From the vault of my engineering newsletter
"Systems That Scale"

Saurav Prateek's

Transactions in Distributed Systems

Understanding how Transactions enable **Concurrency Control**, **Fault Tolerance** and **Data Consistency**

Table of Contents

Transactions - Strong Isolation Levels

Transactions 4
Atomicity 4
Consistency 5
Isolation 6
Durability 8
Is re-trying failed Transactions safe? 8
Read Committed 9
No Dirty Reads 9
No Dirty Writes 11
Implementing Read Committed Scheme 13

Distributed Transactions and Concurrency Control

Introduction 15
Serializability 16
Concurrency Control 17
Why holding the lock for the entire Transaction execution 17
Case 1: Transactions reading a modified data item of a partially complete Transaction 18
Case 2: Partially executed transaction got aborted due to some unfortunate reasons 20

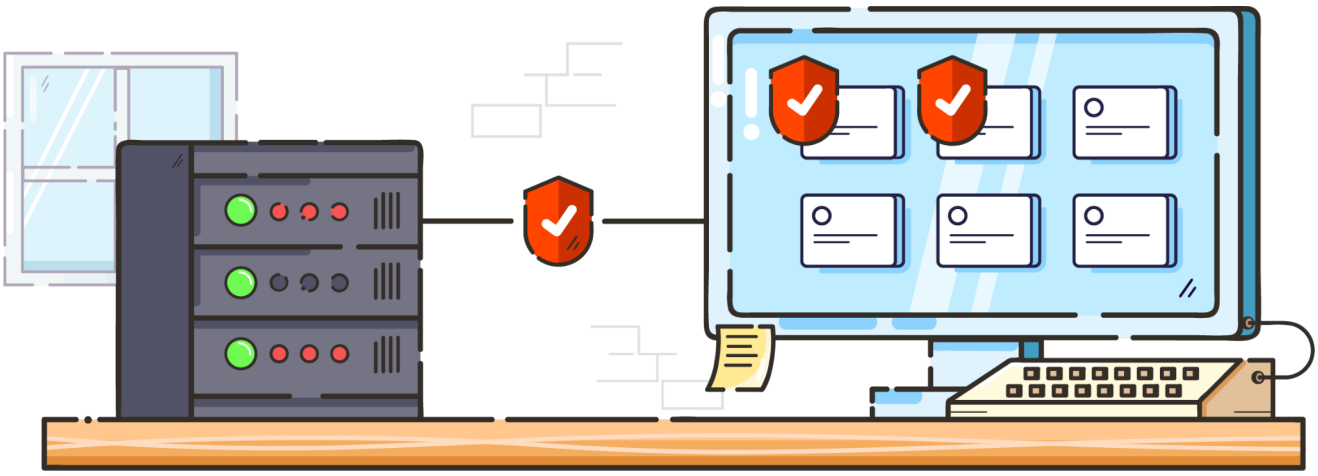
Transactions - Weak Isolation Levels

Introduction 23
Non Repeatable Read 23
Snapshot Isolation Scheme 26
Multi-Version Database 26
Implementing Snapshot Isolation scheme 26

Chapter 1

Transactions - Strong Isolation Levels

The chapter discusses the concept of **Transaction** in detail. It also discusses the **Read Committed** scheme and how it guarantees no dirty reads and writes in the system. It also discusses the **Strong Isolation Levels** implemented in the system.



Transactions

Transactions is defined as the group of several **Read** and **Write** operations which either succeeds altogether or fails entirely. The concept of transaction is to club all the reads and writes into a single logical unit and execute them at an atomic level.

Transactions avoid multiple concurrency issues from taking place in a database. It ensures that none of the reads or writes are committed in the case of partial failure. It allows the system to perform safe retries. The implementation of Transaction in a system can avoid:

- Multiple clients writing the same data-item at the same time and over-writing each other's changes.
- A client reading a data-item which has been partially written by another client.
- Multiple race-conditions between the client causing surprising behaviours.

