Ankit Sati – as14128
Akshat Jain – aj3186

1. For each of the following example executions, determine if it is serializable, assuming each active transaction ultimately commits. Say why or why not.

A. Read1(x); Write2(x); Write2(y); Read3(y); Read3(z)

The transactions above have the following conflicts:
T1 → T2
T2 → T3

The graph of the above transaction is as follows:
T1 → T2 → T3

Since there exist no cycles, we can conclude that the transactions are **Serializable**. The order of the serial execution is as follows:

Removing the node (transaction) with no incoming edges first yields:
T1 T2 → T3
Remove the next node with no incoming edges yields:
T1 T2 T3

Hence, the serial order of execution is T1, T2, and T3.

B. Read1(x); Write2(x); Write2(y); Read3(y); Read3(z); Write1(z)

The transactions have the following conflicts:
T1 → T2
T2 → T3
T3 → T1

The graph of the transaction is as follows:
T1 → T2 → T3
^ |
|_____|

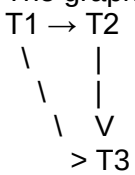
There does exist a cycle in the serialization graph, transactions are **Not Serializable.**

C. Read1(x); Write2(x); Write2(y); Read3(y); Write1(z); Read3(z)

The transactions have the following conflicts:
T1 → T2

T2 → T3
T1 → T3

The graph of the transaction is as follows:

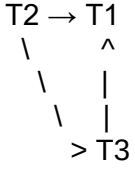


We can see that there are no cycles. Hence, we can conclude
This is **Serializable.**

D. Read1(x); Write2(y); Read1(y); Write2(z); Read3(z); Write3(z); Read1(z)

The transactions have the following conflicts:
T2 → T1
T2 → T3
T3 → T1

The graph is as follows:



The graph above has no cycles. Hence, we can conclude that it is **Serializable.**

2. Try to complete the proof of the serialization graph theorem by showing the final write portion.

To Prove: Serialization Graph Theorem by showing final write portion.

Theorem: If the serialization graph of computation is acyclic, then every transaction reads the same values and writes the same values as it would in a serial execution consistent with the graph.

Proof: (By Contradiction)

Assumptions: An acyclic serial execution that is consistent with the serialization graph of computation can be looked at as any topological sort of the serialization graph. We aim to prove that the writes in the graph are consistent with the writes in the serial execution.

Since we are proving by contradiction, we will assume that the actual execution and the serial execution both have different final writes. Let them be Wp(x) and Wq(x) for the respective executions.

Now, we know that,

Wp(x) belongs to transaction Tp, and

Wq(x) belongs to transaction Tq,

Hence they must be connected in the serialization graph. There are two ways that is possible, they are:

1. $T_p \rightarrow T_q$: This violates the assumption as Wp is not the final write in the actual execution.
2. $T_q \rightarrow T_p$: This violates the assumption as Wq is not the final write in the serial execution.

This implies that Wp and Wq cannot be different and they must both be final writes. We can conclude that Tp and Tq must be the same transaction. And the final write must be same in both the execution graph and the final execution for any x. Hence proved.

3. Consider the two-phase locking protocol with reads and writes. Is two-phase locking deadlock-free? If so, prove it. If not, illustrate a deadlock situation. That is, you must illustrate a situation in which there is a cycle in the waits-for graph.

Two-phase locking is not deadlock free. For example, consider the situation where in transaction T1 acquires a lock on an item y and another following transaction T2 wants to acquire a similar lock on y as well. Likewise, T2 has acquired a lock on another item x and T1 wants to acquire a similar lock on x. In this way, neither T1 nor T2 would release their locks before they complete their execution and neither of them would be able to finish execution because they won't be able to reach their lockpoints. This creates a situation where one transaction waits for the other and vice versa, creating a potential deadlock. Their waits-for graph will contain a cycle indicating a deadlock situation.

