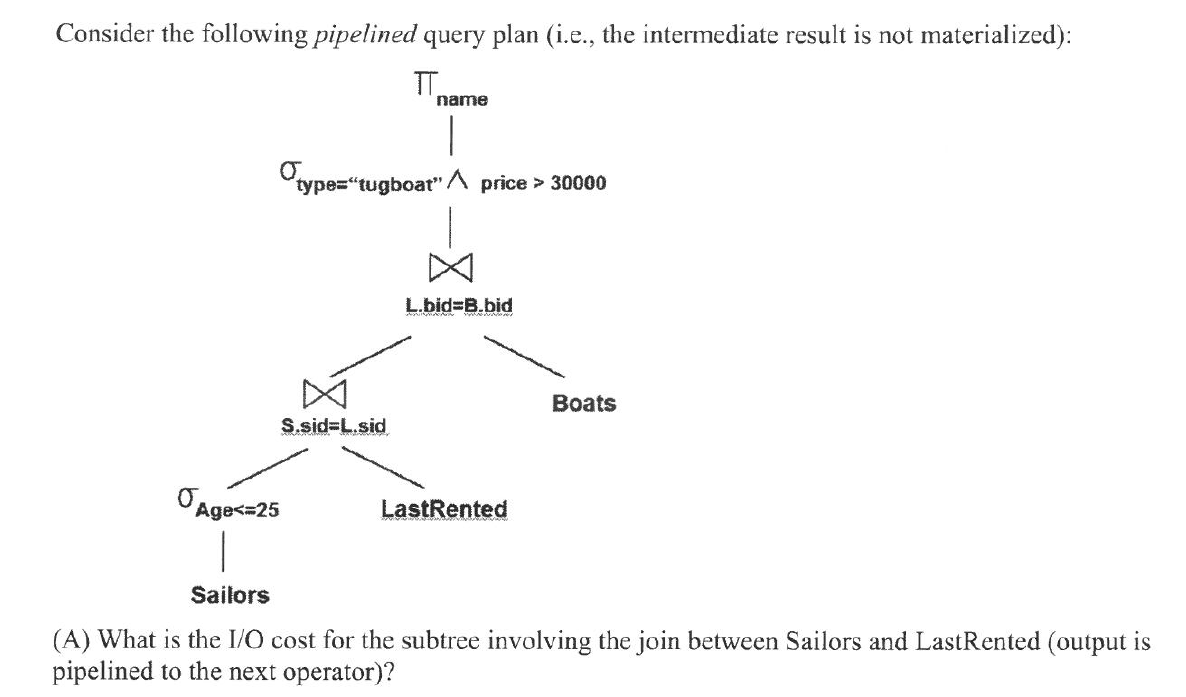
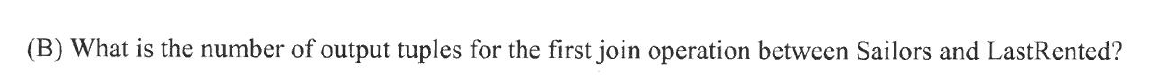
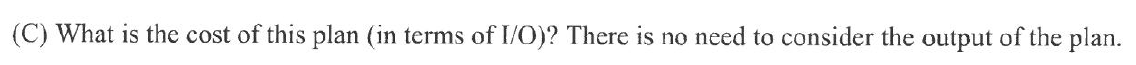
Q1



Page Nested Loop Join S, LR = |Sailors| \* |LastRented|= 20 \* 100 = **2000** I/O



# tuples in intermediate result = **20000** (every tuple in lastRented will match with some tuple in Sailors due to foreign key relation, and assuming independence of attributes btw S.Age and S.sid)



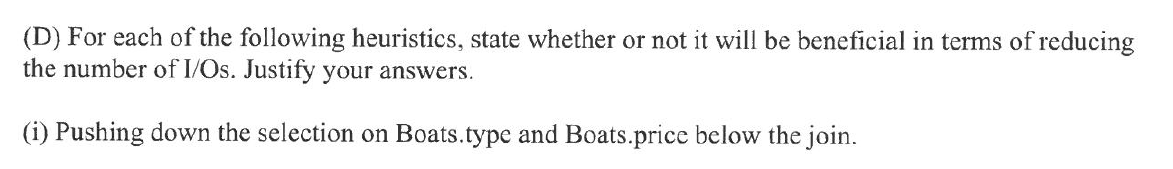
(ignore cost for writing out final output)

