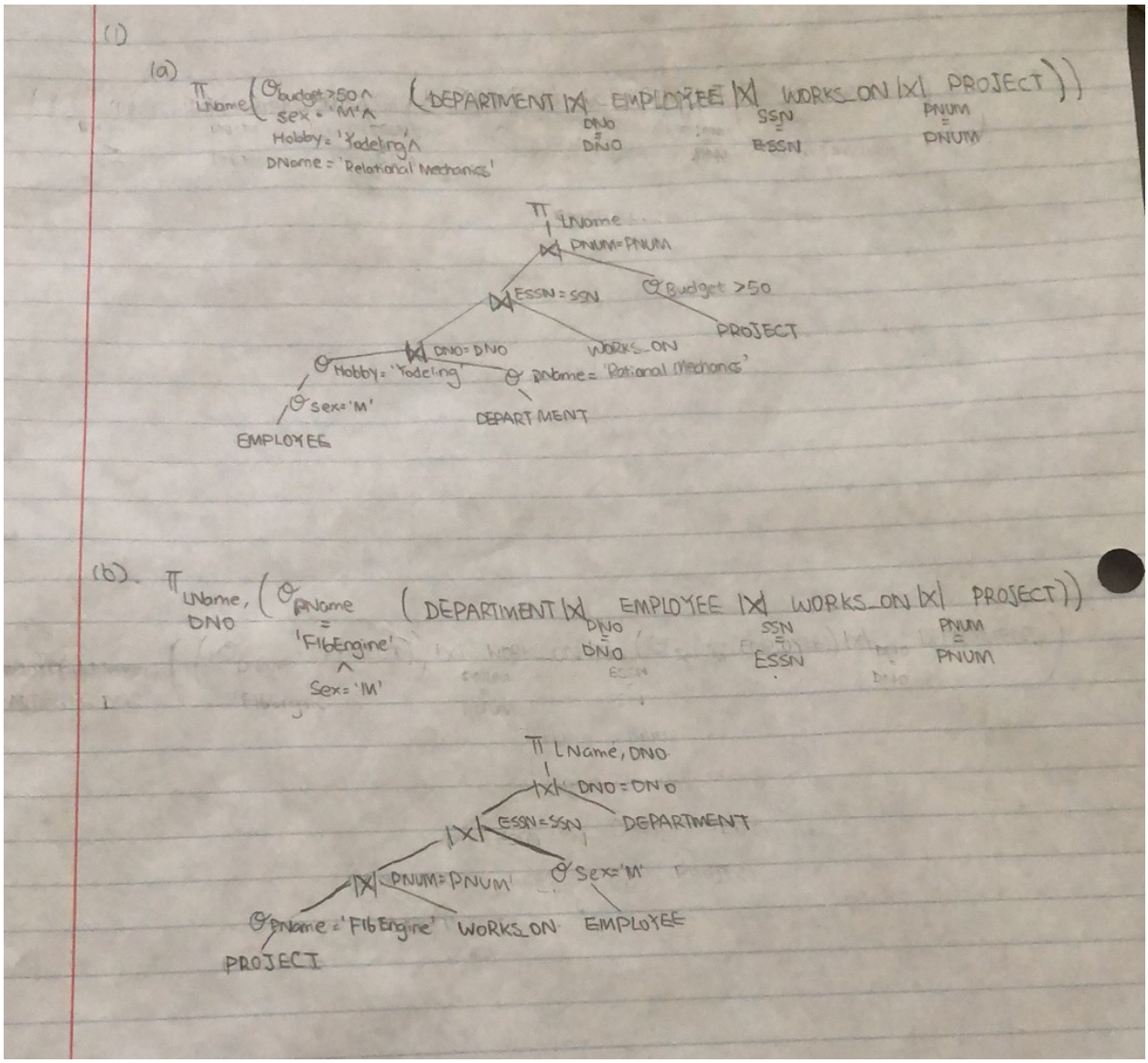
(1)



(2)

(a)