Example executions: R1(y)R2(x)W2(y)W1(x)

With the locks, RL1(y) $\rightarrow$ WL2(y)

RL2(x) $\rightarrow$ WL1(x)

i.e., $T_1 \rightarrow T_2$,

$T_2 \rightarrow T_1$

We can observe a cycle ($T_1 \rightarrow T_2 \rightarrow T_1$) indicating a deadlock.

4. In class, we discussed strict two-phase locking which is two-phase locking with the added constraint that all locks must be released at the end of the transaction and a transaction acquires locks immediately before it first accesses a database item. Suppose you remove these added constraints (so locks can be released earlier than the end of the transaction provided no new locks are acquired afterward and locks may be obtained way before they are needed), giving "pure" two-phase locking.
An execution is order-preserving serializable if it is serializable and the following holds: whenever all operations of some transaction T1 precede all operations of some other

transaction T2 (i.e. whenever T1 completes before T2 begins), then there exists an equivalent serial execution in which T1 precedes T2. Prove that pure two-phase locking produces order-preserving serializable executions.

Solution:

Step 1: Prove that Pure two phase locking produces serializable executions.

Lets assume that there are T1, T2 transactions and there exists a cycle such that ($T_1 \rightarrow T_2 \rightarrow T_1$). Both the transactions have their lockpoints where it needs all the required locks to execute. Lets call that point as LP(T1) and LP(T2).

Now, since there is an edge in the graph from T1 to T2, it means that there is atleast one lock that T2 requires from T1 which implies that the lockpoint of T2 should be after lockpoint of T1. i.e. LP(T1) < LP(T2).

But according to the graph, we also have an edge from T2 to T1. That would mean that LP(T1) > LP(T2) which is a contradiction to our previous conclusion.

This implies that our original assumption was wrong. Hence there cannot be a cycle and it is hence proved that pure two phase locking produces serializable executions.

Step2: Prove that Pure two phase locking is order-preserving.

Lets assume, for some T1 and T2, all operations of T1 precede all operations of T2. Two cases are generated:

1. T1 and T2 access a common element n. Then T1 accesses n before T2 accesses it and consequently T1 will release it before T2 starts its execution. This implies that the lockpoint of T1 will naturally precede the lockpoint of T2. Hence, in the serial order T1 precedes T2.
2. T1 and T2 don't access a common element. This doesn't affect their serializability so T1 can precede T2 in the serial order.

Thus, we have proved that Pure phase locking can produce order-preserving serializable executions.

5. Imagine an intention locking protocol in which normal locks are acquired in a two-phase manner, but intention locks can be released at any time. Would such a protocol be serializable? If so, prove it. If not, show a counterexample.

The protocol is **NOT Serializable.**

Example:

Transaction 1: Write to File1 in the Database
Transaction 2: Read from Database

The protocol would yield the following execution:

T1 acquires an intention write-lock on the Database

T1 acquires an intention write-lock on File
T1 acquires a write-lock on record1
T1 releases the intention write-lock on File
T1 releases the intention write-lock on the Database
T2 acquires an intention read-lock on the database
T2 acquires a read-lock on the File
T2 releases the intention read-lock on the Database
T1 writes end
T2 reads end
T1 and T2 release their locks.

In the above executions, the result of T2 can be corrupted as T1 is modifying the data in the Database while T2 is reading the data. Thus, we can conclude that the above execution can't be serialized and thus the protocol is **Not Serializable.**

6. Recall the Kung and Robinson optimistic concurrency control algorithm as discussed in class. How would you modify that algorithm to ensure that every transaction eventually completes (i.e., no transaction is restarted forever)?