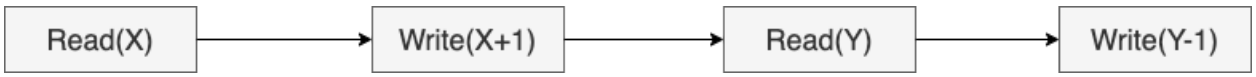
But not every time a system needs transactions. Sometimes these transactions may cause performance over-heads and higher response time for a system. Many systems avoid transactions and implement Weak Isolation Levels in order to enhance their performance and reduce their response time.

Transactions avoid concurrency issues in a database by safety guarantees which are described by the well known **ACID** (**A**tomicity, **C**onsistency, **I**solation and **D**urability) properties. The systems which do not involve transactions or guarantee the **ACID** properties are known as **BASE** meaning **B**asically **A**vailable **S**oft state and **E**ventual consistency.

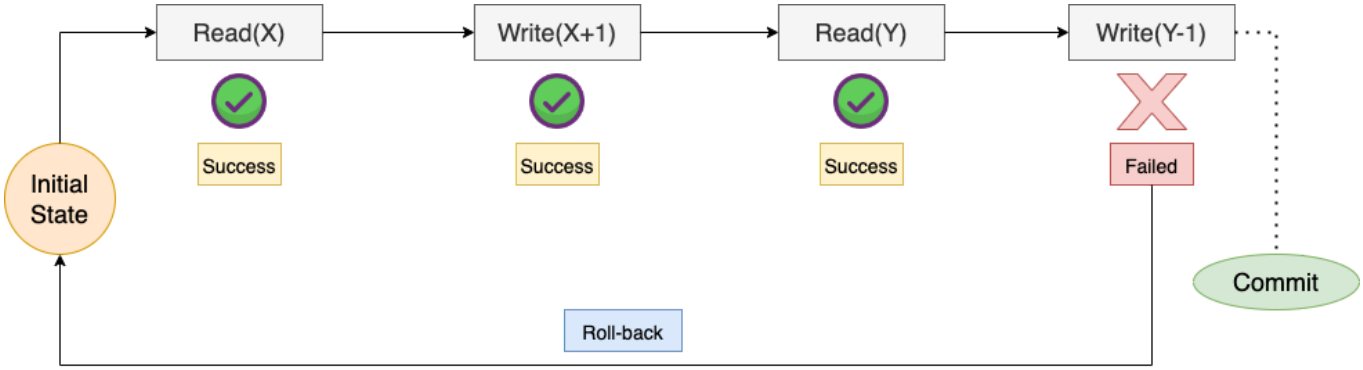
Let's understand these guarantees provided by transaction in detail.

Atomicity

Atomicity refers to an operation or entity that can not be broken further into smaller ones. Suppose a transaction **T** performs a series of operations sequentially.



If during the execution of the transaction **T**, any failure takes place. Suppose a network failure, memory failure or something like that, then all operations executed as a part of the transaction before failure must be rolled-back.



