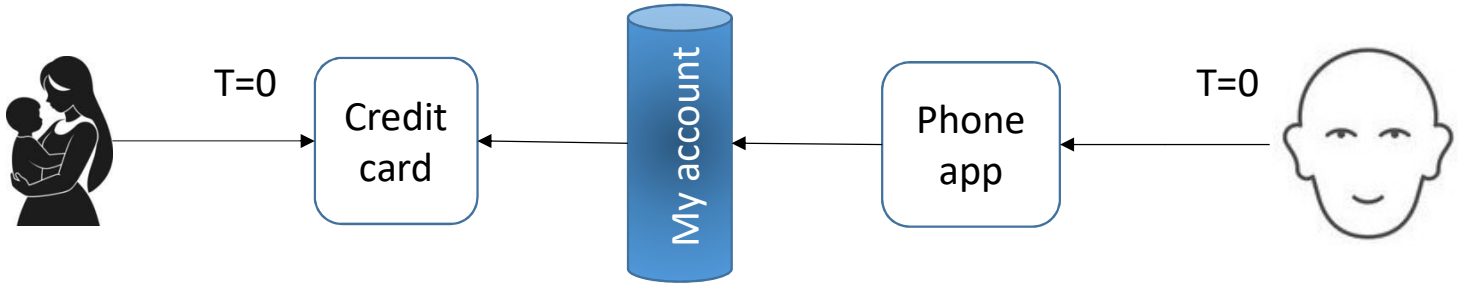
OPTIMISTIC LOCKING AND PESSIMISTIC LOCKING

```java
@Entity  soufiane mouhtaram *
@Getter
@Setter
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class AccountBalance {
    @Id
    private Integer id;
    private Integer balance;
    @Version
    private Long version;
}
```

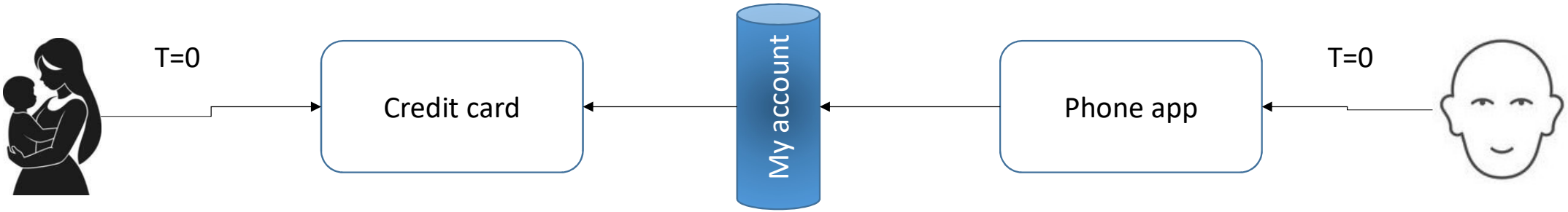
Soufiane MOUHTARAM

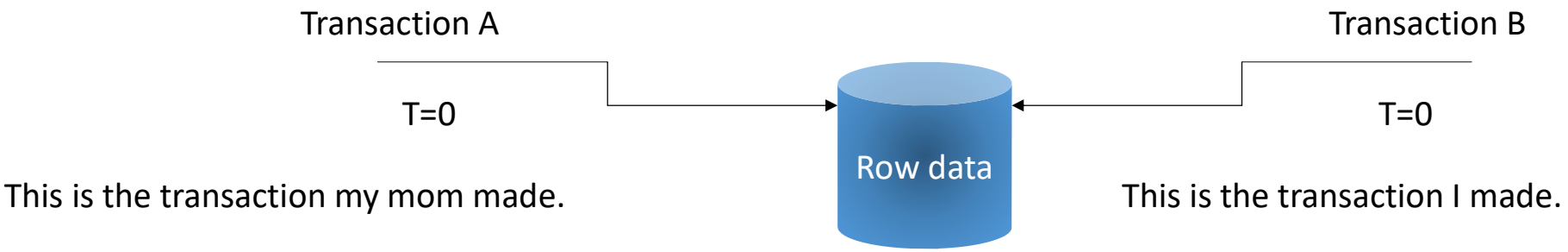


Example:

