# PROBLEM 1

1.

**A**tomicity, **c**onsistency, **i**solation, **d**uration

---

2.

**Atomicity**: that one transaction gets commited fully or not at all

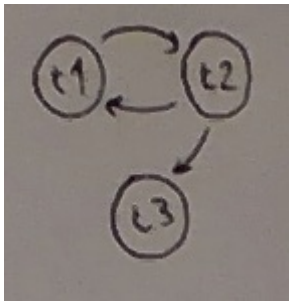
**Concistency**: That transactions happen "symmetrically". If one for example is to transfer funds from one account, one must make sure that the receiver gets the same amount the sender sends. Like the Friends example we did in class.

**Isolation**: That one transaction gets treated isolated and do not "see" parts of other transactions or partially committed transactions.

**Duration**: That the data is durable and does not get lost from system crashes and so on.

---

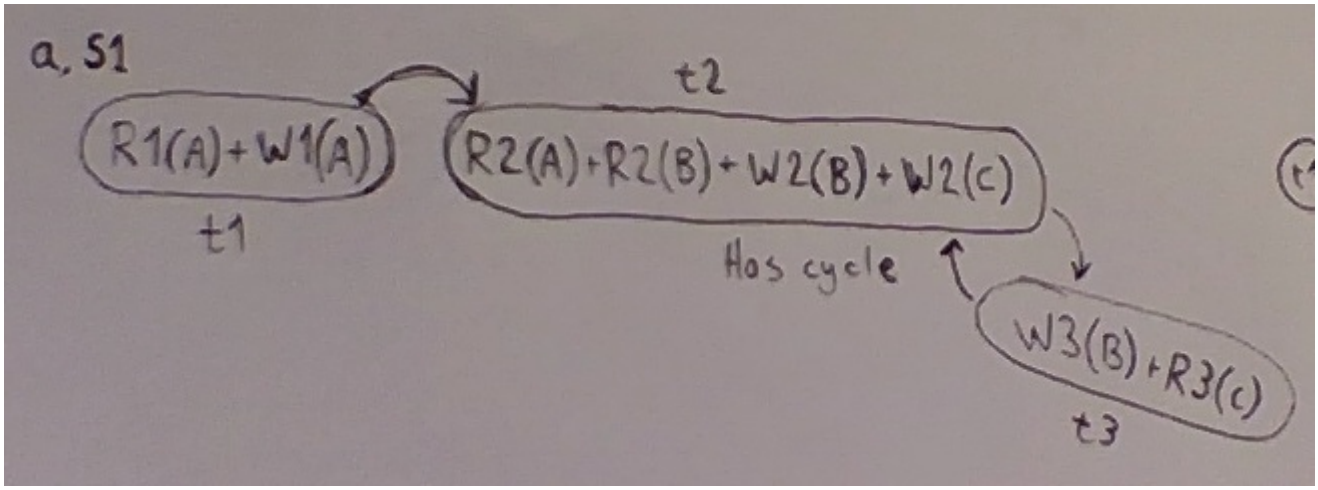
3.




You can check this by seeing if there is one or more cycles. There only needs to be one cycle for it to be a cyclic precedence graph, like in the picture, it does not neccesarily need to go $(t1 \rightarrow t2 \rightarrow t3)^n$ for it to be considered  a cycle.

