes P102
2	update: T2 writes P102
3	update: T2 writes P103
4	begin_checkpoint
5	update: T1 writes P101
6	commit: T1
7	end_checkpoint
8	end: T1
9	abort: T2
10	CLR: Undo T2 LSN 3; undoNextLSN = ?
11	update: T3 writes P104
12	update: T3 writes P105
13	update: T4 writes P104
14	update: T5 writes P101
15	commit: T4
16	end: T4
CRASH, RESTART (Not a log record)	

Q: Which LSN is stored in the master record?

A: **4**

LSN	LOG
1	update: T1 writes P102
2	update: T2 writes P102
3	update: T2 writes P103
4	begin_checkpoint
5	update: T1 writes P101
6	commit: T1
7	end_checkpoint
8	end: T1
9	abort: T2
10	CLR: Undo T2 LSN 3; undoNextLSN = ?
11	update: T3 writes P104
12	update: T3 writes P105
13	update: T4 writes P104
14	update: T5 writes P101
15	commit: T4
16	end: T4
CRASH, RESTART (Not a log record)	

Q: Which LSN should the analysis phase start from?

A:

LSN	LOG
1	update: T1 writes P102
2	update: T2 writes P102
3	update: T2 writes P103
4	begin_checkpoint
5	update: T1 writes P101
6	commit: T1
7	end_checkpoint
8	end: T1
9	abort: T2
10	CLR: Undo T2 LSN 3; undoNextLSN = ?
11	update: T3 writes P104
12	update: T3 writes P105
13	update: T4 writes P104
14	update: T5 writes P101
15	commit: T4
16	end: T4
CRASH, RESTART (Not a log record)	

Q: Which LSN should the analysis phase start from?

A: **4**

LSN	LOG
1	update: T1 writes P102
2	update: T2 writes P102
3	update: T2 writes P103
4	begin_checkpoint
5	update: T1 writes P101
6	commit: T1
7	end_checkpoint
8	end: T1
9	abort: T2
10	CLR: Undo T2 LSN 3; undoNextLSN = ?
11	update: T3 writes P104
12	update: T3 writes P105
13	update: T4 writes P104
14	update: T5 writes P101
15	commit: T4
16	end: T4
CRASH, RESTART (Not a log record)	

Transaction table and dirty page table stored within the checkpoint:

Transaction Table	
TxID	lastLSN
T1	1
T2	3

Dirty Page Table	
PgID	recLSN
P102	1
P103	3

Q: Give the Transaction Table and Dirty Page Table after the analysis phase is done.

A:

LSN	LOG
1	update: T1 writes P102
2	update: T2 writes P102
3	update: T2 writes P103
4	begin_checkpoint
5	update: T1 writes P101
6	commit: T1
7	end_checkpoint
8	end: T1
9	abort: T2
10	CLR: Undo T2 LSN 3; undoNextLSN = ?
11	update: T3 writes P104
12	update: T3 writes P105
13	update: T4 writes P104
14	update: T5 writes P101
15	commit: T4
16	end: T4
CRASH, RESTART (Not a log record)	

Transaction table and dirty page table stored within the checkpoint:

Transaction Table	
TxID	lastLSN
T1	1
T2	3

Dirty Page Table	
PgID	recLSN
P102	1
P103	3

Q: Give the Transaction Table and Dirty Page Table after the analysis phase is done.

A:

Transaction Table	
TxID	lastLSN
T2	10
T3	12
T5	14

Dirty Page Table	
PgID	recLSN
P101	5
P102	1
P103	3
P104	11
P105	12

LSN	LOG
1	update: T1 writes P102
2	update: T2 writes P102
3	update: T2 writes P103
4	begin_checkpoint
5	update: T1 writes P101
6	commit: T1
7	end_checkpoint
8	end: T1
9	abort: T2
10	CLR: Undo T2 LSN 3; undoNextLSN = ?
11	update: T3 writes P104
12	update: T3 writes P105
13	update: T4 writes P104
14	update: T5 writes P101
15	commit: T4
16	end: T4
CRASH, RESTART (Not a log record)	

Transaction table and dirty page table after analysis phase:

Transaction Table	
TxID	lastLSN
T2	10
T3	12
T5	14

Dirty Page Table	
PgID	recLSN
P101	5
P102	1
P103	3
P104	11
P105	12

Q: Which LSN should the redo phase start from?