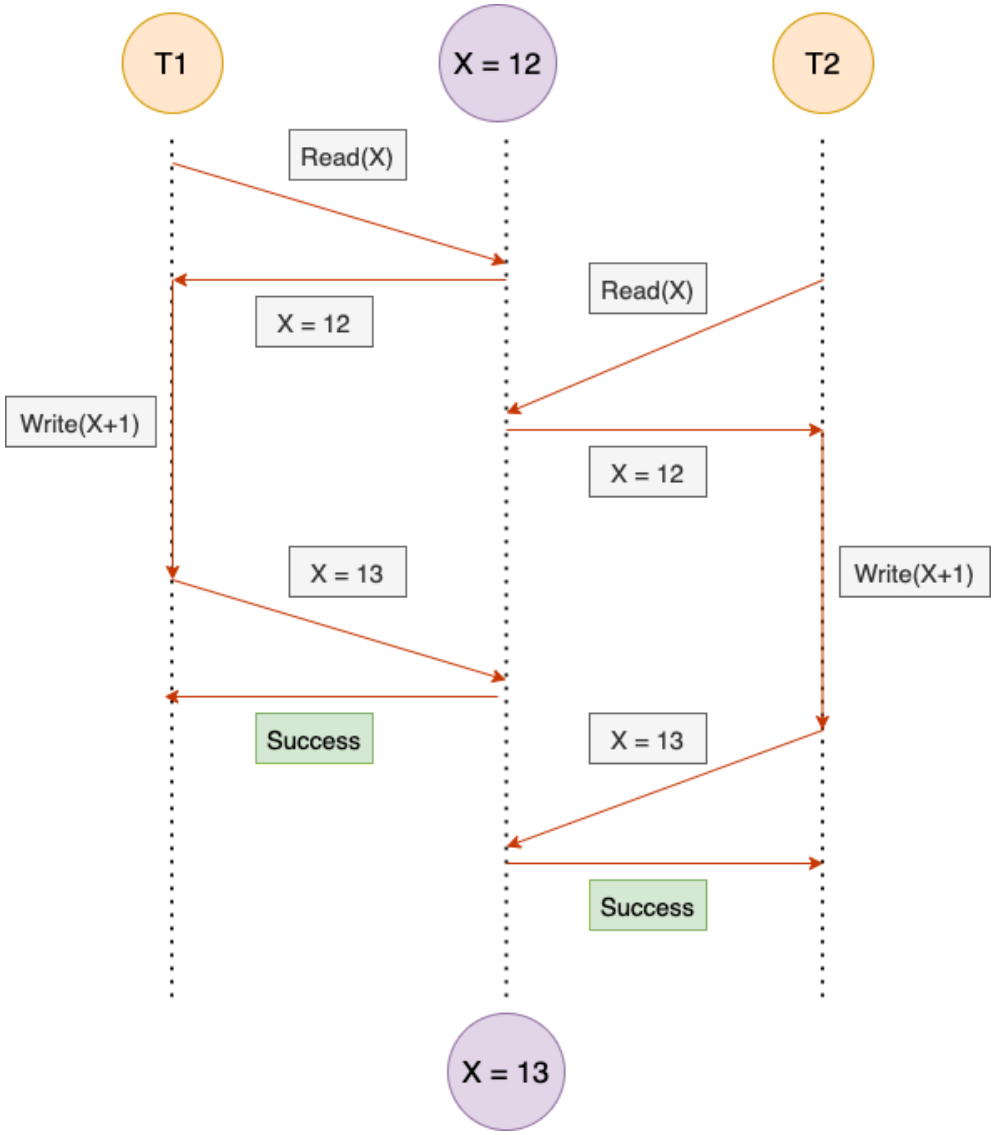
This allows a system to retry the transaction **T** once again safely. Without atomicity, if transaction **T** would have failed at the last operation [**Write(Y-1)**], then the system might not have rolled-back and the previous write operation [**Write(X+1)**] would have persisted. Then re-trying the transaction **T** would result in the commitment of the previous write operation [**Write(X+1)**] twice and hence incrementing the value of data-item **X** twice, incorrectly.

Consistency

It states that a certain statement about the data must always be true even after or before a transaction completes. For Example: In an Accounting System, the credits and debits must balance.

