CS143 Quiz 2 In-person

CHARLES XIAN ZHANG

TOTAL POINTS

86.5 / 100

QUESTION 1

Problem 1: Database Integrity 20 pts

1.1 1.1 **8 / 8**

✓ **- 0 pts** Correct

1.2 1.2.1 **3 / 3**

✓ **- 0 pts** Correct

1.3 1.2.2 **3 / 3**

✓ **- 0 pts** Correct

1.4 1.2.3 **3 / 3**

✓ **- 0 pts** Correct

1.5 1.2.4 **3 / 3**

✓ **- 0 pts** Correct

QUESTION 2

Problem 2: Disks and Files 20 pts

2.1 2.1 **5 / 5**

✓ **- 0 pts** Correct (271, $$\lceil{\frac{10,000}{\lfloor \frac{4096}{108}\rfloor}}\rceil$$)

2.2 2.2.a **5 / 5**

✓ **- 0 pts** Correct (0.01 ms, 10 microseconds, $$1 \times 10^{-5}S$$)

2.3 2.2.b **3 / 5**

✓ **- 2 pts** wrong number of blocks (off by more than 1)

2.4 2.3.a **1 / 1**

✓ **- 0 pts** Correct (false)

2.5 2.3.b **1 / 1**

✓ **- 0 pts** True (Correct)

2.6 2.3.c **1 / 1**

✓ **- 0 pts** Correct (False)

2.7 2.3.d **1 / 1**

✓ **- 0 pts** Correct (True)

2.8 2.3.e **1 / 1**

✓ **- 0 pts** True (Correct)

QUESTION 3

Problem 3: Index 20 pts

3.1 3.1.a **4 / 4**

✓ **- 0 pts** Correct: Studio and (title OR prequel)

3.2 3.1.b **4 / 4**

✓ **- 0 pts** Correct (studio)

3.3 3.2 **4 / 4**

✓ **- 0 pts** Correct (ecdf)

3.4 3.3 **4 / 8**

✓ **- 4 pts** mistakes in merging(updating) underflow non-leaf node / deleting key from non-leaf node

QUESTION 4

Problem 4: Cost of Join 20 pts

4.1 Memory usage (Q 4.1) and Average traverse cost (Q 4.2) **15 / 15**

✓ **- 0 pts** Correct

4.2 Reducing traverse cost(Q 4.3) **5 / 5**

✓ **- 0 pts** Correct

QUESTION 5

5 Problem 5: NoSQL 2.5 / 10

d (CASE stmt for B)

✓ **- 1 pts** **Missing or incorrect syntax (beyond minor errors), potentially including 2 or more of deductions here.**

e ("GROUP BY A, `d-answer`")

(note: UNION attempts did not require GROUP BY A+`d-answer`, but do require A)

✓ **- 1 pts** **Missing or incorrect**

f ("SUM(C)")

✓ **- 1 pts** **Missing or incorrect**

g ("HAVING `f-answer` > 20")

✓ **- 0.5 pts** **Incorrect use (e.g., WHERE or wrong attribute instead of `f-answer`)**

h.1: separate query to reflect rBK grouping with fixed key (global group)

✓ **- 1 pts** **Missing or incorrect (all)**

h.2: Projection of attribute `A` from inner query (or if no inner query, then for use in final result projection `i`)

✓ **- 1 pts** **Missing or incorrect**

i ("MIN(A)") (used MIN(`h.2-answer`) if applicable)

✓ **- 1 pts** **Missing or incorrect**

j ("SELECT `i-answer` FROM `h1-answer`)

✓ **- 1 pts** **Missing or incorrect**
(common case: when `h.1` is also missing)

QUESTION 6

Problem 6: Transactions 10 pts

6.1 5 / 5

✓ **- 0 pts** **Correct**

6.2 5 / 5

✓ **- 0 pts** **Correct**

UCLA
Computer Science Department
Fall 2021

Instructor: J. Cho

CS143 Quiz 2: 2 hours

Student Name: Charles Zhang

Student ID: 305-413-659

(IMPORTANT PLEASE READ **):**

- The exam is *closed book* and *closed notes.* You may use *two double-sided cheat-sheets,* 4 pages in total. You can use a calculator.
- *Simplicity and clarity of your solutions will count.* You may get as few as 0 point for a problem if your solution is far more complicated than necessary, or if we cannot understand your solution.
- If you need to make any assumption to solve a question, *please write down your assumptions.* To get partial credit, you may want to write down how you arrived at your answer step by step.
- If a question asks for a numeric answer, you don't have to calculate. You may just write down a numeric expression.
- Please, write your answers neatly.