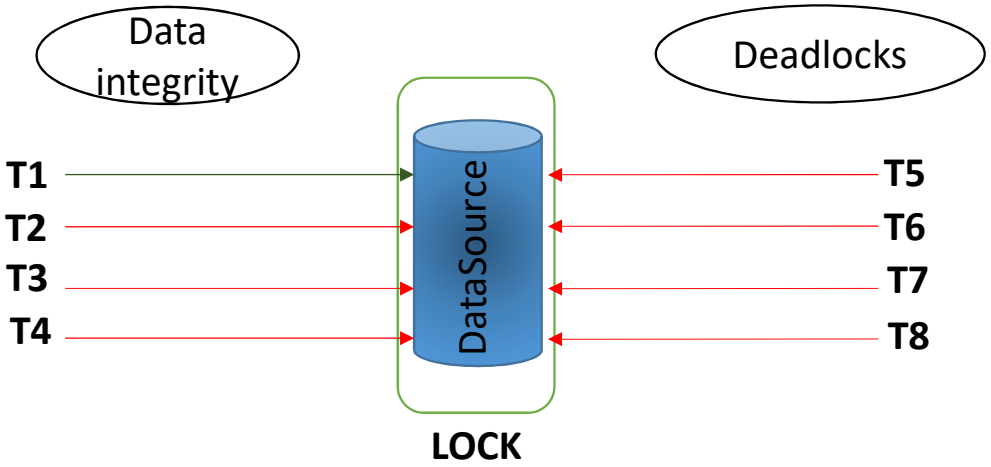
Let's start with a simple scenario to understand what **locking** means in a database system.
Imagine I have a **bank account** with a balance of **1300 Dhs**.
I can make payments either through the **mobile banking app** on my phone or using my **bank card**.
Now, I give my **bank card to my mother**, and she goes to **Marjane** to buy groceries worth **1000 Dhs**.
At the same time, I remember that I forgot to pay the **internet bill**, which costs **500 Dhs**, so I pay it through the app.
Transaction 1 – Mom: 1000 Dhs
Transaction 2 – Internet (Me): 500 Dhs
Account Balance: 1300 Dhs
What's going to happen?

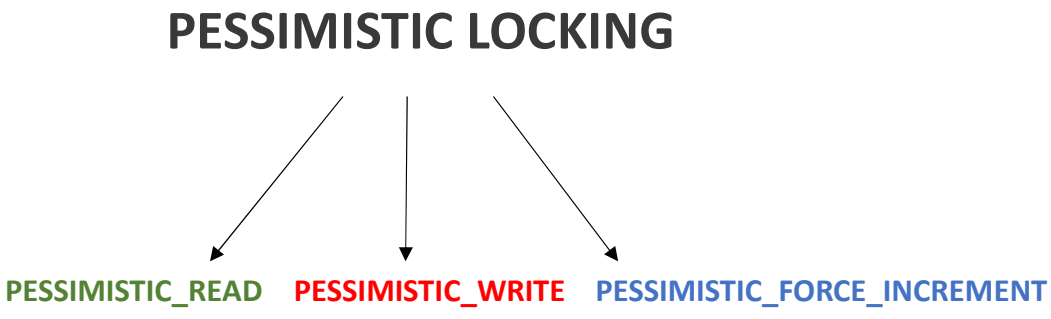
If both transactions are processed **at the same time**, and there's **no proper locking mechanism** in place, the system might allow both payments to go through—causing the balance to drop below zero or leading to **data inconsistency**.





If both transactions happen at the same time, we'll run into a **conflict**.
The solution to this problem is to use **locking**.
For example, when my mom is making her payment, the system should **lock the corresponding row** in the database.
That way, if I try to make a transaction at the same time, I won't be able to access or update that data — **because it's currently being used by another transaction** (my mom's).
There are two main types of locking in this context:
☞ **Optimistic Locking** and
☞ **Pessimistic Locking**.