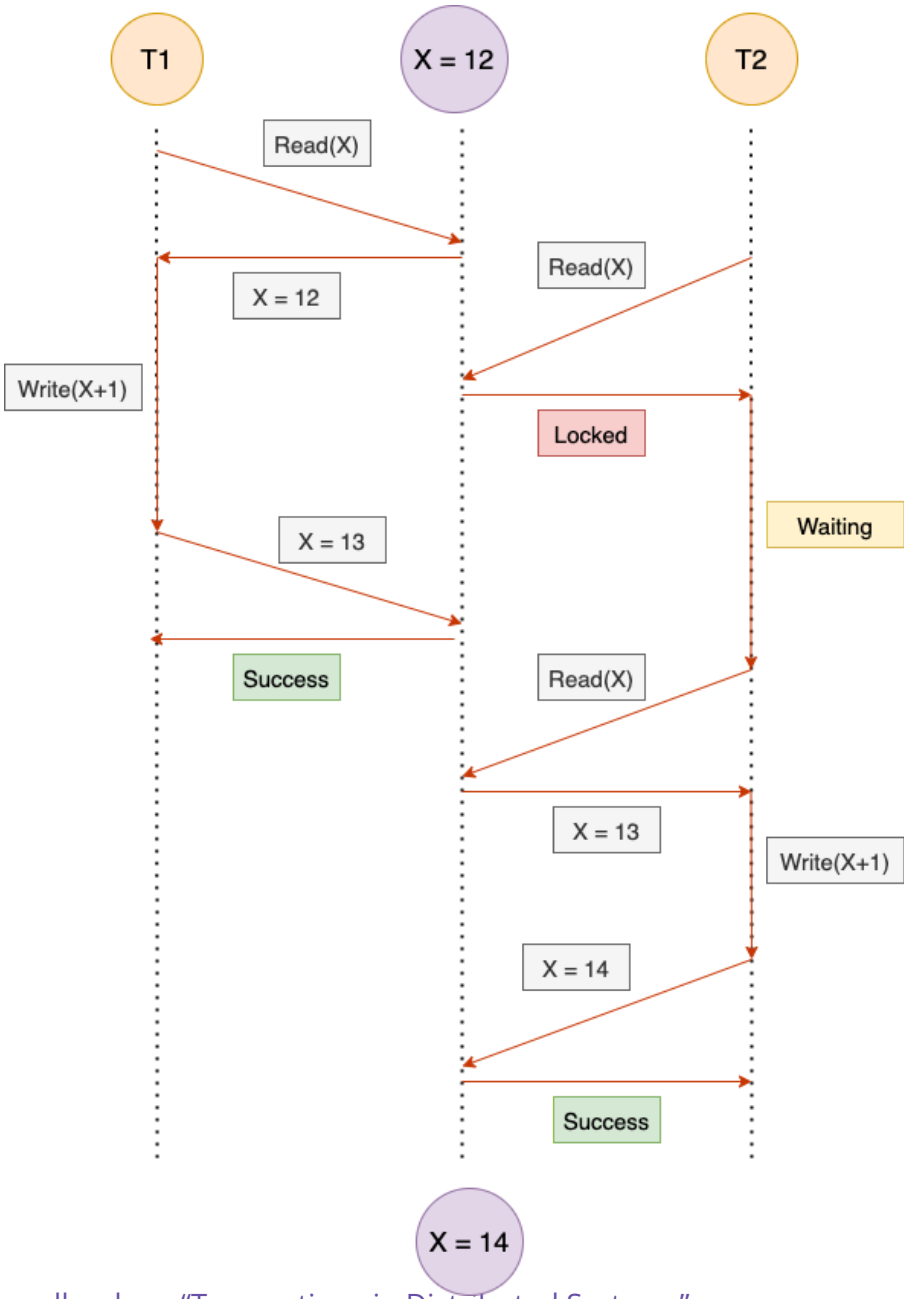
Isolation

It ensures that the two transactions concurrently executing will always be isolated from each other.
Suppose there are two transactions **T1** and **T2** which are reading the same data-item **X** and increments **1** to it. In that case, if isolation is not guaranteed then following race conditions might occur.



In the above scenario, transaction **T1** reads the data-item **X** (**12**) and increments its value to **13**. During the process of writing (by **T1**) the transaction **T2** also reads the value of data-item **X** which was **12** at that time. Then **T1** writes the incremented value (**13**) back to the data-item **X** and after that **T2** also writes the incremented value (**13**) back to data-item **X**. The final value of **X** was **13** which was incorrect.

Ideally while **T1** was executing, **T2** must have waited for data-item **X**. Once **T1** updated the data-item **X** to **13** then only **T2** should have read data-item **X** and incremented it to **14**.