Problem	Score	
1	20	
2	20	
3	20	
4	20	
5	10	
6	10	
Total	100	

Problem 1: Database Integrity (20 points)

For all questions in this problem, we refer to three tables created by the following statements:

```
CREATE TABLE Studio(
    id      INT,
    name    VARCHAR(50),
    PRIMARY KEY(id),
    UNIQUE(name)
);

CREATE TABLE Movie(
    id      INT,
    title   VARCHAR(50),
    year    INT,
    sid     INT,
    PRIMARY KEY(id),
    FOREIGN KEY(sid) REFERENCES Studio(id)
        ON DELETE CASCADE ON UPDATE CASCADE
);

CREATE TABLE ShowTime(
    theater     VARCHAR(50),
    mid         INT,
    time        DATETIME,
    ticketPrice DECIMAL(5,2),
    PRIMARY KEY(theater, mid),
    FOREIGN KEY(mid) REFERENCES Movie(id)
        ON UPDATE CASCADE
);
```

As you can imagine from their names, `Studio` table contains the movie production studio information (like "Disney"), `Movie` table contains the movie information (like "Avengers: Endgame"), and the `ShowTime` table contains the movie showtime information at theaters.

1. Assume that the production studio of a movie never changes once this information is inserted into the three tables. Given this, is it possible to use a SQL92 CHECK constraint to enforce that the ticket price of any Disney movie should never be below $5.00? If your answer is yes, write down the CHECK constraint and the name of the table to which the constraint should be added. If your answer is no, briefly explain why it is not possible. (8 points)

Yes

```
CREATE TABLE ShowTime (
    ...
    CHECK(ticketPrice ≥ 5.00 OR (SELECT S.name FROM Studio S
        WHERE S.id = (SELECT sid FROM Movie
            WHERE Movie.id = mid))
        <> 'Disney'
    ...
)
```

2. For this problem, consider the three tables created with the earlier `CREATE TABLE` statement, but ignore any assumption that we made in P1.1 The three tables currently have the following tuples:

Studio

id	name
1	Disney
2	~~Universal~~
3	20th Century

Movie

id	title	year	sid
1	Black Panther	2018	1
2	~~Minions~~	~~2015~~	~~2~~
3	Frozen	2013	1
4	Avatar	2009	3

ShowTime

theater	mid	time	ticketPrice
AMC	1	2021-11-25 10:00:00	10.00
Regency	3	2021-11-24 21:00:00	9.00
Landmark	3	2021-11-23 20:00:00	11.00
AMC	4	2021-11-27 09:00:00	7.00

Now, for the statement(s) given on the following page, briefly state what will happen when the statements are issued in the given sequence. If some tuples are inserted, updated, or deleted due to the statements, clearly indicate the exact list of tuples that are affected (and their new values if they are updated). If any statement is rejected, briefly explain why. Consider each subproblem separately from others. Each of the following subproblems amounts to 3 points.

1. `INSERT INTO Movie Values (5, 'Spiderman', 2012, 4);`
   `INSERT INTO Studio Values (4, 'Sony');`

   The first tuple will be rejected due to referential integrity, since it tries to insert a tuple with sid=4, and all sids in Movie must be an id in Studio (4 is not yet).

   The second tuple succeeds in insertion.

2. `DELETE FROM Studio WHERE id = 2;`

   This deletion will succeed, deleting the tuple (2, 'Universal') from Studio.

   It will also delete the tuple (2, 'Minions', 2015, 2) from Movie, since Movie's sid references Studio's id, and is set to cascade on deletion.

3. `DELETE FROM Studio WHERE name = '20th Century';`

   This deletion will fail, since Movie's sid references Studio's id and ShowTime's mid references Movie's id. Deleting the tuple with name='20th Century' would delete Studio id 3, This will cascade, resulting in the deletion of (4, 'Avatar', 2009, 3) from Movie. However, the tuple ('AMC', 4, '2021-11-27 09:00:00, 7.00) in ShowTime needs a tuple in Movie with id=4 to exist. Since no behavior is specified on delete, this will reject, resulting in no tuples being deleted from any table.

4. `UPDATE Movie SET id = 5, sid = 1 WHERE id = 4;`

   This statement sets the tuple (4, 'Avatar', 2009, 3) in Movie to (5, 'Avatar', 2009, 1), since mid in ShowTime references Movie's id, and on update behavior is set to cascade, any tuple in ShowTime where mid=4 must be updated to mid=5. This results in ('AMC', 4, '2021-11-27 09:00:00', 7.00) being update to ('AMC', 5, '2021-11-27 09:00:00, 7.00).

Problem 2: Disks and Files (20 points)

Consider a movie table created by the following SQL statement:

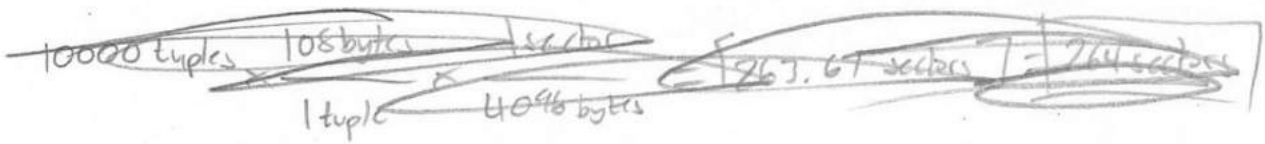
```
CREATE TABLE Movie(
    id        INT,
    title     CHAR(50),
    studio    CHAR(50),
    year      INT
)
```

An integer value takes up 4 bytes to store and `CHAR(n)` type takes n bytes. The Movie table contains 10,000 tuples and is stored as fixed-length records. The tuples are stored in a row-oriented format and are *not spanned*. We use a disk of the following parameter to store the table.