Modified algorithm:

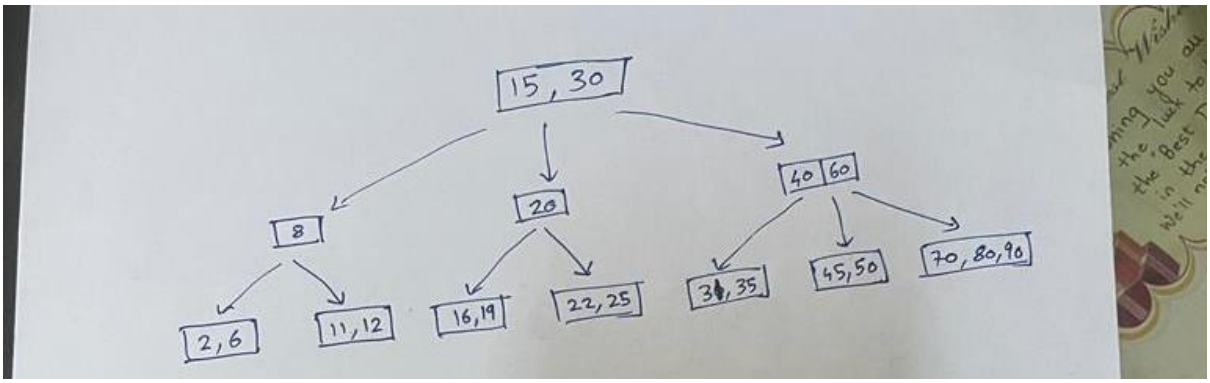
1. For each transaction: note transaction start time, abort count
2. Wait for transaction to be ready to commit
3. Let T <- ready to commit transaction
4. Check Conflicts of T
5. If no conflicts: execute T
6. Else:
    a. T <- earliest start timed transaction
    b. Abort and increase abort count
    c. in case of multiple transactions with the same start time, chose by giving priority to abort count.
    d. Execute T

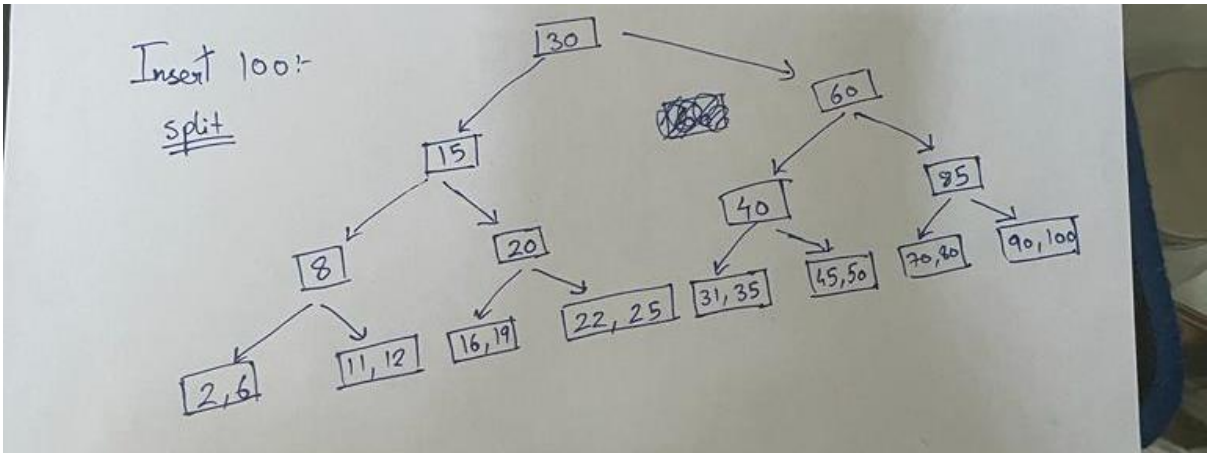
With this way, we can avoid starvation, even if there is a potential conflict.