For each tuple in intermediate result, probe B+ tree for Boats

* No match = 3 I/O
* Match = **4** I/O (all will match since L.bid is a foreign key)

Index Nested Loop Join = 20000 \* 4 = **80000** I/O

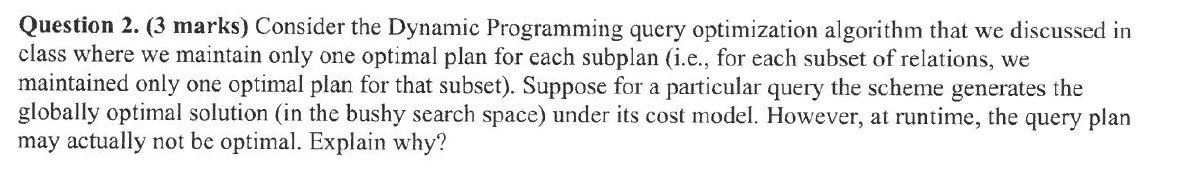
**Total cost** = 80000 + 2000 = **82000** I/O



Yes. Even though this prevents us from making use of the B+ tree index on Boats.bid, pushing down the projection of Boats.type and Boats.price will result in a significantly smaller proportion of Boat tuples (10% \* 50% = 5%, assuming independence of attributes and uniform distribution) to be checked for when performing the join between the intermediate result and the Boat table. The number of tuples is small enough to fit in a page and can possibly fit in the buffer, hence removing the need for the additional 80000 I/Os in performing the join.



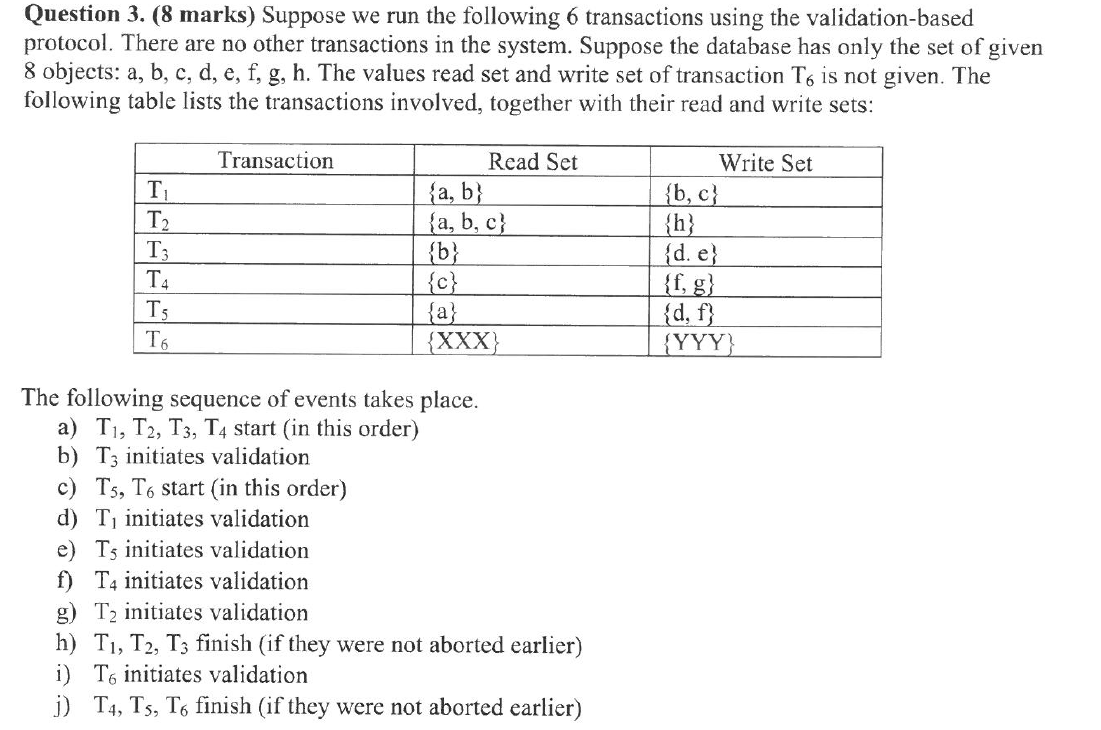
No, this will not reduce the number of I/O. Since the pipelined approach is used, the number of I/Os for scanning Sailors tuples will still be the number of pages in the Sailors table.