- 4 platters (8 surfaces)
- 10,000 tracks
- 1,000 sectors per track
- 4,096 bytes per sector
- 6,000 RPM rotational speed
- 10ms average seek time, 2ms minimum seek time between adjacent tracks, 30ms maximum full-stroke seek time

Assume that all blocks for the table are all allocated sequentially. That is, if the table fits in one track, all blocks for the table are allocated as consecutive blocks on the same track. If the table size is bigger than one track, the blocks are allocated in multiple adjacent tracks. Assume that one disk sector corresponds to one disk block.

1. What is the minimum number of blocks that we need to store the movie table? (5 points)



$$\left\lfloor \frac{1 \text{ tuple}}{108 \text{ B}} \times \frac{4096 \text{ B}}{1 \text{ sector}} \right\rfloor = 37 \text{ tuples/sector}$$

$$\left\lceil \frac{1 \text{ sector}}{37 \text{ tuples}} \times 10000 \text{ tuples} \right\rceil = \boxed{271 \text{ sectors}}$$

2. For this subproblem, consider the following SQL query:

```
SELECT * FROM Movie WHERE year = 2020 FETCH FIRST 1 ROWS ONLY;
```

Assume that the table has not been sorted by any attribute and no index has been constructed on the table. Therefore, the system executes the above query by scanning the table from the beginning until a matching tuple is identified. In year 2020, assume that only one movie was released due to pandemic.

a. In the *best possible scenario*, how long does it take to retrieve the tuple that satisfies the above SQL statement from the disk? Briefly explain how you arrived at your answer. (5 points)

Access Time = Seek Time + Rot. Delay + Transfer Time

$= 0\text{ms} + 0\text{ms} + \left(\frac{1\text{min}}{6000\text{revs}} \times \frac{60000\text{ms}}{1\text{min}} \times \frac{1\text{rev}}{1000\text{sectors}}\right)$

$= \boxed{0.01\text{ms}}$

In the best possible scenario, the head is already in the correct track, at the beginning of where the correct sector is. As a result, there is no seek time or rotational delay. From there, we just need to calculate the transfer time for the sector the tuple is in, which is the amount of time to complete 1/1000 of a rotation.

b. In the *average case scenario*, how long does it take to retrieve the tuple that satisfies the above SQL statement from the disk? Briefly explain how you arrived at your answer. (5 points)

Access Time = Seek Time + Rot. Delay + Transfer Time

$= 10\text{ms} + \frac{1}{2}\left(0\text{ms} + \frac{1\text{min}}{6000\text{revs}} + \frac{60000\text{ms}}{1\text{min}}\right) + 0.01\text{ms}$

$= 10\text{ms} + 5\text{ms} + 0.01\text{ms}$

$= \boxed{15.01\text{ms}}$

In the average case, we take the average seek time and rotational delay, and add it to the constant transfer time from part a). The average seek time is given, and we calculate the average rotational delay as the average of no rotation time and the time it takes for a full rotation.

3. For this subproblem, assume that the table is stored as a sorted file in increasing order of (studio, year) attributes. That is, the tuples are first sorted by the studio attribute and then by the year attribute if two tuples have the same studio value. For each of the following statement, circle either True or False and explain your answer briefly. Assume a single-level index unless indicated otherwise. (5 points)

a. It is possible to build a sparse index on title. TRUE / FALSE

A sparse index on an attribute requires the table to be sorted by that attribute. In this problem, Movie is sorted by (studio, year), not title.

b. It is possible to build a dense index on studio. TRUE / FALSE

A dense index has a pointer for every tuple, and therefore has no restrictions relevant to this problem.

c. It is possible to build a sparse index on year. TRUE / FALSE

To build a sparse index on year, we would need to sort primarily by year. However, since we first sort by studio, with year as a tiebreaker, this will not work.

d. It is possible to build a dense index on year. TRUE / FALSE

A dense index doesn't require the attribute to be sorted, therefore a dense index on year is possible.

e. It is possible to build a sparse index on studio. TRUE / FALSE

Movie is sorted primarily by studio, so this will work.

Problem 3: Index (20 points)

1. Consider the Movie table with the following schema:

   Movie(title, year, studio, prequel, revenue)

   The revenue column has the box-office revenue for the movie. The prequel column has the title of the movie's prequel if any and NULL otherwise. Assume that a movie's title is unique even though our table definition does not include either the primary key or the unique constraint on title.

   Consider the following two queries that we want to execute repeatedly on the table

```
Q1: SELECT SUM(revenue) FROM Movie WHERE studio={studio} GROUP BY year;
Q2: SELECT M1.title
    FROM Movie M1, Movie M2
    WHERE M1.prequel = M2.title AND M1.revenue > M2.revenue;
```

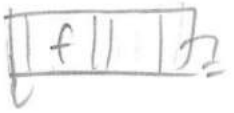
   Note that `{studio}` in Q1 is provided by the user when the query is executed so this value varies in each execution.