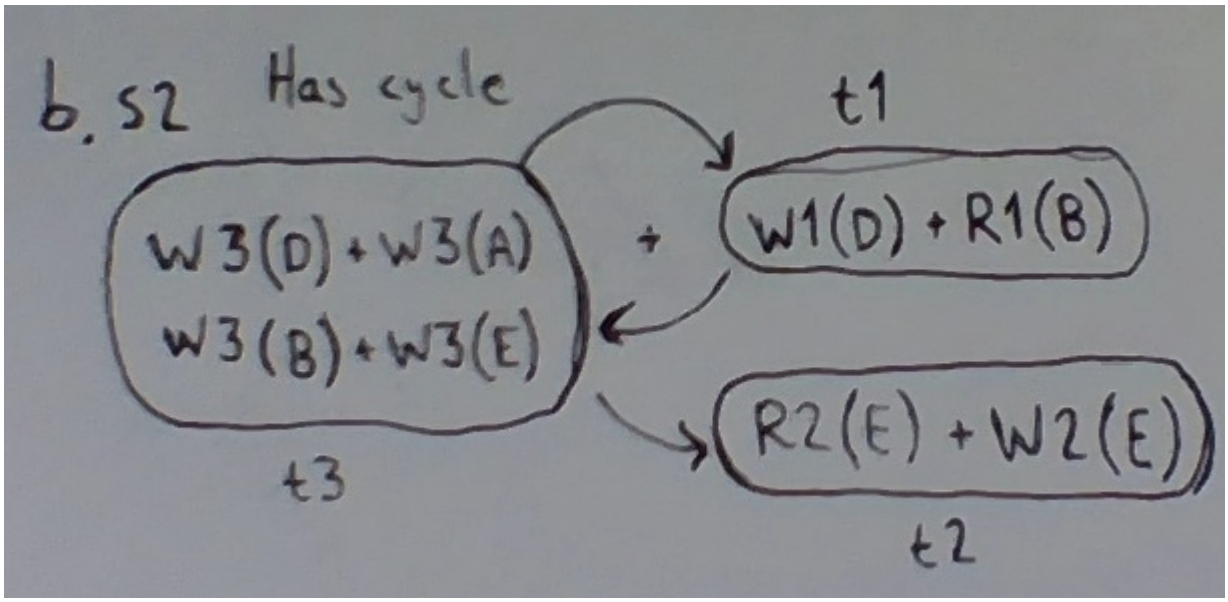
---

4a.



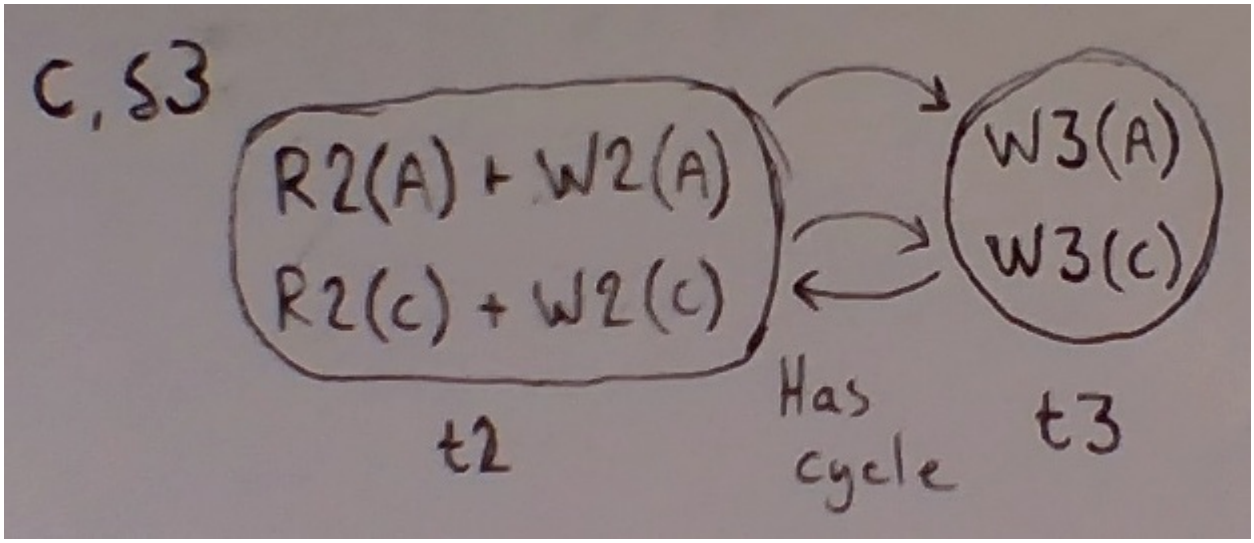

Conflict serializable, has cycle. R3(C) + W2(C) is the conflict.