A:

LSN	LOG
1	update: T1 writes P102
2	update: T2 writes P102
3	update: T2 writes P103
4	begin_checkpoint
5	update: T1 writes P101
6	commit: T1
7	end_checkpoint
8	end: T1
9	abort: T2
10	CLR: Undo T2 LSN 3; undoNextLSN = ?
11	update: T3 writes P104
12	update: T3 writes P105
13	update: T4 writes P104
14	update: T5 writes P101
15	commit: T4
16	end: T4
CRASH, RESTART (Not a log record)	

Transaction table and dirty page table after analysis phase:

Transaction Table	
TxID	lastLSN
T2	10
T3	12
T5	14

Dirty Page Table	
PgID	recLSN
P101	5
P102	1
P103	3
P104	11
P105	12

Q: Which LSN should the redo phase start from?

A: **1**

LSN	LOG
1	update: T1 writes P102
2	update: T2 writes P102
3	update: T2 writes P103
4	begin_checkpoint
5	update: T1 writes P101
6	commit: T1
7	end_checkpoint
8	end: T1
9	abort: T2
10	CLR: Undo T2 LSN 3; undoNextLSN = ?
11	update: T3 writes P104
12	update: T3 writes P105
13	update: T4 writes P104
14	update: T5 writes P101
15	commit: T4
16	end: T4
CRASH, RESTART (Not a log record)	

Transaction table and dirty page table after analysis phase:

Transaction Table	
TxID	lastLSN
T2	10
T3	12
T5	14

Dirty Page Table	
PgID	recLSN
P101	5
P102	1
P103	3
P104	11
P105	12

Q: Which operations must be redone? Give the corresponding LSNs.

A:

PageID	pageLSN
P101	5
P102	NULL
P103	NULL
P104	NULL
P105	12

LSN	LOG
1	update: T1 writes P102
2	update: T2 writes P102
3	update: T2 writes P103
4	begin_checkpoint
5	update: T1 writes P101
6	commit: T1
7	end_checkpoint
8	end: T1
9	abort: T2
10	CLR: Undo T2 LSN 3; undoNextLSN = ?
11	update: T3 writes P104
12	update: T3 writes P105
13	update: T4 writes P104
14	update: T5 writes P101
15	commit: T4
16	end: T4
CRASH, RESTART (Not a log record)	

Transaction table and dirty page table after analysis phase:

Transaction Table	
TxID	lastLSN
T2	10
T3	12
T5	14

Dirty Page Table	
PgID	recLSN
P101	5
P102	1
P103	3
P104	11
P105	12

Q: Which operations must be redone? Give the corresponding LSNs.

A: 1 2 3 10 11 13 14

PageID	pageLSN
P101	5
P102	NULL
P103	NULL
P104	NULL
P105	12

LSN	LOG
1	update: T1 writes P102
2	update: T2 writes P102
3	update: T2 writes P103
4	begin_checkpoint
5	update: T1 writes P101
6	commit: T1
7	end_checkpoint
8	end: T1
9	abort: T2
10	CLR: Undo T2 LSN 3; undoNextLSN = ?
11	update: T3 writes P104
12	update: T3 writes P105
13	update: T4 writes P104
14	update: T5 writes P101
15	commit: T4
16	end: T4
CRASH, RESTART (Not a log record)	

Transaction table and dirty page table after analysis phase:

Transaction Table	
TxID	lastLSN
T2	10
T3	12
T5	14

Dirty Page Table	
PgID	recLSN
P101	5
P102	1
P103	3
P104	11
P105	12

Q: Which LSN should the undo phase start from?

A:

LSN	LOG
1	update: T1 writes P102
2	update: T2 writes P102
3	update: T2 writes P103
4	begin_checkpoint
5	update: T1 writes P101
6	commit: T1
7	end_checkpoint
8	end: T1
9	abort: T2
10	CLR: Undo T2 LSN 3; undoNextLSN = ?
11	update: T3 writes P104
12	update: T3 writes P105
13	update: T4 writes P104
14	update: T5 writes P101
15	commit: T4
16	end: T4
CRASH, RESTART (Not a log record)	

Transaction table and dirty page table after analysis phase:

Transaction Table	
TxID	lastLSN
T2	10
T3	12
T5	14

Dirty Page Table	
PgID	recLSN
P101	5
P102	1
P103	3
P104	11
P105	12

Q: Which LSN should the undo phase start from?