**DP query optimization (System R) can result in Bushy Trees? Thought it is left-deep tree only.**

The question is essentially saying: even if we had allowed bushy plan (which would also means extending the search space to bushy - no longer restricted to left-deep), the generated optimal plan (based on the cost model) may still not be the actual optimal plan at runtime. You are required to explain why.

The query plan may not be optimal because the true globally optimal plan could be in the right-deep tree search space, and therefore not covered within the left deep tree and bushy tree search space. Another possible reason might be because the optimization overhead would be much higher, since for each subset of relations in the DP algorithm, there are more cases to consider, leading to O(3^k) instead of O(2^k) time complexity.



b) T3 COMMITTED

d) T1 COMMITTED

* R and W set of T1 does not overlap with W set of T3.

e) T5 ABORTED

* R and W set of T5 OVERLAPS with W set of T3.

f) T4 ABORTED

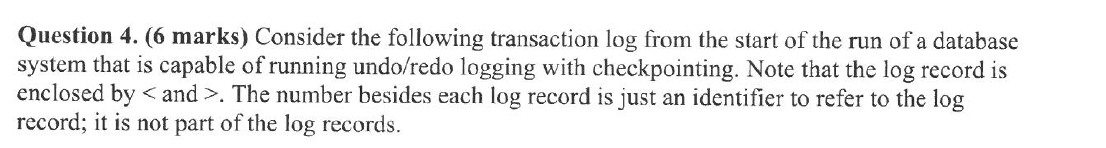
* R and W set of T4 does not overlap with W set of T3.
* R and W set of T4 OVERLAPS with W set of T1.

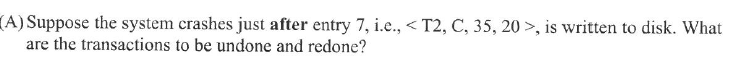
g) T2 ABORTED

* R and W set of T2 does not overlap with W set of T3.
* R and W set of T2 OVERLAPS with W set of T1.

i) T6 COMMITTED

* R set of T6 does not overlap with W set of T3.
* R set of T6 does not overlap with W set of T1.
* XXX = any permutation of {a,f,g,h}
* YYY = anything



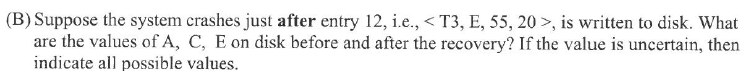


Undo:

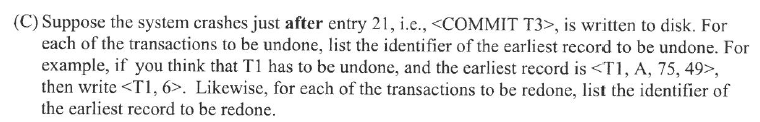
T1 and T2 need to be undone (log 7,6,4,2)

Redo:

None



|  |  |  |
| --- | --- | --- |
|  | Before Recovery | After Recovery |
| A | 49 or 75 (event 2 is alr reflected on disk) | 75 (redo T1) |
| C | 35 or 20 | 20 (undo T2) |
| E | 20 or 55 | 20 (undo T3) |



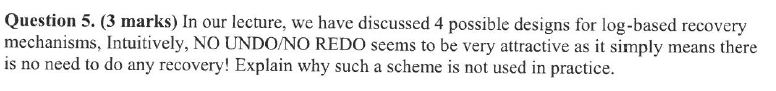
Undo:

<T4, 16>

Redo:

T3 has no actions to be redone because it is committed and its last action is before the start CKPT.

<T2, 14>



Such a mechanism will have the negative side-effects of both force and no-steal. The force characteristic incurs high number of random I/Os as all dirty pages need to be flushed to disk whenever a transaction commits. No-steal could be memory costly because all updates made by a transaction must be stored in memory until the transaction commits. Furthermore, adopting a scheme of no recovery (maximum guarantee of correctness) on a system implies the system has a high tendency to fail in the first place, which is unlikely in practice as it implies a design flaw in the system.