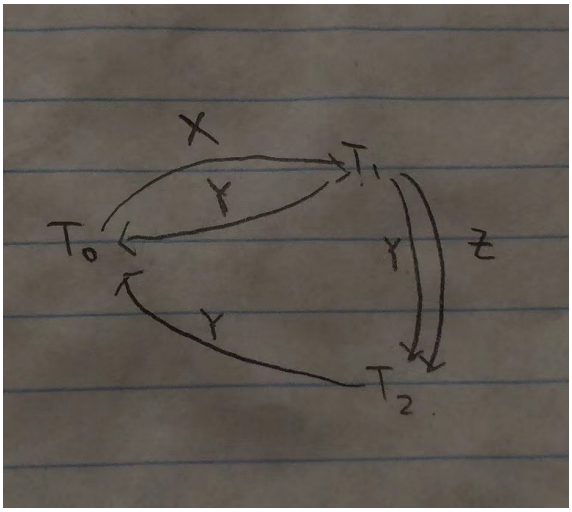
T_0	T_1
read_item(A) read_item(B) if A=0 then B :=B+1 write_item(B)	read_item(B) read_item(A) if B = 0 then A:=A+1 write_item(A)

(b)

T_0	T_1
read_item(A) read_item(B) if A=0 then B :=B+1 write_item(B)	 read_item(B) read_item(A) if B = 0 then A:=A+1 write_item(A)

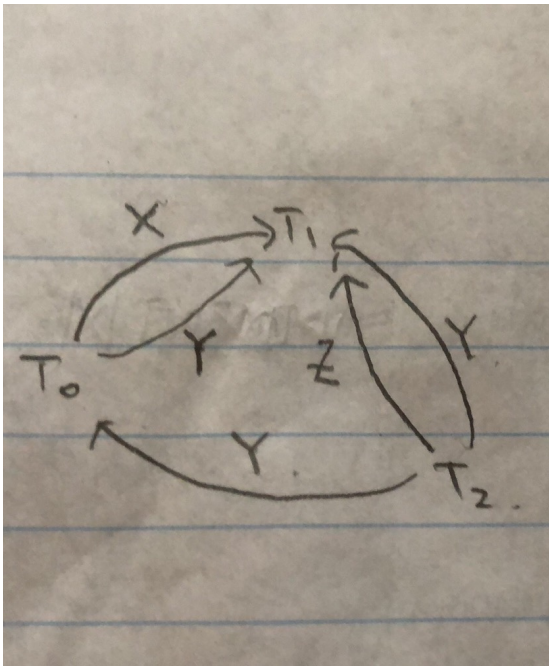
(3)

This transaction is not conflict serializable, since a cycle is in the precedence graph.



(4)

This transaction is conflict serializable, since there is no cycle in the precedence graph.



An equivalent serial schedule would be the following:

T_0	T_1	T_2
read_item(X) write_item(X) read_item(Y) write_item(Y)	read_item(Z) read_item(Y) write_item(Y) read_item(X) write_item(X)	read_item(Y) read_item(Z) write_item(Y) write_item(Z)

(5)

The transaction is not conflict serializable since there is a cycle in the graph. The following path makes a cycle in the graph: $T_1 \rightarrow T_2 \rightarrow T_4 \rightarrow T_5 \rightarrow T_3 \rightarrow T_1$

(6) The transaction is conflict serialization since there is not a cycle in the graph. The following path is a valid serial schedule: $T_4 \rightarrow T_5 \rightarrow T_1 \rightarrow T_2 \rightarrow T_3$

(7)

(a) The San Jose plant location would send the query to 'Boca' plant , and 'Boca' plant would compute and send the results back to San Jose.
(b) The San Jose plant location would send the query to the New York office, and the New York office would compute and send the result back to San Jose.
(c) The San Jose plant location would send the query to Toronto, Edmonton, Vancouver, Montreal and these sites would compute the query results respectively and send the result back to San Jose.
(d) San Jose would send the query to New York, and New York would compute the result and send it back to San Jose.

（8）

(a) First, we send a query to Armonk to retrieve the Plant Location that contains machine number 1130. Then, we send the query to retrieve all employees to the Plant Location returned from the first query. The query would be computed and returned from the respective Plant Location. Finally, we join the results together at our destination location.
(b) First, we send a query to Armonk to retrieve the Plant Location that contains machine type 'Milling Machine'. Then, we send the query to retrieve all employees to the Plant Location returned from the first query. The query would be computed and returned from the respective Plant Location. Finally, we join the results together at our destination location.
(c) We send a query to Armonk to retrieve Machines with Plant Location at "Almaden". The query would be computed and returned from Armonk to our destination location.
(d) We first send a query to do GROUPBY based on PlantLocation to Armonk to retrieve the corresponding machines for each of the location, and then send the machine relation, along with a query to join EMPLOYEE at each PlantLocation. Finally, we union the corresponding results returned from each PlanLocation at the destination location.