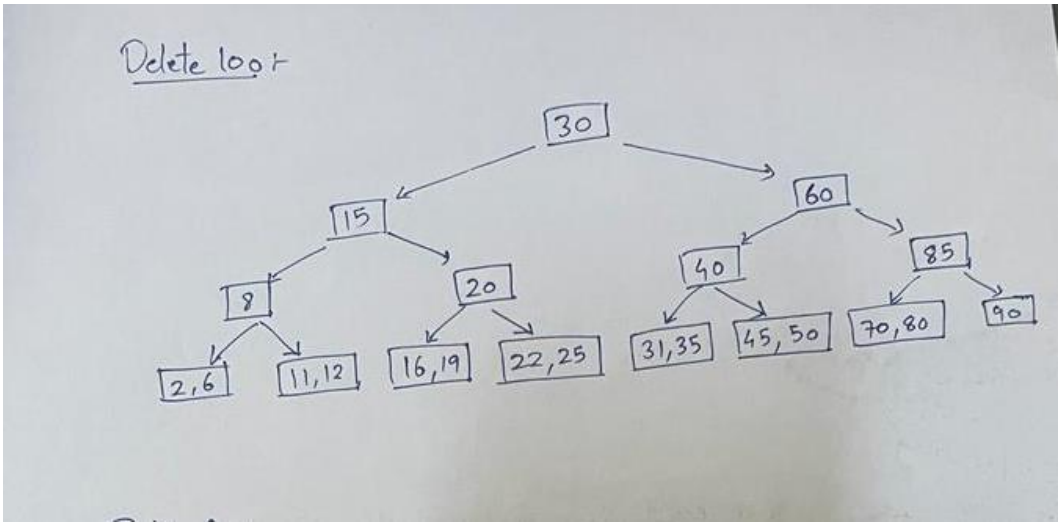
7. Consider a B+ tree allowing splits and free-on-empty. In structure, this is a standard B+ tree (no "horizontal" links, only parent to child pointers). Free-on-empty means that the B+ tree does not use merges, but rather waits until a node is empty before freeing it. Different processes execute on the B+ tree in a concurrent fashion and a process may delay arbitrarily long. Assume that each split works as follows: lock a node n, transfer the highest half of the keys from n to some newly allocated node n', adjust the fences (representing the insets) appropiately, release lock on n, lock parent(n), change pointers and key separators in parent(n) appropriately, and release lock on parent(n). Assume that free-on-empty works like this: lock(n), confirm that n is empty, set its fence to the empty set (this is a shorthand for the following operation: the fence field in node n contains what the inset should be; normally that value is an interval between a and b where a and

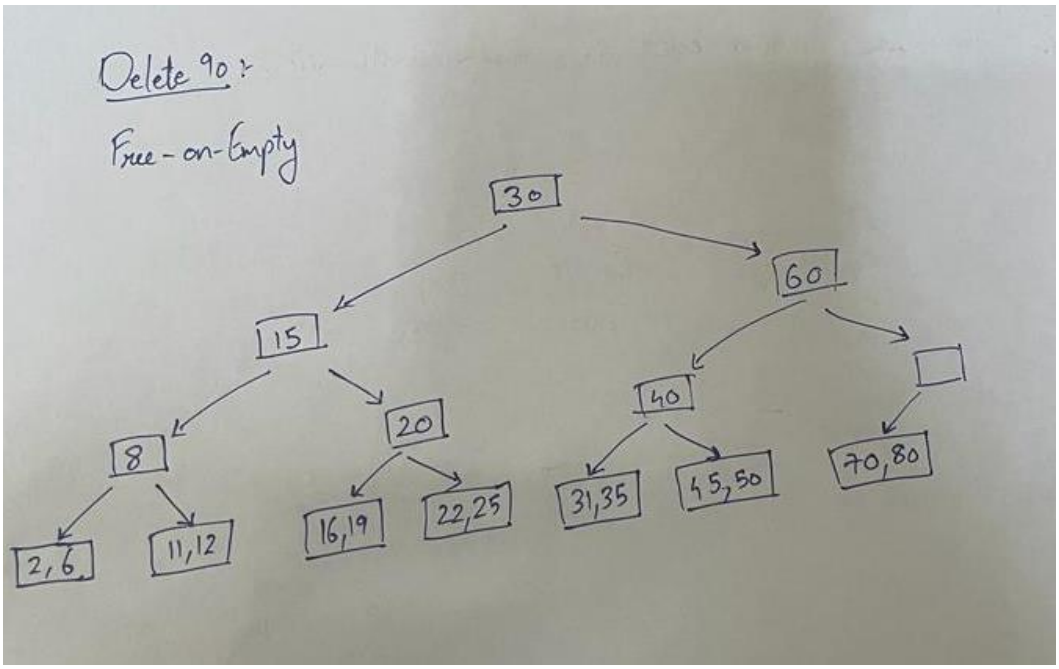
b are the separators of parent(n); in this case however we will set the fence to the empty set as a signal to searchers that this node will soon have an empty keyset), release lock on n, lock parent(n), adjust pointers in parent(n), release lock on parent(n). Here are your tasks: (a) Please show an example of these structure-changing operations on a data structure that starts with 15 data items (all data items in a B+ tree are at the leaves), a fanout of three (each interior node has at most three children) , and at most three data items per leaf node. That is, show an insert that causes a split to happen and a delete that causes a free-at-empty to happen. Say how the fences will change. (b) Specify the search algorithm (hint: use a give-up scheme, but propose a give-up scheme that goes somewhere below the root); (c) Show that the structure and search invariant conditions hold as specified in the lecture notes on concurrent search structure algorithms. (d) Say when to free a node that has had its inset(n) set to empty.