🔒 What is Pessimistic Locking?

Pessimistic Locking is a locking strategy where a record is locked as soon as a transaction reads it, preventing other transactions from accessing or modifying it until the lock is released. It follows a blocking approach to ensure data consistency.

This approach is typically used in systems where conflicts are highly likely, especially in high-concurrency environments. When a transaction reads a row, it immediately locks it, and no other transaction can access or modify that row until the first transaction is completed.

Types of Pessimistic Locks:

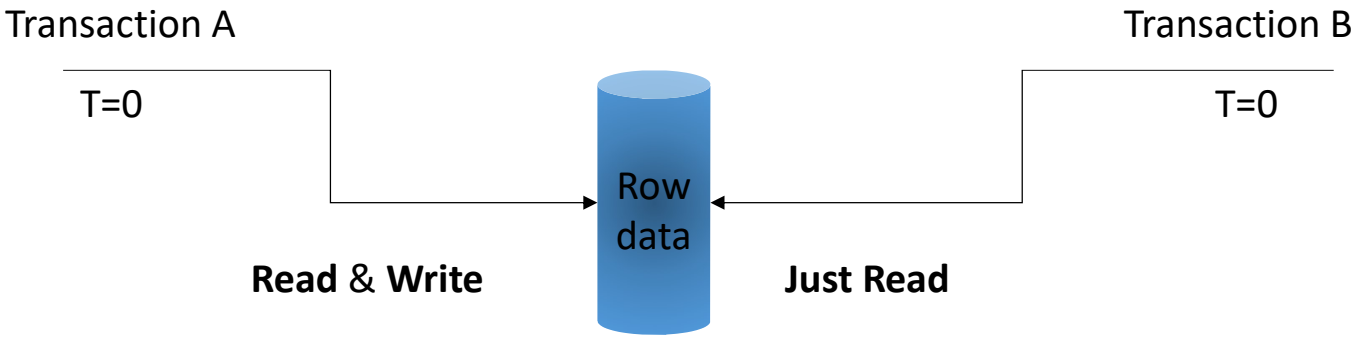
PESSIMISTIC_READ – Allows reading, blocks writing.

PESSIMISTIC_WRITE – Blocks both reading and writing.

PESSIMISTIC_FORCE_INCREMENT – Like PESSIMISTIC_WRITE, but also increments the version number even if no changes are made.

PESSIMISTIC_READ

In the case of PESSIMISTIC_READ, the locking strategy allows shared access — meaning other transactions are allowed to read the data, but no transaction can modify or update it while the lock is held. It's still a blocking strategy, but only for write operations.
This mode is useful when you're okay with concurrent reads, but you want to block any concurrent writes to avoid conflicts or data inconsistency.