A: 14

LSN	LOG
1	update: T1 writes P102
2	update: T2 writes P102
3	update: T2 writes P103
4	begin_checkpoint
5	update: T1 writes P101
6	commit: T1
7	end_checkpoint
8	end: T1
9	abort: T2
10	CLR: Undo T2 LSN 3; undoNextLSN = ?
11	update: T3 writes P104
12	update: T3 writes P105
13	update: T4 writes P104
14	update: T5 writes P101
15	commit: T4
16	end: T4
CRASH, RESTART (Not a log record)	

Transaction table and dirty page table after analysis phase:

Transaction Table	
TxID	lastLSN
T2	10
T3	12
T5	14

Dirty Page Table	
PgID	recLSN
P101	5
P102	1
P103	3
P104	11
P105	12

Q: Complete the LOG table after the recover process.

A:

LSN	LOG
1	update: T1 writes P102
2	update: T2 writes P102
3	update: T2 writes P103
4	begin_checkpoint
5	update: T1 writes P101
6	commit: T1
7	end_checkpoint
8	end: T1
9	abort: T2
10	CLR: Undo T2 LSN 3; undoNextLSN = ?
11	update: T3 writes P104
12	update: T3 writes P105
13	update: T4 writes P104
14	update: T5 writes P101
15	commit: T4
16	end: T4
CRASH, RESTART	
17	CLR: Undo T5 LSN 14; undoNextLSN = NULL
18	end: T5
19	CLR: Undo T3 LSN 12; undoNextLSN = 11
20	CLR: Undo T3 LSN 11; undoNextLSN = NULL
21	end: T3
22	CLR: Undo T2 LSN 2; undoNextLSN = NULL
23	end: T2

HW6 Q1.4

1.4: Schedule D [10 points]

T1	T2	T3
R(A)		
	W(B)	
R(C)		
		W(C)
R(B)		
	W(A)	
		R(A)
W(A)		

1. [4 points] What conflicts exist between T1 and T3?

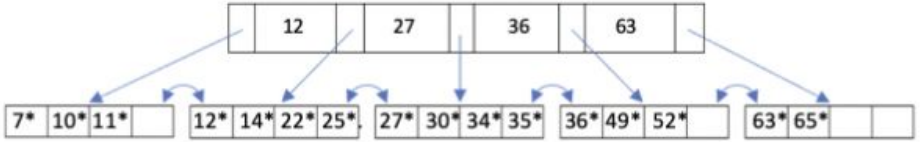
2. [4 points] Draw the precedence graph. Is the schedule conflict serializable?

3. [2 points] Is the schedule serializable? If so, provide a serial schedule that gives the same output.

Final Exam Review

B+ Tree

B+ Tree Practice Problem



I. What is the minimum number of insertions required to increase the height of the B+ tree? We prefer redistribution with the left sibling, then the right sibling over splitting. (2 points)

A. 2
B. 3
C. 4
D. 5
E. 6

II. Assume the following **deletions** are performed favoring merging over redistribution: 65*, 49*, 10*, 14*, 7*. What entries will be stored in the root node? (3 points)

A. 12, 27, 36, 63
B. 12, 22, 36, 52
C. 12, 27, 36
D. 12, 22, 36
E. 27, 36

B+ Tree Practice Problem

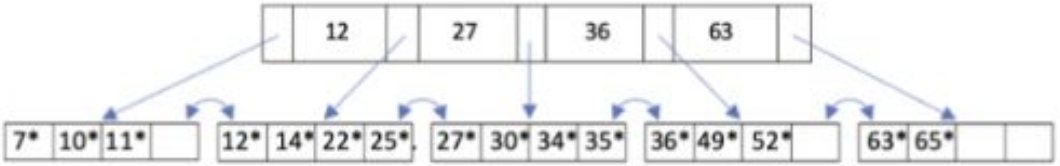
What is the minimum number of insertions required to increase the height of the B+ tree? We prefer redistribution with the left sibling, then the right sibling over splitting.
(2 points)

A. 2
B. 3
C. 4
D. 5
E. 6

Answer: A

Explanation:
First insertion can be on any of the first four leaf nodes.
Second insertion can be on the full leaf node where both left and right sibling are full, resulting in a split, as there isn't a sibling that the leaf node can share redistribute to.