(a) This is our initial B+ tree









(b) Algorithm:

Start at some non-leaf node n

While x not in n:

If x in keyset(n):

Return x not found

Elif x in inset(n) and edgeset(n,n'):

n = n'

else: set n to some ancestor node

if n is leaf:

return x as key and its value

else: return x not found

(c)

For searching in an item B beginning at node m, the following invariant holds:

A path from node n1 to node n2 exists such that B is in keyset(n2) and every edge E along that path has B in its edgeset. If the search/insert/delete for item B is at node n1, then B is in keyset(n1) or there is a path from n1 to node n2 such that B is in keyset(n2).

Above hold true for each element of the set if searching for a set.

The invariant generally applies to the B-trees described in the class, but in our instance, since non-leaf nodes only store indexes rather than actual data, we must locate data items in leaf nodes. Keep in mind that we have made sure that no values in nonleaf nodes ever overlap with information in leaf nodes.

In order to make the searching algorithm operate properly with our B-tree, we compel it to find something in leaf nodes; otherwise, the result is invalid.

(d) When the node is a leaf or all its children are empty, we free a node with its inset(n) set to empty.

8. Show that the multi-version read consistency algorithm ensures serializability. That is, read-only transactions use the multi-version technique whereas read-write transactions use strict two-phase locking.

We have already demonstrated that serializability is guaranteed by strict two-phase locking. Since all of the read-write transactions are serializable, we can be sure that they will all occur in a sequential manner every time. Next, we take into account all read-only transactions. Because read-only transactions employ multi-version read consistency, even when concurrent changes are being made, we are saving the previous copies of the transactions as we proceed. As a result, we are aware that the outcome of each read-only transaction Tpx, originates from a previous copy.

Let us assume two read-write transactions Tpi and Tqj from a serialized order, and we have a read-only transaction Tk. Since we already know that the read-write transactions (Tp and Tq) happen one after the other, implying Tpi would happen before Tqj. Assuming that Rk of Tk happens after Tpi is committed and before Tqj is committed, it will appear to be reading all the data items that we committed by Tpi. Hence, we can insert Rk of Tk in between Tpi and Tqj. Similarly, this can be done for all the read-only operations.

In this approach, we demonstrate that the multi-version read consistency method achieves serializability by identifying a serial order for all read-write and read-only transactions combined.