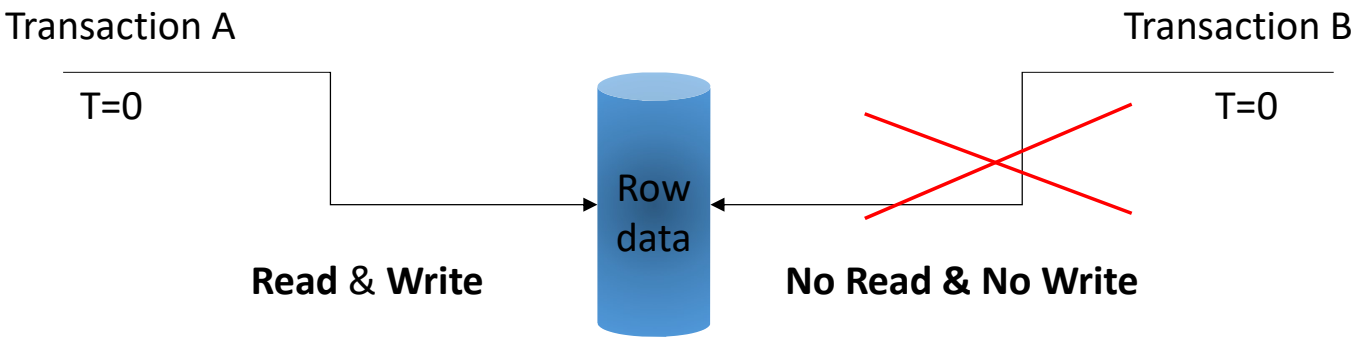


In the AccountBalanceRepository we have marked the findById with @Lock annotation.
To undersatnd pessimistic_read,
Transaction A : findById(1) **done**
Transaction B : findById(1) **done**
Transaction A : update(1,account) **done**
Transaction B : update(1,account) **not**

```
@Lock(value = LockModeType.PESSIMISTIC_READ)
Optional<AccountBalance>  findById(Integer id);
```

PESSIMISTIC_WRITE

- No other transactions can read or write the row until the lock is released.
- Use when you're going to modify the data and need to **block all other access** to avoid dirty reads or lost updates.

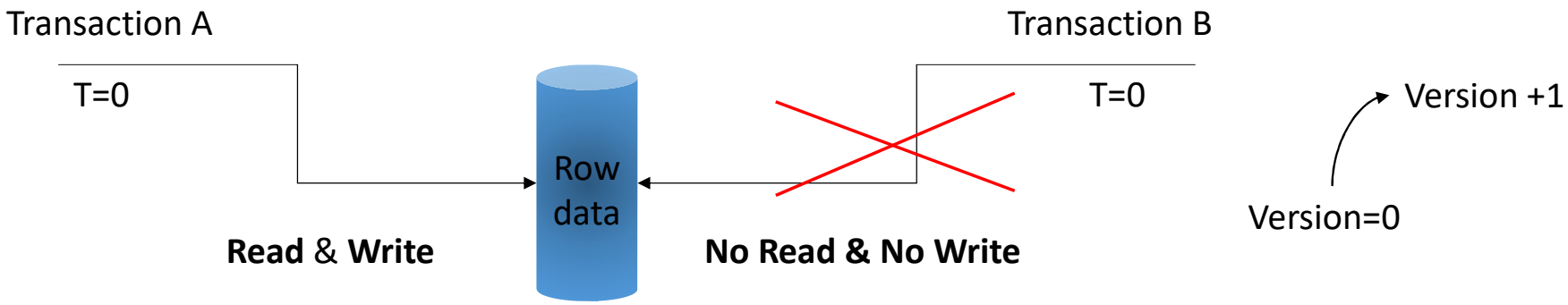


In the AccountBalanceRepository we have marked the findById with @Lock annotation.
To understand pessimistic_write,
Transaction A: findById(1) **done**
Transaction B : findById(1) **not**
Transaction A: update(1,account) **done**
Transaction B : update(1,account) **not**

```
@Lock(value = LockModeType.PESSIMISTIC_WRITE)
Optional<AccountBalance>  findById(Integer id);
```