B+ Tree Practice Problem



II. Assume the following **deletions** are performed favoring merging over redistribution: 65*, 49*, 10*, 14*, 7*. What entries will be stored in the root node? (3 points)

A. 12, 27, 36, 63
B. 12, 22, 36, 52
C. 12, 27, 36
D. 12, 22, 36
E. 27, 36

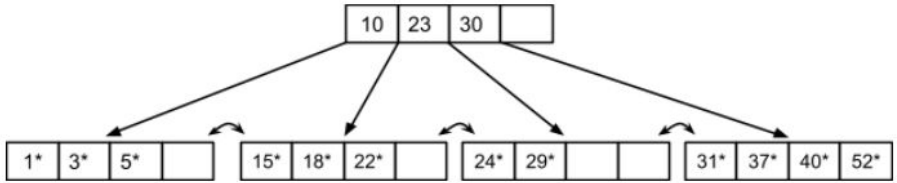
Answer: E

Explanation:
Deleting 65* will merge the fourth and fifth leaf nodes, making the root node only have (12, 27, 36)
Deleting 49*, 10*, 14* doesn't result in any merging or redistributions because each leaf node still has >=2 entries
Deleting 7* makes the first leaf node have < 2 entries, causing it to merge with the second leaf node, making the root node only have (27, 36).

B+ Tree Practice Problem

Assuming these B+ tree properties:

- The order of both non-leaf nodes and leaf nodes is 2.
- The left pointer points to values that are strictly less than the key value.
- For insertions, favor splitting over redistribution.
- During the splitting of a node, the left node will have one more value than the right node.
- For deletions, favor redistribution over merging.
- Only redistribute one entry at a time. During redistribution, when both are possible redistribution partners, always favor shifting elements from left to right.



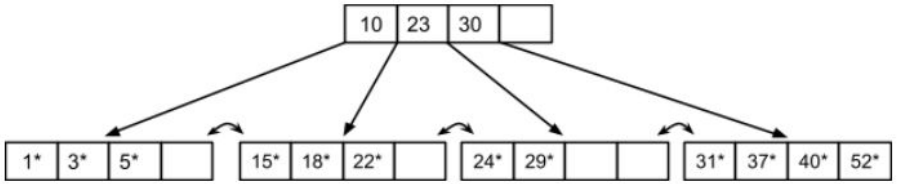
Answer the problems **3~5** given the B+ tree above. Problems **3~5** are not cumulative. They should be answered independently without assuming any previous modifications on the B+ tree given above. (Additional copies of the above are available on the last few sheets of the exam.)

3. Given the original tree above, which of the following would be the values inside the root node after performing the following operations: insert 16*, insert 21*? (3 points)
   A. 10, 22, 30
   B. 16, 23, 30
   C. 10, 21, 23, 30
   D. 10, 18, 23, 30
   E. None of the above.

B+ Tree Practice Problem

Assuming these B+ tree properties:

- The order of both non-leaf nodes and leaf nodes is 2.
- The left pointer points to values that are strictly less than the key value.
- For insertions, favor splitting over redistribution.
- During the splitting of a node, the left node will have one more value than the right node.
- For deletions, favor redistribution over merging.
- Only redistribute one entry at a time. During redistribution, when both are possible redistribution partners, always favor shifting elements from left to right.



Answer the problems **3~5** given the B+ tree above. Problems **3~5** are not cumulative. They should be answered independently without assuming any previous modifications on the B+ tree given above. (Additional copies of the above are available on the last few sheets of the exam.)

3. Given the original tree above, which of the following would be the values inside the root node after performing the following operations: insert 16*, insert 21*? (3 points)
   A. 10, 22, 30
   B. 16, 23, 30
   C. 10, 21, 23, 30
   D. 10, 18, 23, 30
   E. None of the above.

Answer: C

Explanation:

Insert 16* will fill up the second leaf node.
Insert 21* will cause the second node to split.
Since it states that the left node will have one more value than the right and that the left pointer points to values strictly less than the key value, the root node element's left pointer must be greater than 18 but less than or equal to 21.

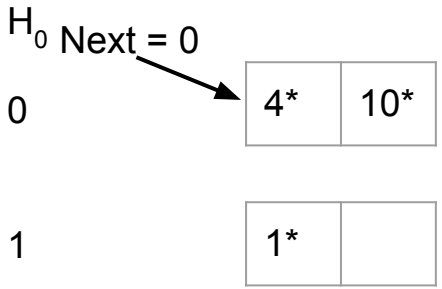
10, X, 23, 30
18 < X <= 21
Therefore, C is our only option.