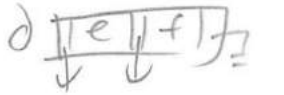
   a. Assume that we want to build *two* indexes to support the previous two queries efficiently. Specify two attributes, one attribute per each index on which the index should be built. (4 points)

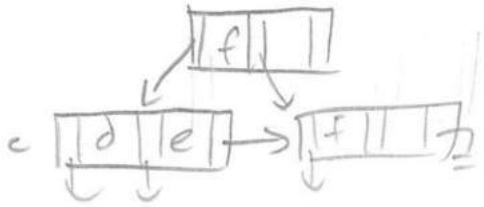
   One index on studio, one index on title.

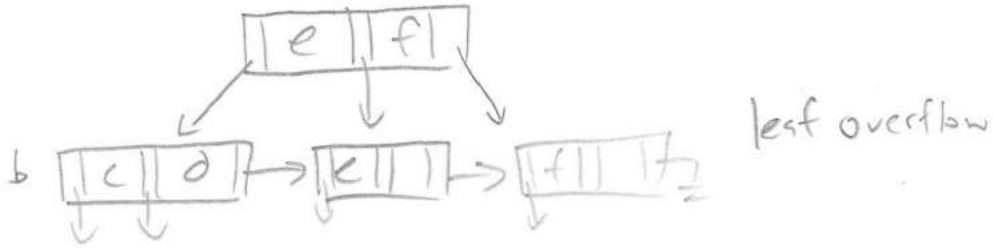
   b. State which of the two indexes should be clustered. Assume that both queries are executed with roughly equal frequencies and we do not run any other query on the table. Explain your answer briefly. (4 points)

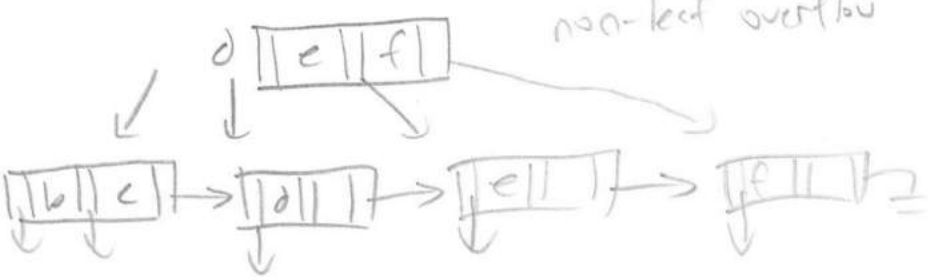
   The index on studio should be clustered, as Q1 is a range query which will benefit from sorting, while Q2 is a point query since titles are unique. → Q1 wants all of a given studio