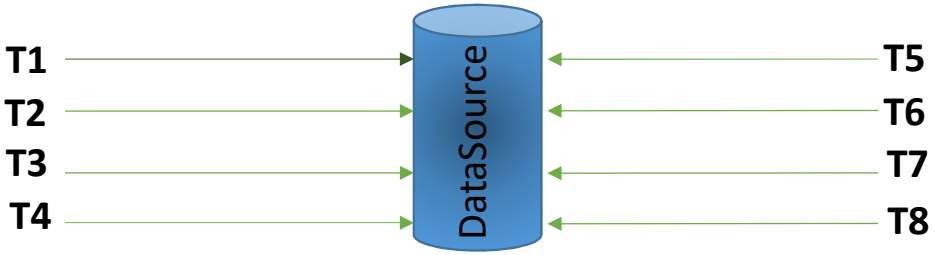
PESSIMISTIC_FORCE_INCREMENT

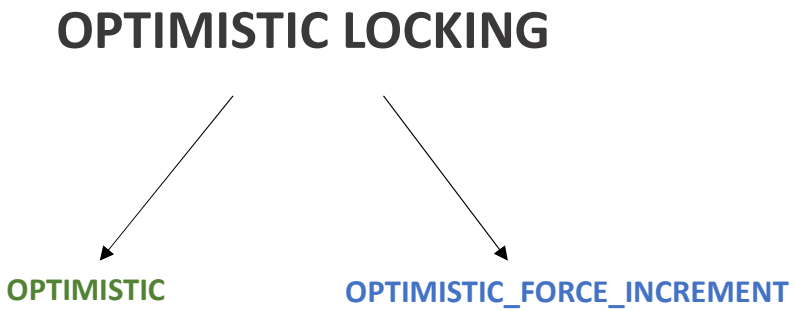
It's similar to PESSIMISTIC_WRITE, but when used with an entity that has a @Version field (we'll discuss this further in optimistic locking), it forcibly increments the version column, even if no fields in the entity have changed.



In the AccountBalanceRepository we have marked the findById with @Lock annotation.
To understand force_increment,
Transaction A: findById(1) **done**
Transaction B : findById(1) **not**
Transaction A: update(1,account) **done**
Transaction B : update(1,account) **not**

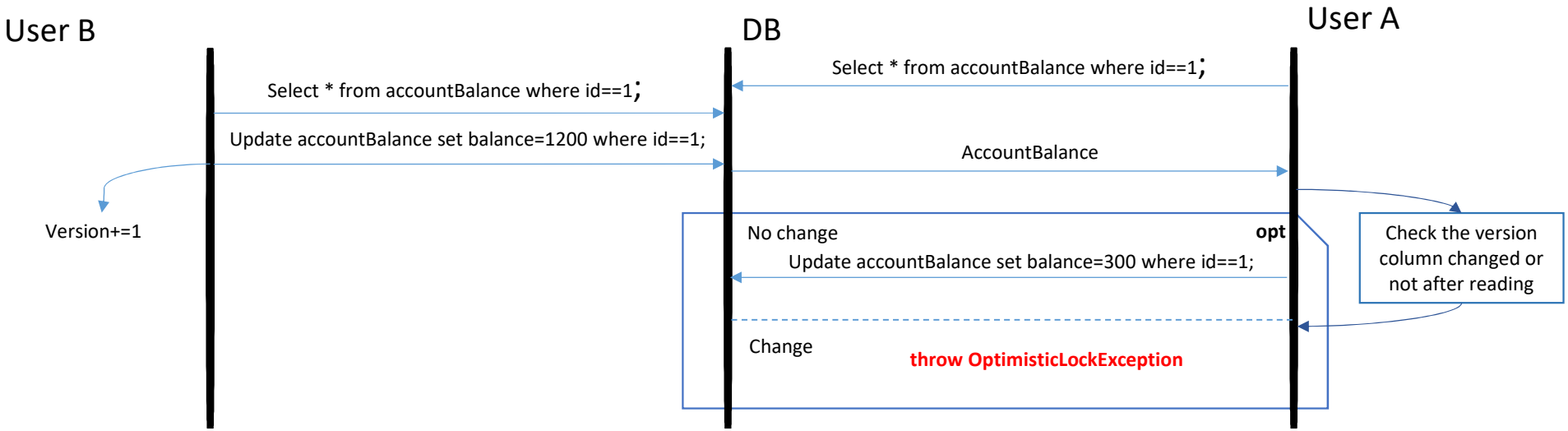
```
@Lock(value =
LockModeType.PESSIMISTIC_FORCE_INCREMENT)
Optional<AccountBalance>  findById(Integer id);
```