Hash Tables

Linear Hashing

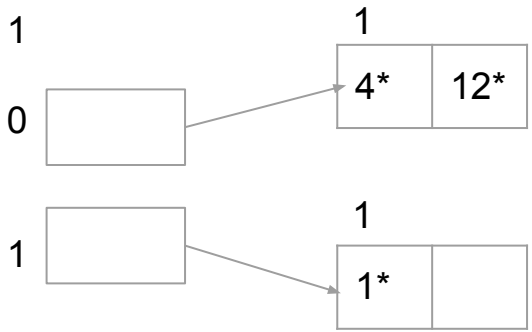
N = 2
$H_i(x) = x \pmod{N * 2^i}$
Level = 0

- Family of hash functions
- Split one bucket at a time upon an overflow
- N = **fixed** base number of buckets
- Level = current level in hash family
- Next = pointer to next bucket to be split
- Split policy: split on insertion into overflow page

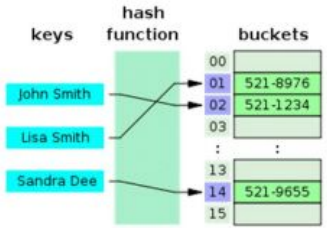


Extendible Hashing

- Use directory of pointers
  - Split on overflow
  - Once we're out of room in directory, double size
  - Global depth = number of bits considered globally
  - Local depth = number of bits considered locally



Hashing Overview



Hash browns < hash tables

- Linear Hashing
  - Only splits one bucket at a time
  - No directory doubling
  - Often has overflow pages, but over time these are minimized
  - Usually better memory usage, since directory typically takes up more space than overflow pages
- Extendible Hashing
  - Directory size can double
  - Can have recursive splitting
  - Global depth >= local depth always
  - # pointers to any specific bucket = 2^{GD-LD}
  - Overflow pages only in rare cases (duplicate keys)

Hash Tables Practice Problem

Question 7 (2 points)

Consider a linear hashing structure with 4 initial buckets (level = 0). How many buckets will exist at the beginning of round 4 (level = 4)?

A. 4
B. 8
C. 16
D. 32
E. 64

Hash Tables Practice Problem

Question 7 (2 points)

Consider a linear hashing structure with 4 initial buckets (level = 0). How many buckets will exist at the beginning of round 4 (level = 4)?

A. 4
B. 8
C. 16
D. 32
E. 64

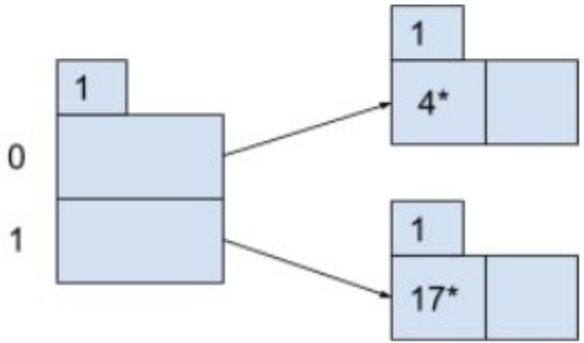
Answer: E
Explanation:
N = 4 at Level = 0
Bucket # at beginning of round = $N*2^{Level}$
$4*2^4 = 64$

Hash Tables Practice Problem

8. Given the original extendible hashing index (repeated below), what would be the local depth of the bucket containing 17* after performing the following operations: insert 6*, insert 8*? (3 points)

   A. 1
   B. 2
   C. 3
   D. 4
   E. None of the above

Decimal	Binary
6	110
8	1000



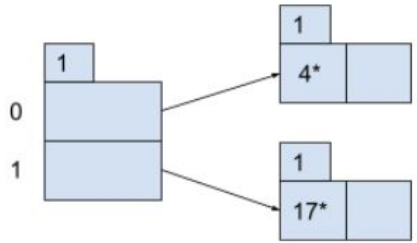
Hash Tables Practice Problem

Answer: A
Explanation:
6* would be inserted into the bucket containing 4*, making it full. 8* would be inserted into the bucket containing 4* and 6*, causing an overflow page. The directory will double, and the ‘0’ bucket would split to 00 and 01. Since the ‘1’ bucket did not split, the local depth would still be 1, with two directory pointers pointing to it.