4b.




Conflict serializable, has cycle. W3(A) + W1(D) is in conflict. R1(B) + W3(B) is in conflict. W3(E) + R2(E) is in conflict. R2(E) + W2(E) is in conflict.

4c.




Not conflict serializable, has cycle. R2(A) + W2(A) not in conflict. W2(A) + W3(A) not in conflict. W3(A) + R2(C) not in conflict. R2(C) + W2(C) not in conflict. W2(C) + W3(C) not in conflict.

---

5.

All the transactions must be redone. From a technically functional perspective T3 and T6 will likely be corrupted files and you do not have to redo the other ones since they are completed. However, from a practical standpoint considering the ACID-properties, since T1 has not been completed before the system failure, this likely will ensue in the following values being skewed and/or wrong. So in a practical viewpoint, one needs to ensure data integrity and therefore one must redo all of the transactions.

---

6.

You rebuild only part of the database. The transactions that were fully committed you can let be, but the ones that got interrupted will need to be rebuilt using the .log-file to "fill in the blanks". The log file logs completed and planned events before the transaction is committed at all. This way you can redo only what needs to be redone instead of stressing about reindexing everything.

---

7.

In the case of figure 1, one would only need to rebuild T1, T3 and T6. However, to ensure data consistency, it is best to redo all of them to avoid domino errors from T1.

---

# PROBLEM 2

1.

``` PostgreSQL
SELECT position, branchNum, city FROM (E.position = 'Manager') AND
(E.branchNum = B.branchNum) AND (B.city = 'New York') AND (E.salary>1000)
WHERE E.name = '{name}'
```

---

2.