(9)

(a) r ⋈ s:

A	B	C	D	E
6	4	3	4	3
2	3	4	5	6
2	3	4	7	9

s semijoin r:

C	D	E
4	5	6
3	4	3
4	7	9

(10)

The hash function usually distributes key values randomly and uniformly throughout all buckets, thus making it not the best choice for search key on range queries since it has to go through all buckets. However, if key values in the range are assigned to occupy consecutive locations in the bucket with order, then it would make hash structure more optimal.

(11)

1. First the hash join algorithm adds the join attribute of B to the hash table, then hashes on A into the same table
2. If a match is found for the attribute of B, then the join that attribute succeeds, and A ⋈ B is added to the hash table
3. If no match is found for the attribute of B, we fill the void entries with NULL and add it to the table
4. Finally, we add the remaining unmatched entries of A to the hash table and fill void attributes with NULL.

(12)

Suppose we want to learn about a_{10}, then we can write a query to compute the $sum(a_1, a_2, a_3 ..., a_{10})$, and then another query to compute $sum(a_{11}, a_{12}, a_{13} ..., a_{20})$. Then we write a query for $sum(a_1, a_2, a_3, a_4, a_5, a_{11}, a_{12}, a_{13}, a_{14}, a_{15})$, and $sum(a_6, a_7, a_8, a_9, a_{16}, a_{17}, a_{18}, a_{19}, a_{20})$. We then we sum the last two queries, subtract it from the sum of the first two queries to get a_{10}.

(13)

$R_1 \cap R_2 = (A, B, C) \cap (A, D, E) = A$

Since $A \rightarrow BC$, $A \rightarrow ABC$ is in F^+, thus the decomposition is a lossless-join decomposition.

(14)

$E \rightarrow ABCDE,\ BC \rightarrow ABCDE,\ A \rightarrow ABCDE,\ B \rightarrow D,\ CD \rightarrow ABCDE$

(15)

The canonical cover F_c of the set of function dependencies is $\{A \rightarrow BC, CD \rightarrow E, B \rightarrow D, E \rightarrow A\}$

(16)

B^+ of B in F is $B \rightarrow BD$

(17)

Suppose $\alpha \rightarrow \beta$ and $\gamma\beta \rightarrow \delta$, then $\gamma\alpha \rightarrow \gamma\beta$ (by augmentation). Since $\gamma\alpha \rightarrow \gamma\beta$, $\gamma\beta \rightarrow \delta$, then $\gamma\alpha \rightarrow \delta$ (by transitivity). Thus proves pseudotransitivity rule..

(18)

If we want to prove $b \to c$, then we want to prove that each *b* value in *R* is associated with precisely one *c* value in *R*. So we can implement the following:

```
SELECT b
FROM R
GROUP BY b
HAVING count(c) !=1
```

If the query returns an empty set, then functional dependency is satisfied.

(19)

```
CREATE ASSERTION dependecy_preseve CHECK
        (NOT EXIST (SELECT b
        FROM R
        GROUP BY b
        HAVING count(c)  != 1
)
)
```

(20)

The decomposition is not a dependency preserving decomposition. r_1 has the relation dependency $A \to BC$, and r_2 has the relation $E \to A$. However, the relations $CD \to E$ and $B \to D$ is not preserved in $r_1 \cup r_2$. Therefore, $r_1 \cup r_2$ is a subset of r. So the given decomposition is not dependency preserving.

(21)

$R' = (A, B, C, D)$ and assume the set of functional dependencies $F' = \{A \to B,\ A \to C,\ A \to D\}$. Then, since A is the superkey, we can have the following three ways of decomposition:

1. {A,B} {A,C} {A,D}
2. {A,B,C} {A,D}
3. {A,C,D} {A,B}

This is also a lossless decomposition since the intersection of all of these decomposed sets is just A. For 1, since $A \to B$, $A \to AB$. For 2, since $A \to D$, $A \to AD$. For 3, since $A \to B$, $A \to AB$. Thus these are lossless decompositions.