8. Given the original extendible hashing index (repeated below), what would be the local depth of the bucket containing 17* after performing the following operations: insert 6*, insert 8*? (3 points)

A. 1
B. 2
C. 3
D. 4
E. None of the above

Decimal	Binary
6	110
8	1000



Sorting

General External Merge Sort

- Step 1:
  - Have a large dataset of N pages that you would like to sort using B buffer pages
- Step 2:
  - Divide the dataset into ceiling(N/B) runs (each of which is B pages long)
- Step 3:
  - Sort each run by itself normally using your favorite algorithm
  - We can fit the entire run of B pages into our RAM so no problem
- Step 4:
  - Sort the runs amongst each other
  - We can merge B-1 runs at a time
    - B-1 pages for each run plus 1 page to store the output
    - Each run is larger than 1 page though!
      - Load the first (sorted) page of each run and once it's empty, read the next page
      - Similarly, write the output buffer each time we run out of space and keep going

General External Merge Sort Math

- We have a dataset with N pages
  - We'll use B buffer pages
  - We'll have ceiling(N/B) runs initially
  - We need to make passes over the runs until entire dataset is sorted
    - We merge B-1 runs together at a time
    - That means we have ceiling(ceiling(N/B)/(B-1)) merged runs afterwards
    - Each time we make a pass we've merged all runs in sets of size B-1
    - We must continue to do this till we have 1 output dataset in sorted order
    - Takes $1+\text{ceiling}(\log_{B-1}\text{ceiling}(N/B))$ passes
  - Total IO cost is #passes * 2N
    - Each pass we read each page and write each page in a new sorted order

Sorting Practice Problem

18. We need to sort 1960 pages of unsorted data using the general external mergesort algorithm and use quicksort for the initial pass. There are 8 buffer pages in memory. Right after the initial pass of the algorithm, we suddenly have a different program using one page of our memory, so we cannot use that buffer page for the subsequent passes of the algorithm. How many sorted runs will be produced after a total of three passes (including the initial pass)? (4 points)

    A. 6
    B. 7
    C. 8
    D. 9
    E. 10

Sorting Practice Problem

18. We need to sort 1960 pages of unsorted data using the general external mergesort algorithm and use quicksort for the initial pass. There are 8 buffer pages in memory. Right after the initial pass of the algorithm, we suddenly have a different program using one page of our memory, so we cannot use that buffer page for the subsequent passes of the algorithm. How many sorted runs will be produced after a total of three passes (including the initial pass)? (4 points)
    A. 6
    B. 7
    C. 8
    D. 9
    E. 10

Answer: B
Explanation:
Passes = 1+ceiling($\log_{B-1}$ceiling(N/B))
After the initial pass, there are 1960 / 8 = 245 sorted runs
After the 2nd pass, there are 245 / (8 - 1 - 1) = 41 sorted runs
After the 3rd pass, there are 41 / (8 - 1 - 1) = 6.8 or 7 sorted runs
The (8 - 1 - 1) is the 8 buffer pages, minus 1 for the output buffer page, and another minus 1 for the memory page being used by the other program

Joins

Block Nested Loop Join

- Use B buffer pages
  - Read B-2 pages of R at a time
  - 1 page of S at a time
  - 1 page for output
- For each block of B-2 pages in R
  - For each Page of S
    - For each tuple in the B-2 pages of R
      - For each tuple in the page of S
        - Do the join
- 4 loops???
  - IO Cost = Load all M pages of R but load each N pages of S only (M/(B-2)) times
    - M+N*ceiling(M/(B-2))

```
for(int w=0; w<0; w++)
{
    for(int e=0; e<0; e++)
    {
        for(int l=0; l<0; l++)
        {
            for(int c=0; c<0; c++)
            {
                for(int o=0; o<0; o++)
                {
                    for(int m=0; m<0; m++)
                    {
                        for(int e=0; e<0; e++)
                        {
                            for(int t=0; t<0; t++)
                            {
                                for(int o=0; o<0; o++)
                                {
                                    for(int h=0; h<0; h++)
                                    {
                                        for(int e=0; e<0; e++)
                                        {
                                            for(int l=0; l<0; l++)
                                            {
                                                for(int l=0; l<0; l++)
                                                {
                                                    System.out.pr
```

Sort-Merge Join

- **Sort** both relations on the join attribute
  - May omit one final passe if the number of sorted runs from two relations is smaller than the number of buffer pages
- Look for qualifying by **merging** the two relations
- I/O Cost:
  - Sort R: 2M * #passes needed to sort R
  - Sort S: 2N * #passes needed to sort S
  - Typical merge cost: M + N
  - Worst merge cost: M * N
    - When?
  - Typical Sort-Merge Join cost:
    - 2M * 2 + 2N * 2 + M + N = 5(M + N)
    - Possible to finish in 3(M + N)! When?