Durability

It ensures that once a transaction commits then the results are permanent to the database.

Any write operation committed successfully must persist in the database. Although there is no one specified technique which guarantees **100%** durability. But we can combine multiple techniques like writing to disk, replicating to remote machines and backups in-order to ensure high durability.

Is re-trying failed Transactions safe?

We read earlier that Transaction allows the system to perform safe-retries. But, is it always safe to retry a failed Transaction? Let's discuss.

1. What if the Transaction succeeded and got committed but the network call failed while the server was trying to acknowledge the successful commit. Retrying the transaction in this case would cause the transaction to be performed twice.

2. What if the Transaction failed due to increasing load over the system. Retrying the failed transaction in this case will make the situation even worse by further adding more load over the system.

3. What if the Transaction has some side effects which could happen outside of the database and couldn't be rolled back on failure. Suppose a transaction sends an email to a group of users in between its execution. In this case future failure and re-trying of the transaction might cause the same email to be sent twice to the same group of users.

Let's discuss a scheme that ensures Transaction Isolation.

Read Committed

This scheme generally make two guarantees:

1. No Dirty Reads
2. No Dirty Writes

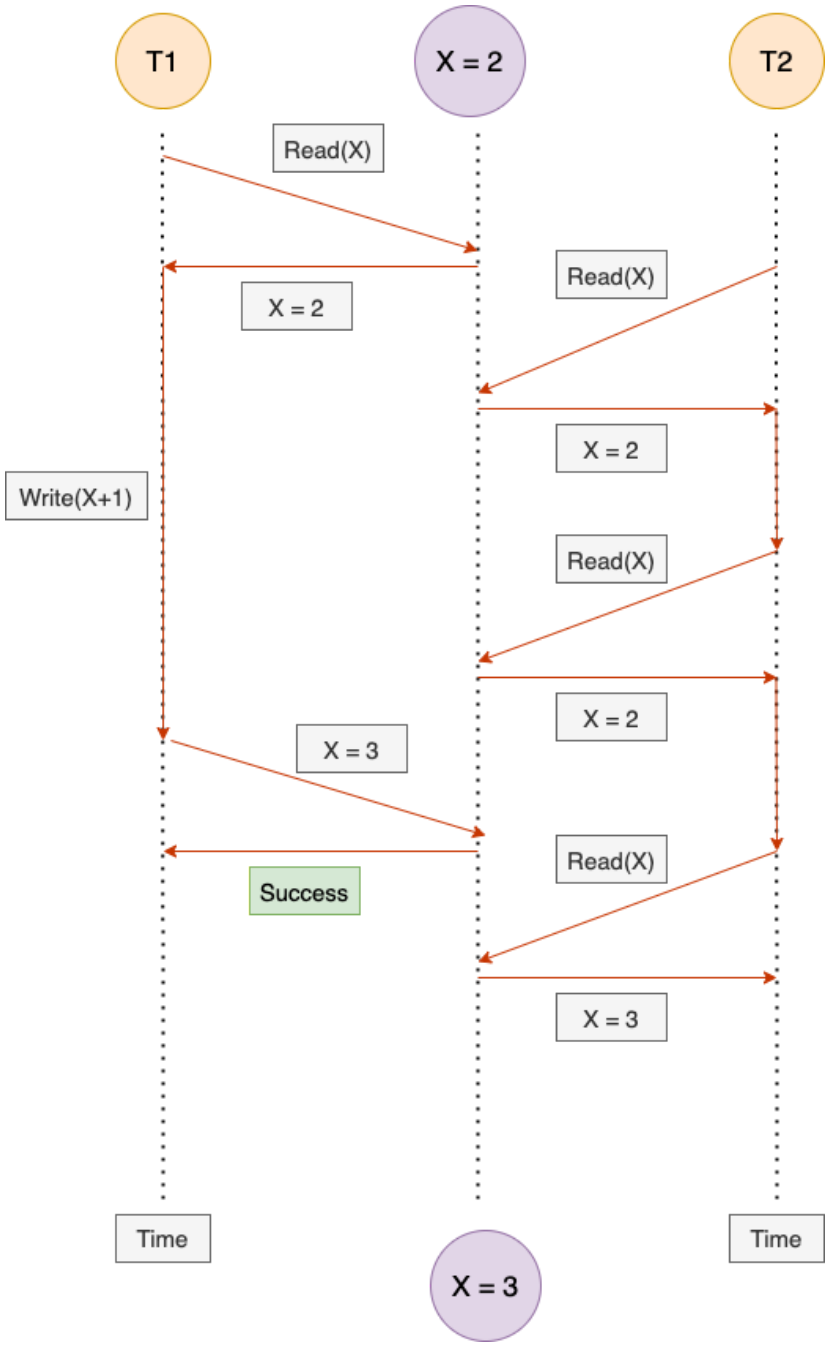
Let's understand the above mentioned terms.