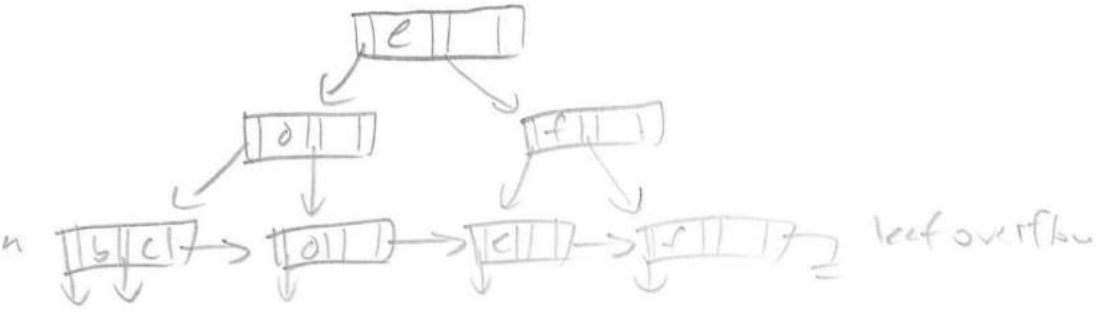


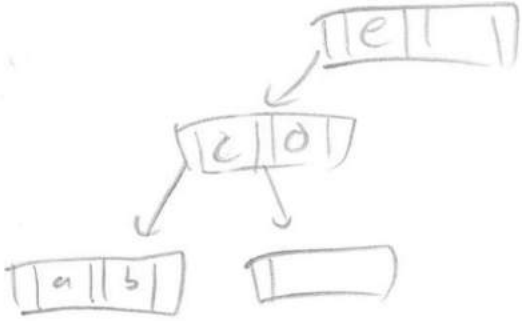

d
leaf overflow


c
leaf overflow


b
leaf overflow


a
non-leaf overflow

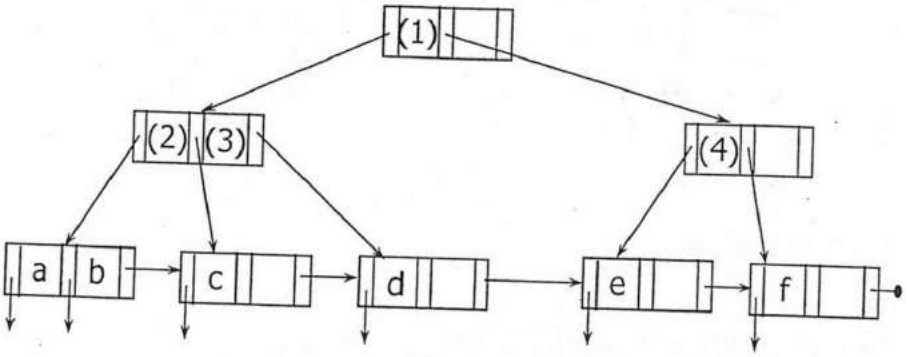

n
leaf overflow



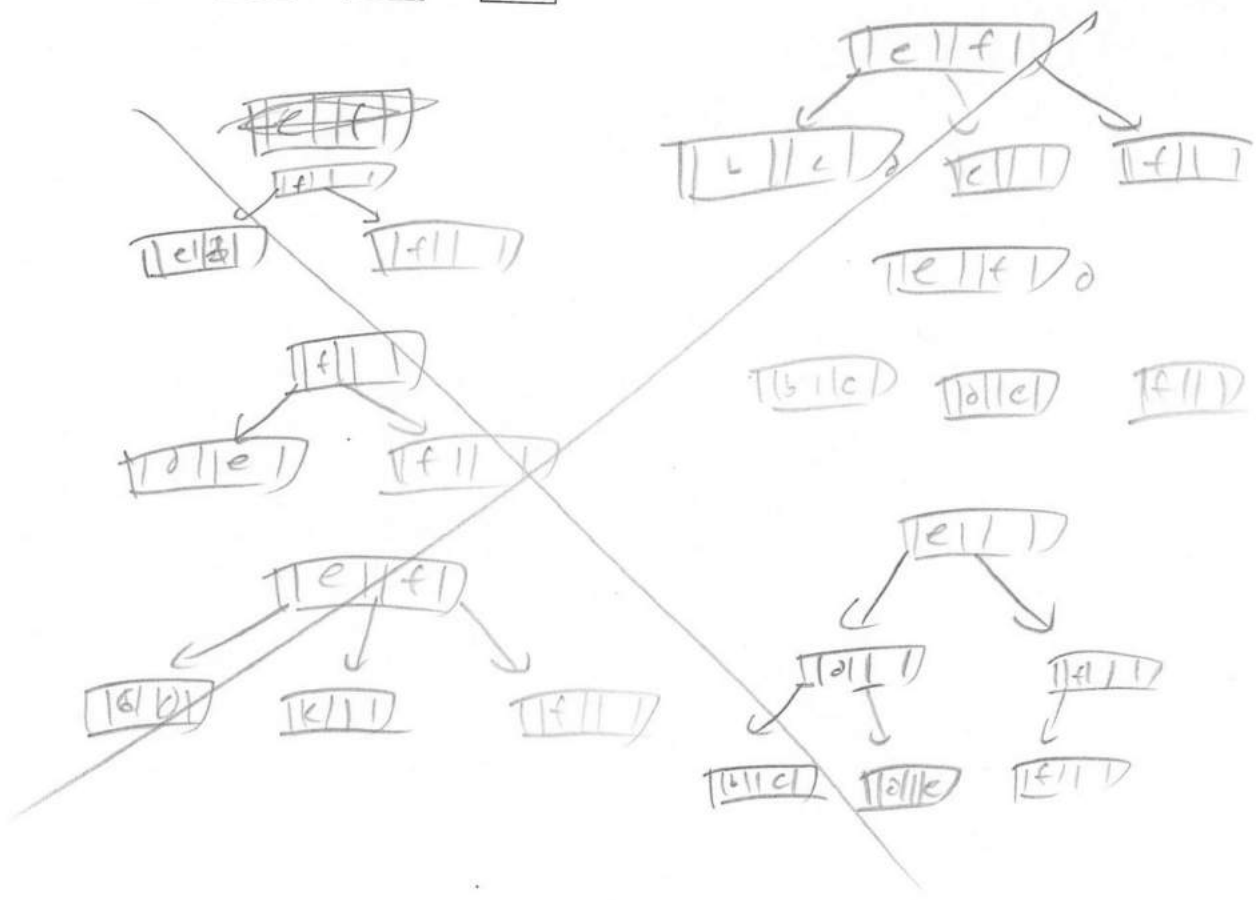
2. Consider the B+Tree insertion algorithm that we learned in the class. Assume that if there is an overflow after an insertion and a node is split, the left node gets 1 more keys/pointers than the right node, if the overflowing keys/pointers cannot be evenly distributed between the two nodes.