Grace Hash Join

- What if we used advanced data structures(hash tables) to help us sort?
  - If h(r) = h(s) then very likely that r = s!
    - Assume we take the hash on the columns we are joining on
- 2 phases
  - Phase 1: Build
    - Put each tuple r and s into partitions
    - Any two elements in the same partition share a hash
    - Might have multiple partition rounds - need to enforce that each partition fits into buffer space
  - Phase 2: Probe
    - Load in each partition of the smaller relation (assume R for this case) and rehash
    - Read the corresponding partition for the larger relation (assume S for this case) and rehash
    - Compare the tuples and add to output if they match
- IO Cost: (2p+1)(M+N) where p is number of partition rounds
  - Partition = 2p(M+N) - Read + Write each page across both relations once
  - Probe = M+N - Read each page across both relations once more

Joins Practice Problem

Question 16 (4 points)

You are about to design a "join machine" to perform a grace hash join on two tables: students (600 pages) and school (15000 pages). What is the minimum number of buffer pages you need to minimize the cost of the join in terms of page writes. You may ignore the page read counts and final page writes. Assume the relation is evenly distributed in the partitioning phase.

A. 22
B. 24
C. 25
D. 26
E. 31

Joins Practice Problem

Question 16 (4 points)

You are about to design a "join machine" to perform a grace hash join on two tables: students (600 pages) and school (15000 pages). What is the minimum number of buffer pages you need to minimize the cost of the join in terms of page writes. You may ignore the page read counts and final page writes. Assume the relation is evenly distributed in the partitioning phase.

A. 22
B. 24
C. 25
D. 26
E. 31

Answer: D

Explanation:

The minimum number of buffer pages you need to minimize the cost of the join in terms of page writes can just be calculated using the smaller relation with 600 pages. We are looking for the B number of buffer pages to minimize the number of partition stages:

$600/(B-1)^{\#\text{ of partition rounds}} < B-2$

If we try out A, B, C, we get 2 partitioning rounds

If we try D and E, we get 1 partitioning round.

Since they both result in 1 round, we can just choose the lower number of buffer pages, in this case, 26 — D.

Query Optimization

Query Optimization Practice Problem

Question 11 (3 points)

Given the following schema and query, which of the following represents the correct and most efficient plan to execute the query assuming projections are pipelined?

StaffMember(Uniqname, FirstName, LastName, Role, Cid)
Course(Cid, Name)

SELECT S.FirstName FROM StaffMember S, Course C
WHERE S.cid = C.cid AND C.name = “EECS 484”;

A. $\pi_{FirstName}(\pi_{FirstName,\ uniqname}(S) \bowtie (\sigma_{name = "EECS\ 484"}(\pi_{name}(C)))$
B. $\pi_{FirstName}(\pi_{FirstName,\ cid}(S) \bowtie (\sigma_{name = "EECS\ 484"}(\pi_{name}(C)))$
C. $\pi_{FirstName}(S \bowtie (\sigma_{name = "EECS\ 484"}(C))$
D. $\pi_{FirstName}(\pi_{FirstName,\ cid}(S) \bowtie (\sigma_{name = "EECS\ 484"}(C))$
E. $\pi_{FirstName}(\pi_{FirstName,\ uniqname}(S) \bowtie (\sigma_{name = "EECS\ 484"}(C))$

Query Optimization Practice Problem

Answer: D

Explanation:

We know we can apply selection and projection prior to the join. A and B are eliminated as projecting on ‘name’ wouldn’t allow the join on ‘cid’ to occur. We know we can apply a selection on C. We can apply a projection on S since it only needs ‘cid’ and ‘firstName’ for and after the join. This only leaves D as the option for the answer.

Question 11 (3 points)

Given the following schema and query, which of the following represents the correct and most efficient plan to execute the query assuming projections are pipelined?

StaffMember(Uniqname, FirstName, LastName, Role, Cid)
Course(Cid, Name)

SELECT S.FirstName FROM StaffMember S, Course C
WHERE S.cid = C.cid AND C.name = “EECS 484”;

A. $\pi_{FirstName}(\pi_{FirstName,\ uniqname}(S) \bowtie (\sigma_{name = "EECS\ 484"}(\pi_{name}(C))))$
B. $\pi_{FirstName}(\pi_{FirstName,\ cid}(S) \bowtie (\sigma_{name = "EECS\ 484"}(\pi_{name}(C))))$
C. $\pi_{FirstName}(S \bowtie (\sigma_{name = "EECS\ 484"}(C))$
D. $\pi_{FirstName}(\pi_{FirstName,\ cid}(S) \bowtie (\sigma_{name = "EECS\ 484"}(C))$
E. $\pi_{FirstName}(\pi_{FirstName,\ uniqname}(S) \bowtie (\sigma_{name = "EECS\ 484"}(C))$

Index Matching

Index Matching Practice Problem

14. Given the USERS relation with the attributes (uid, age, MOB) and a hash index on this relation with the search key <age, MOB>, which of the following queries matches the index and we can perform an index-only scan? (3 points)

A. SELECT age, MOB FROM users WHERE MOB = 10;
B. SELECT age FROM USERS WHERE age = 10;
C. SELECT uid, age FROM USERS WHERE age = 10 AND MOB = 10;
D. SELECT MOB FROM users WHERE age = 10 AND MOB = 10;
E. None of the above.