No Dirty Reads

A transaction being able to see the data written by another uncommitted transaction is known as Dirty Read.

In this scheme if a transaction tries to read a data-item, then the system automatically returns the old previously committed data. The newly written version of the data-item is only made available when the transaction writing it is actually committed.

Suppose a data-item X having value 2 is being incremented by the transaction T1. In the meantime if another transaction T2 tries to read the data-item X multiple times, then the system will return the past committed value 2 until the transaction T1 is committed.


T1
X = 2
T2
Read(X)
X = 2
Read(X)
X = 2
Write(X+1)
Read(X)
X = 2
X = 3
Read(X)
Success
X = 3
Time
X = 3
Time

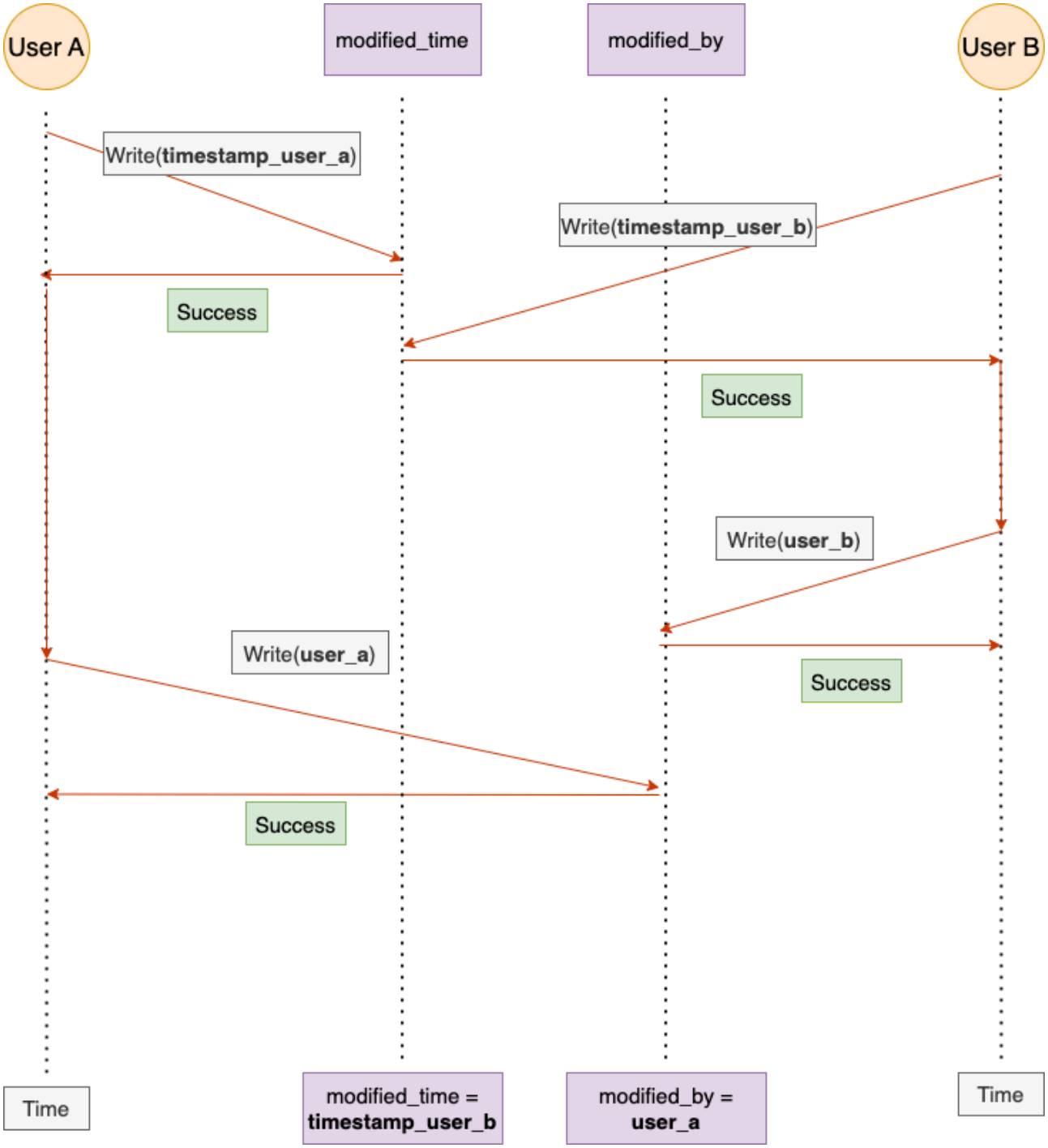
No Dirty Writes

When a later transaction over-writes a data-item written by an earlier transaction which is yet un-committed is known as **Dirty Write**.

The **Read Committed** scheme makes sure that there are no dirty writes involved in a system.

Dirty Writes can leave a database in an in-consistent state. For Example: There's a file system which is updated by multiple users. Suppose **User A** updates the file system and as a result updates the modified_time with the current timestamp (**timestamp_user_a**). In