   Consider the following B+Tree constructed using the algorithm by inserting the keys a, b, ..., f in the reverse order. In the space provided below, write down the key values that should be at the positions labeled as (1), (2), (3), and (4) in the tree. (4 points)

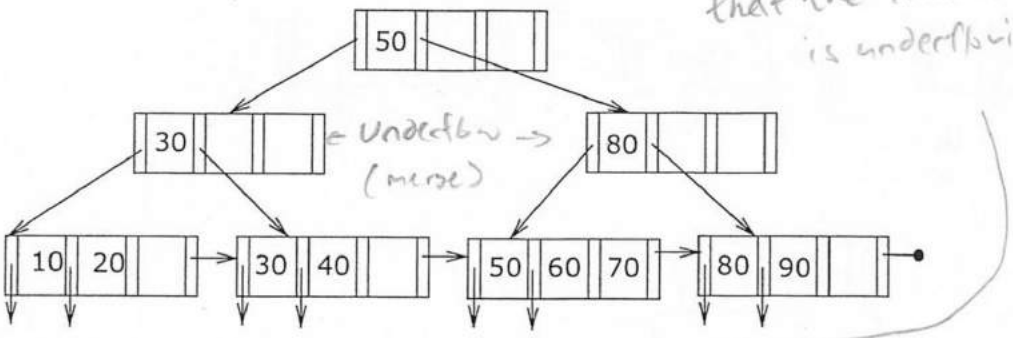
Work



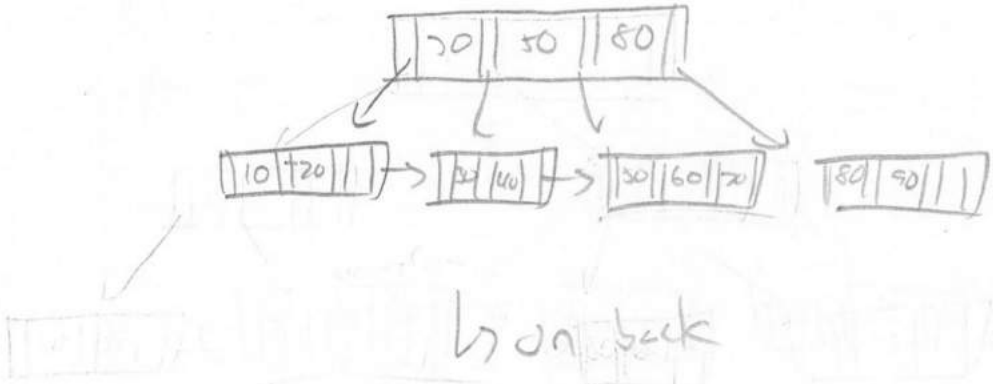
(1) e (2) c (3) d (4) f

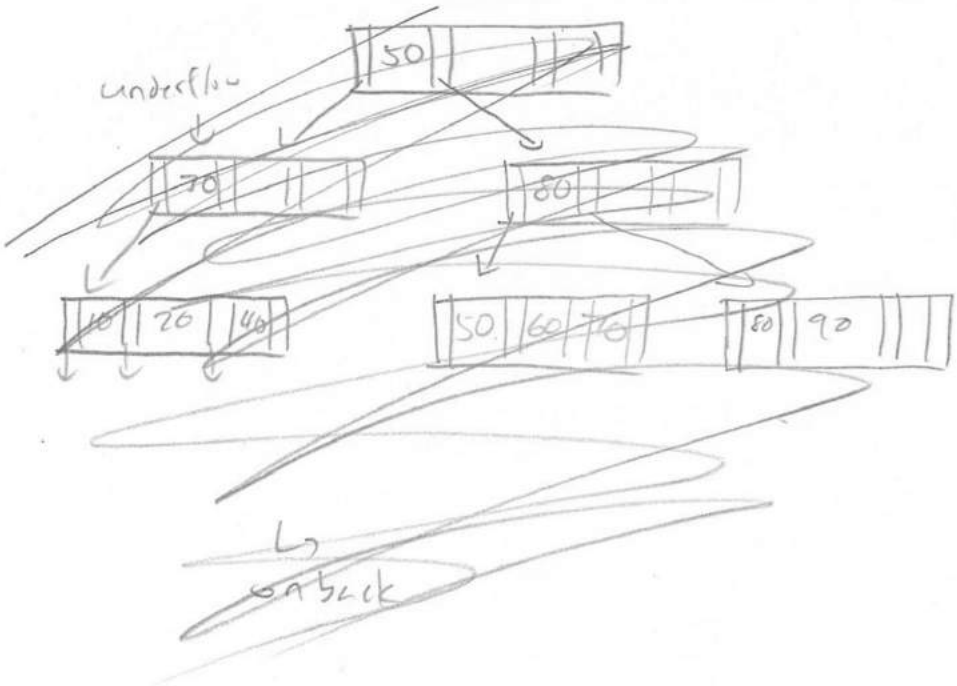


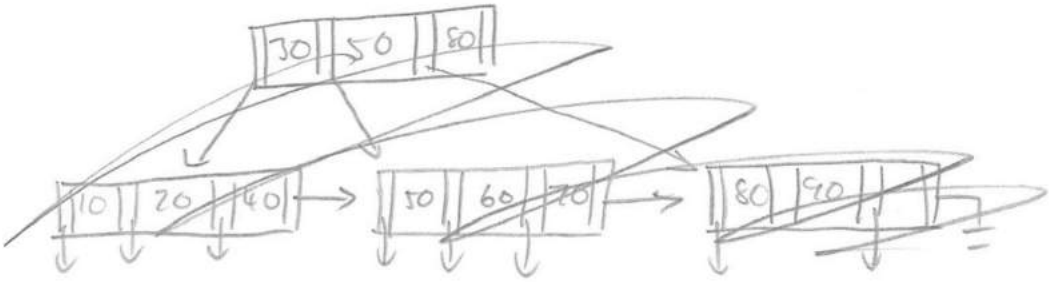
3. Here is another B+tree.