E + B = 15

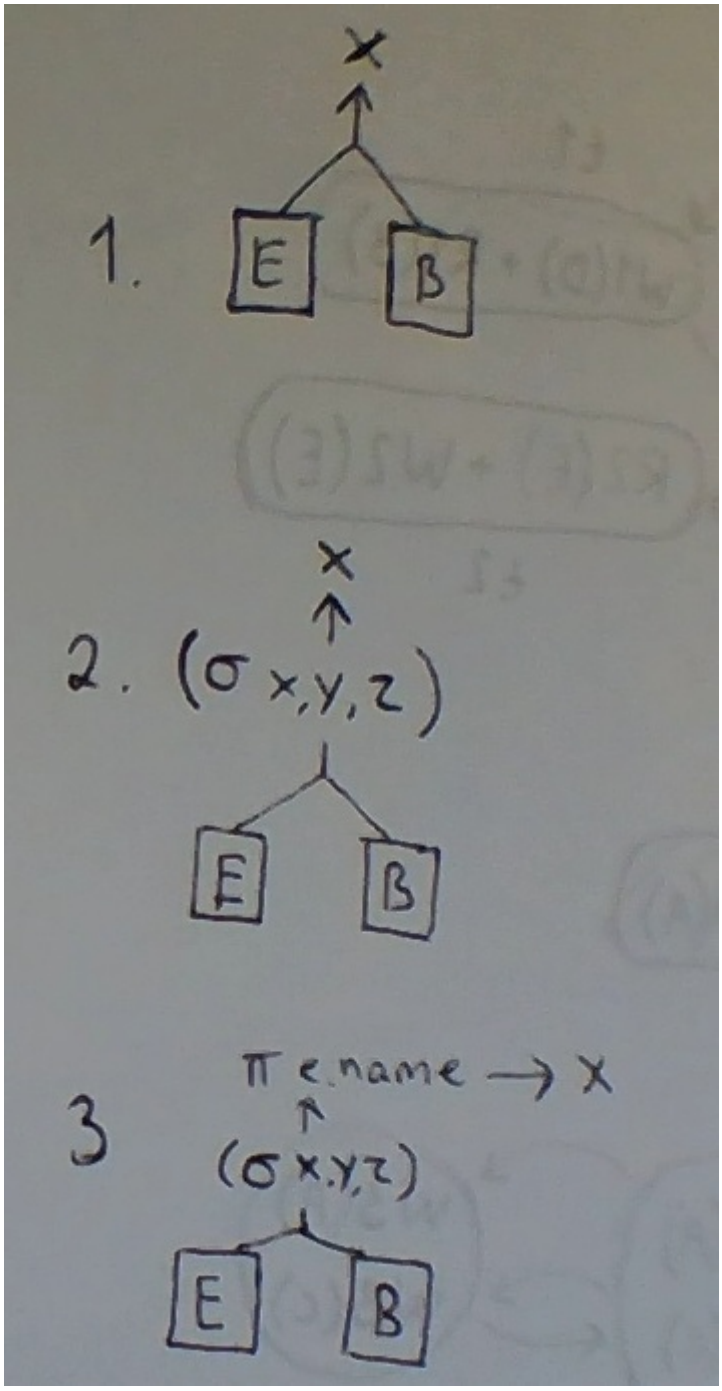
---

3.

See what entities one want attributes from. See what attributes. Limit the query to only include attributes from these entities. In this case that would be entities Employee and Branch. We are only interested in the attributes where Employee.position='Manager', Employee.branchNum = Branch.branchNum, Branch.city='New York' and Employee.salary>1000.

---

4.




---

# PROBLEM 3

1.

It is to be read by an infrared scanner, using the EAN system, making sure the veracity of the primary key is feasible. It also increments new articles automatically.

---

2.

That is because it is a foreign key to the primary key in the Products table. It is using the primary key of the Products table as an attribute to count how many, for example "L T-shirt black, cotton" (with its absolute product_id), instead of "black t-shirts" to keep track of stock in a meaningful way for the employees, so they can do data cube analysis and do sales like "3 for 2 on Dressmann Premium Cotton T-shirts" when it is most fruitful. For this to work we need to know the EAN/GTIN/primary key to analyze in a meaningful way since "black t-shirt" can mean many things. Also, within a database consisting of two tables there are no need to denormalize anything.

---

3.

This is a procedure that adds a product with three attributes (name, price, stock) to the Products-table and raises a notice that says "Product added with ID: {number}". It takes name, price, stock as arguments and returns a product_id into new_product_id. Then it raises the notice and tells the user that the new product is added and showing the new product_id by using substitution in a "raise notice"-function.

---

4.

See attachment or this:

``` PostgreSQL
CREATE OR REPLACE PROCEDURE record_sale(
    p_product_id INT, p_quantity INT)
LANGUAGE plpgsql
AS $$
DECLARE
    new_stock_level INT;
    new_product_ID INT;
    new_stock INT;
    new_quantity

BEGIN

    RETURNING stock into new_stock
    SELECT stock FROM Products WHERE p_product_id = new_product_ID
    IF STOCK <= MIN_VALUE RAISE NOTICE 'Insufficient stock for product'


    new_stock = (stock-new_product_id(COUNT(*));
    UPDATE Products.new_stock = new_stock;


    total_sale_amount := new_product_ID(count(*)) + {count of each product_ID}

    INSERT INTO Sales(productID, quantity, total_sale_amount);

END;
$$
```

---

5.

See attachment or this:

```` PostgreSQL
CREATE OR REPLACE PROCEDURE check_stock(
   p_product_id)
LANGUAGE plpgqsql
AS $$
DECLARE
   new_p_product_id INT, new_stock_ID INT;

BEGIN
   SELECT stock, p_product_id FROM Products;
   RETURNING stock into new_stock_ID
   RETURNING p_product_id INTO new_p_product_id

RAISE NOTICE'We have got:%',new_stock_ID' of %',p_product_id'.';

END;
$$;
```