Index Matching Practice Problem

14. Given the USERS relation with the attributes (uid, age, MOB) and a hash index on this relation with the search key <age, MOB>, which of the following queries matches the index and we can perform an index-only scan? (3 points)

A. SELECT age, MOB FROM users WHERE MOB = 10;
B. SELECT age FROM USERS WHERE age = 10;
C. SELECT uid, age FROM USERS WHERE age = 10 AND MOB = 10;
D. SELECT MOB FROM users WHERE age = 10 AND MOB = 10;
E. None of the above.

Answer: D

Explanation:

A and B are incorrect as the hash index requires both age and MOB in the WHERE clause.

C and D both match the hash index; however, D is the correct answer as the question asks for the query that matches the index **and** we can perform an **index-only** scan. C would require you to scan the disk for the uid, while D would only require scanning of the index.

(Note: we do agree that the query itself is odd as it would only return MOB 10 or no results, but it is a valid question)

Transactions

Transaction Practice Problem

Question 21 (3 points)
Determine if the following schedules are serializable, conflict-serializable or serial schedules.
Select all that apply.

T1	T2
	R(A)
R(B)	
	W(A)
W(B)	
Commit	
	R(B)
	W(B)
	Commit

A. Serializable
B. Conflict Serializable
C. Serial
D. None of the above

Transaction Practice Problem

Question 21 (3 points)

Determine if the following schedules are serializable, conflict-serializable or serial schedules. Select all that apply.

T1	T2
	R(A)
R(B)	
	W(A)
W(B)	
Commit	
	R(B)
	W(B)
	Commit

A. Serializable
B. Conflict Serializable
C. Serial
D. None of the above

Answer: AB

Explanation:

It can't be C since there are two interleaving translation.

We can first test for Conflict Serializability by creating the dependency graph.

T1->T2 (WW conflict)

Since there is no cycle, this schedule is conflict serializable, and as a result, serializable.

Transaction Practice Problem

Strong Strict 2PL

25. Below is the timeline of arrival of actions in three different transactions. Suppose we are following **strict 2PL.** Assume that a transaction will immediately commit after its last action listed is performed. What is the order that three transactions will finish (4 points)?

Time	Transaction and Action
1	T1 R(A)
2	T2 R(A)
3	T3 R(B)
4	T3 R(A)
5	T1 W(A)
6	T2 W(B)

A. Transactions complete in the order T1, T2, T3
B. Transactions complete in the order T3, T1, T2
C. Transactions compete in the order T3, T2, T1
D. Transactions complete in the order T2, T3, T1
E. There is a deadlock or none of the above

Transaction Practice Problem

Strong Strict 2PL

25. Below is the timeline of arrival of actions in three different transactions. Suppose we are following **strict 2PL**. Assume that a transaction will immediately commit after its last action listed is performed. What is the order that three transactions will finish (4 points)?

Time	Transaction and Action
1	T1 R(A)
2	T2 R(A)
3	T3 R(B)
4	T3 R(A)
5	T1 W(A)
6	T2 W(B)

A. Transactions complete in the order T1, T2, T3
B. Transactions complete in the order T3, T1, T2
C. Transactions compete in the order T3, T2, T1
D. Transactions complete in the order T2, T3, T1
E. There is a deadlock or none of the above

Answer: C
Explanation:
With Strong Strict 2PL, the transaction will hold all of its locks until the end of the transaction, or in this case, when the last action from this transaction occurs.
T1 acquires a reader lock on A.
T2 acquires a reader lock on A.
T3 acquires a reader lock on B.
T3 acquires a reader lock on A.
< T3 finishes, releases all locks >
T1 wants to acquire a writer lock on A, but T2 has a reader lock on A, causing T1 to wait for T2 to finish.
T2 acquires a writer lock on B.
< T2 finishes, releases all locks >
T1 acquires a writer lock on A.
< T1 finishes, releases all locks >

Good luck on the final!