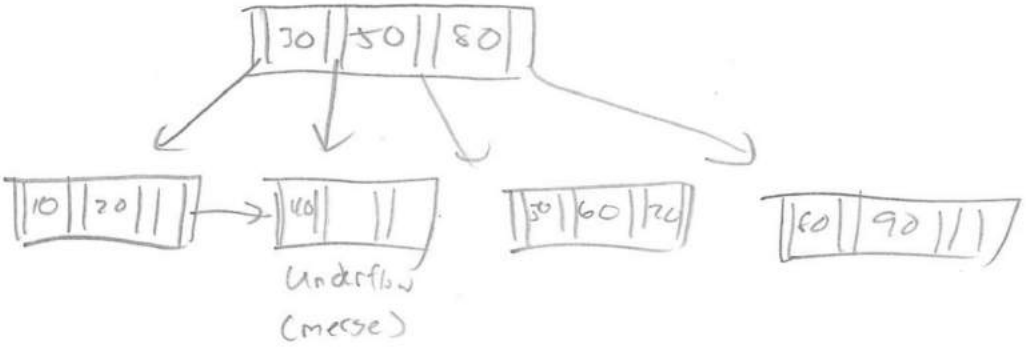


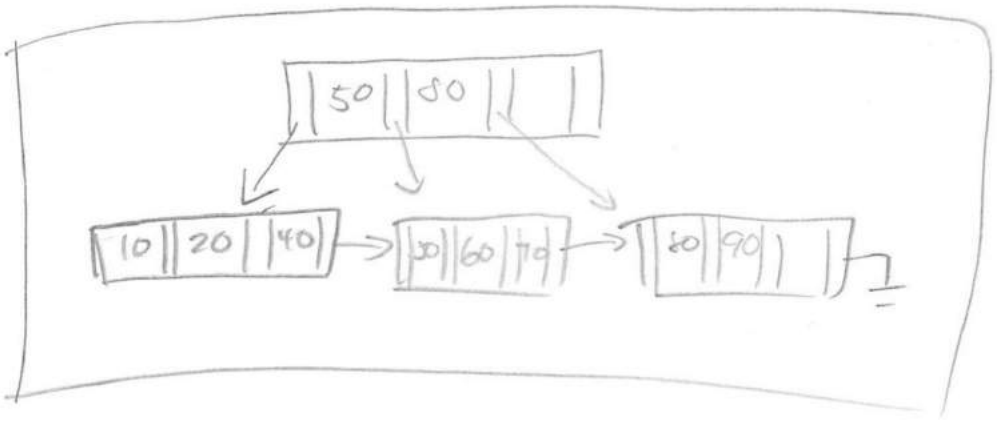
Suppose that we delete the tuple with key 30 from this tree. Draw the new tree after the deletion by using the B+Tree deletion algorithm learned in the class. For partial credit, it may be a good idea to write down every intermediate state of the tree during the deletion. (8 points)






30 50 80
10 20 40
50 60 70
80 90


30 50 80
10 20
40
30 60 70
80 90
Underflow
(merge)


50 80
10 20 40
30 60 70
80 90

Problem 4: Cost of Join (20 points)

Suppose we have 2 tables, $R(A, B)$ and $S(B, C)$, with the following characteristics:

- $|R| = 5,000$ (number of tuples of R), $b_R = 500$ (number of blocks of R).
- $|S| = 10,000$ (number of tuples of S), $b_S = 1,000$ (number of blocks of S).
- Neither table is sorted by any attribute.
- For every tuple of R, there is exactly one matching tuple in S with the same B value.
- We have a dense B+Tree of $n = 50$ constructed on the $S.B$ attribute. The B+Tree is of height 3 with 1 root node, 10 nonleaf nonroot nodes and 200 leaf nodes. Each node of the B+Tree corresponds to one disk block.
- We have 10 blocks of memory buffer that can be used for query processing.

M=10

Consider the following SQL statement:

```
SELECT * FROM R, S WHERE R.B = S.B
```

We decided to execute the above query by performing an index join using the B+Tree on $S.B$. In solving this problem, assume the cost model that we learned in the class. That is, the cost model counts the number of disk blocks read/written during the join, excluding the cost of writing the final result of the join.