9. In the Oracle database system, transactions (including read-write ones) use multi-version read consistency for any SELECT statement, but acquire exclusive row locks for SELECT FOR UPDATE statements and hold those locks until the end of the transaction (including an end-of-file lock to prevent phantoms). Suppose some transactions in an execution issue SELECT statements followed by INSERT statements and issue no SELECT FOR UPDATE statements. (INSERT statements also acquire exclusive locks on the data items they insert and hold those locks until the end of the transaction.) Can this lead to a non-serializable execution? If so, show one. Otherwise, say why not.

Yes, this might lead to non-serializable execution. For example if we had an empty table T <name, age> and the following transactions:

T1: (Select name from T where age>30)

T2: Insert into T Values (“Cat”, 34)

T1: (Select name from T where age>30)

We can see that there are phantom reads in T1 so the execution order is not serializable. This would give us different read results within a single transaction because the SELECT statement didn’t block its effective range and let T2 commit a change before it executed.

10. (AQuery) (i) Given a schema ticks(ID, date, endofdayprice) find the maximum and minimum return of each stock where return is the ratio of the price of day i compared with the price of day i-1 for running ratios, you can use the ”ratios” function (e.g. ratios(price), where price is a column). Use AQuery built-ins wherever possible.

```
aj3186@access2:~/ADB/AQuery2
[aj3186@access2]~/ADB/AQuery2% python3 prompt.py
true
running
-1 (null)
> f stock.a
> exec
QCREATE TABLE IF NOT EXISTS ticks(ID VARCHAR(20), date INT, endofdayprice INT);
QCOPY OFFSET 2 INTO TICKS FROM  '/home/aj3186/ADB/AQuery2/ticks.csv'  ON SERVER    USING DELIMITERS  ',';
QSELECT ID, endofdayprice  FROM ticks ORDER BY ID, date  ;
Pdll_4gC8Vc
g++-11.2 -shared   -fPIC -include server/pch.hpp out.cpp libaquery.a  -I/home/aj3186/ADB/centos/usr/include/monet
onetdbe -O3 -DNDEBUG -fno-stack-protector  -fno-semantic-interposition -march=native -flto  --std=c++1z -o dll.so
Exec Q0: QCREATE TABLE IF NOT EXISTS ticks(ID VARCHAR(20), date INT, endofdayprice INT);
 Exec Q1: QCOPY OFFSET 2 INTO TICKS FROM  '/home/aj3186/ADB/AQuery2/ticks.csv'  ON SERVER    USING DELIMITERS  ',
 Exec Q2: QSELECT ID, endofdayprice  FROM ticks ORDER BY ID, date  ;
 Dbg: Getting col id, type: monetdbe_str
Dbg: Getting col id, type: monetdbe_str
Dbg: Getting col endofdayprice, type: monetdbe_int32_t
Dbg: Getting col endofdayprice, type: monetdbe_int32_t
id | maxreturn | minreturn
==========================
3003 83.5 0.037037
3002 190 0.00869565
3001 117 0.00571429
3000 124 0.00909091
done.
>
```

DROP TABLE IF EXISTS ticks
CREATE TABLE ticks(ID varchar(20), date int, endofdayprice int)
LOAD DATA INFILE "ticks.csv" INTO TABLE TICKS FIELDS TERMINATED BY ","
select ID, max(ratios(1, endofdayprice)) as maxReturn, min(ratios(1, endofdayprice)) as minReturn
from ticks assuming asc ID, asc date
group by ID

Max return for 3000 is 124
Min return for 3000 is 0.0091
Max return for 3001 is 117
Min return for 3001 is 0.00571
Max return for 3002 is 190
Min return for 3002 is 0.0087
Max return for 3003 is 83.5
Min return for 3003 is 0.0370

(ii) If ticks are already sorted by ID, I could directly perform the GroupBY operation on ID and have the ticks in groups of their IDs. Once done, we can perform a sort by date within each group and then traverse each group performing the required calculations. We will maintain to values maximum and minimum within each group and we could get our required solution with a

single pass. Furthermore, AQuery allows us to do all of the above operations by providing required built-ins. This would drastically optimize our original process for large entries.