the meantime another **User B** comes up and updates the file system and as a result updates the modified_time with the current timestamp (**timestamp_user_b**). **User B** then also updates the modified_by entry with their user-name (**User_B**). Later **User A** updates the modifed_by entry with their user-name (**User_A**).

This is an example of Dirty Write where **User B** updates the same file system's **modified_time** and **modified_by** entries while **User A** is in the process of updating them.



Here, the data is inconsistent since after the two transactions are completed, the data shows that the file system was updated by **User A** at **timestamp_user_b** which was actually the time when **User B** updated the file system.

Implementing Read Committed Scheme

The Read Committed scheme can be implemented in the following way such that it guarantees no dirty reads and writes.

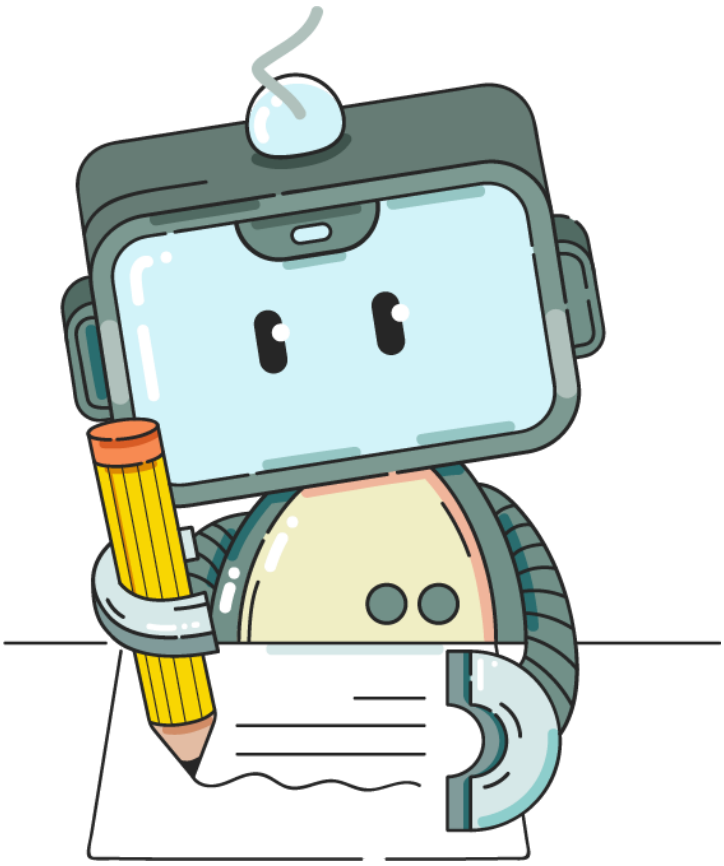
- The scheme ensures no dirty writes by implementing a locking mechanism. Any transaction planning to update a data-item must acquire a lock on it before updating. While locked no other transaction could further modify that data-item. Once the update is done, the transaction can further release the lock to make it available for other transactions.