1. In the space provided below, provide a bullet list of how you will to use the 10 main memory blocks to perform the index join as efficiently as possible. If you use parts of the memory to "cache" the table and/or index, be specific about what blocks/nodes you will cache. For example, your answer may look like the following:

   - 3 memory blocks to read blocks from R sequentially
   - 3 memory blocks to read and cache blocks of S with the matching tuples
   - 1 memory blocks to buffer the output tuples
   - 3 memory blocks to read and cache B+Tree nodes, one memory block per each level of the tree.

   M=10

   Briefly explain answer. (10 points)

   - 1 memory block to read blocks from R sequentially
   - 1 memory block to buffer output tuples
   - 1 memory block to read blocks from S
   - 1 memory block for the root of the B+tree
   - 6 memory blocks for arbitrary B+tree nonleaf, nonroot nodes

   We need a block reserved for R, S, and output so we can read/write. Increasing the reserved blocks for these won't decrease the number of disk I/os. Therefore, we use the remaining blocks for caching the B+tree. We know the upper levels of the B+tree will be used more frequent so we cache them, prioritizing the upper levels.

2. Under the efficient memory-usage policy, what is the expected index-traversal cost per each tuple of R, excluding the initial caching cost (i.e., the number of disk blocks that need to be read to traverse the index to identify the location of the matching S tuple given an R tuple)? (5 points)

We cache 6/10 nonleaf, nonroot nodes

Therefore, 4/10 tuples require a disk I/o to access the nonleaf, nonroot index node

All tuples require a disk I/o to access the leaf node

1.4

3. If you want to reduce the expected index-traversal cost per every tuple of R to be 0.5 or less (excluding the initial caching cost), what is the minimum number of main memory blocks that you need? Briefly explain your answer (5 points)

An expected index-traversal cost of 0.5 means 1/2 of all tuples require a disk I/o. As a result, this means that half of the leaf nodes must be cached, alongside all nonleaf nodes. Taking into account the blocks needed for R, S, and output, we need 3+1+10+100 blocks.

114 blocks in main memory

Problem 5: NoSQL (10 points)

Consider the table $R(A, B, C)$. All columns of the table are integers. The table has no NULL values.

Assume that the text file `R.txt` contains the same data as R. That is, each line of `R.txt` corresponds to a tuple in R with the column values of A, B, and C appearing in this sequence separated by a space character.

Now someone wrote the following PySpark code to analyze the data in `R.txt`:

```
a = sc.textFile("R.txt")
b = a.map(lambda x: x.split(" "))
c = b.filter(lambda x: int(x[0])>10)
d = c.map(lambda x: (int(x[0]), 1 if int(x[1]) < 50 else 2, int(x[2])))
e = d.map(lambda x: ((x[0], x[1]), x[2]))
f = e.reduceByKey(lambda a, b: a+b)
g = f.filter(lambda x: x[1] > 20)
h = g.map(lambda x: (1, x[0][0]))
i = h.reduceByKey(lambda a, b: a if (a < b) else b)
j = i.map(lambda x: x[1])
j.saveAsTextFile("output")
```

You may assume that the table R and the file `R.txt` has enough tuples to output a non-empty result from the above code. The main question for this problem is given on the next page.

In the space provided below, write a SQL query that returns an equivalent answer to the above PySpark code based on the table R. Write your query succinctly and neatly. You may get as few as 0 point if your solution is far more complicated than necessary, or if we cannot understand your solution. Note that your SQL query just needs to return an equivalent answer. It does *not* have to save the result in the `output` file/directory as the PySpark code does.

```
SELECT B
FROM R
WHERE A > 10 AND B > 20;
```

Problem 6: Transactions (10 points)

Consider the relation Movie(id, title, studio, budget), and the following transaction T:

```
(Q1) SELECT SUM(budget) FROM Movie WHERE studio = 'Disney';
     <other SELECT statements that only READ data from the database>
(Q2) SELECT SUM(budget) FROM Movie WHERE studio = 'Disney';
COMMIT
```

1. Suppose all other transactions in the system are declared as SERIALIZABLE, and they involve only SELECT statements and an UPDATE of an existing tuple on the budget attribute. What is the weakest isolation level needed for transaction T to ensure that queries Q1 and Q2 will always get the same result? Circle one and briefly explain your answer:

   1. READ UNCOMMITTED -> allow DR, NR, P
   2. READ COMMITTED -> allow ~~DR~~, NR, P
   3. REPEATABLE READ -> allow P
   4. ~~SERIALIZABLE~~

   Q1 and Q2 will always get the same result if non-repeatable reads are not allowed. The weakest level to ensure this is REPEATABLE READ.

2. Suppose all other transactions in the system are declared as SERIALIZABLE, and we know nothing else about them. What is the weakest isolation level needed for transaction T to ensure that queries Q1 and Q2 will always get the same result? Circle one and briefly explain your answer:

   1. READ UNCOMMITTED
   2. READ COMMITTED
   3. REPEATABLE READ
   4. SERIALIZABLE

   Since we don't know if an INSERT occurs in another transaction, we must use SERIALIZABLE. A phantom may cause the SUM() aggregates to return different values if this were to occur.