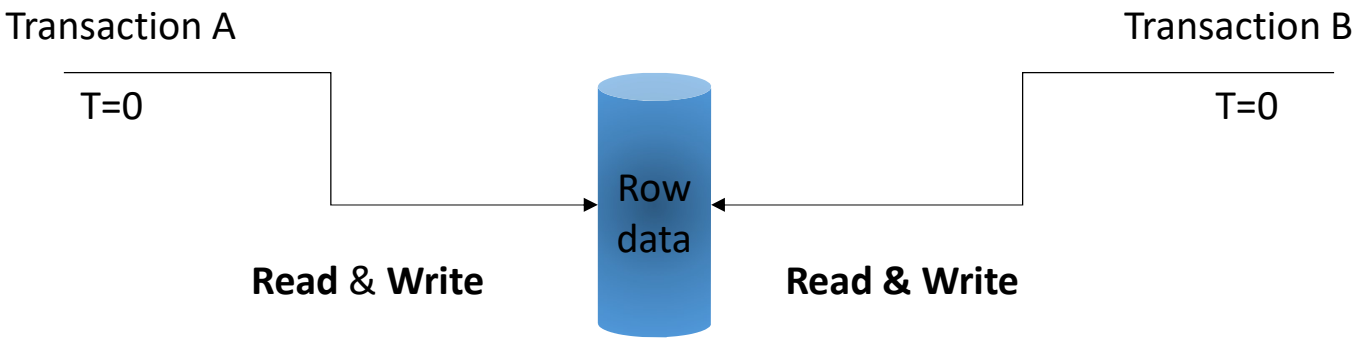


What is optimistic locking?

Optimistic Locking works on the assumption that conflicts are rare — like someone who sees life from a positive perspective.
It allows multiple transactions to read and work with the same data simultaneously, without immediately locking it.
But there's a catch:
Transaction A wants to update the accountBalance.
At the same time, Transaction B also wants to update the same accountBalance.



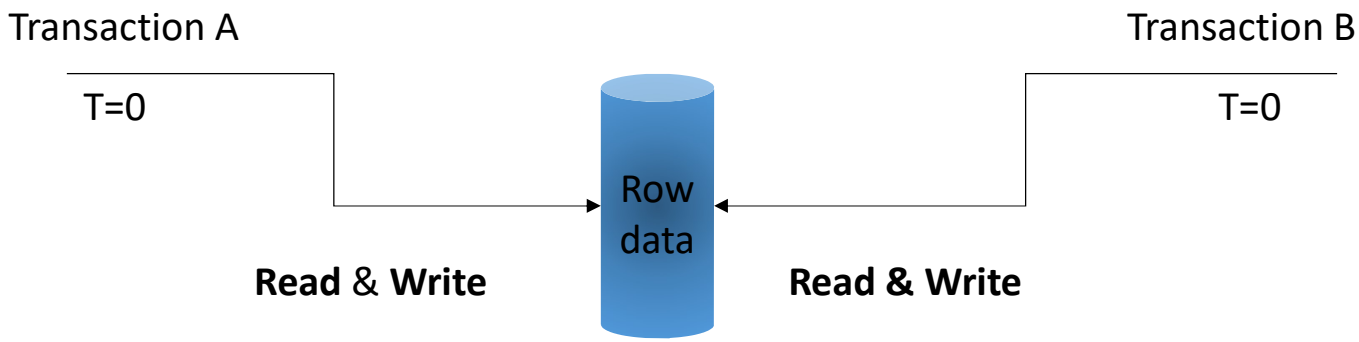
OPTIMISTIC



To understand the **OPTIMISTIC** lock mode, imagine this scenario:
Transaction A: findById(1) **done**
Transaction B : findById(1) **done**
Transaction A: update(1,account) **done (if version column has not changed)**
Transaction B : update(1,account) **done (if the version column has not changed)**

```
@Lock(value = LockModeType.OPTIMISTIC)
Optional<AccountBalance>  findById(Integer id);
```

OPTIMISTIC_FORCE_INCREMENT



To understand the **OPTIMISTIC_FORCE_INCREMENT** lock mode, imagine this scenario:
Transaction A performs both read and write operations on a row.
Meanwhile, Transaction B only performs a read operation on the same row.
After Transaction A reads the row, Transaction B also reads that same row.
If the lock mode is OPTIMISTIC, the version number will only be incremented if a write happens.
However, if the lock mode is OPTIMISTIC_FORCE_INCREMENT, the version number is forcibly incremented, even if no actual changes are made to the data.

```
@Lock(value =
LockModeType.OPTIMISTIC_FORCE_INCREMENT)
Optional<AccountBalance>  findById(Integer id);
```