- The scheme ensures no dirty reads by the similar process. The transaction reading a locked data-item will get the old value of the data-item. The updated value of the data-item will only be made available to other transactions once the lock is released by the transaction updating it.

Chapter 2

Distributed Transactions and Concurrency Control

The chapter discusses distributed transactions and how we can ensure concurrency control over the parallel execution of these transactions.



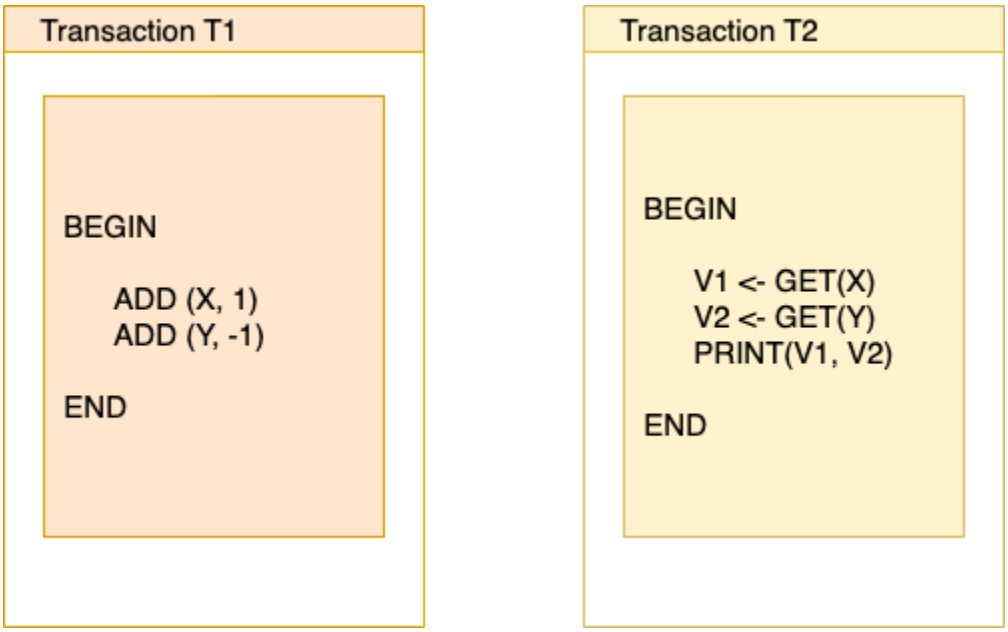
Introduction

For systems having a huge amount of data, they can generally split or shard their data over multiple database servers. Let's take an example of a Bank server where the account of customers can be distributed over a number of database servers. Suppose we are transferring an amount of money from the account of **Customer A** to another **Customer B** then there can be a chance that they might be stored over different servers. In general this complexity is hidden from the Application logic.

Suppose we are running a Bank and we want to transfer an amount of money from the account of user **X** to the account of user **Y**. Initially the balance of both the accounts was **10** units. These accounts are simple records over the database.

```
X = 10
Y = 10
```

And suppose there are two Transactions: **T1** and **T2** which will be running in parallel at the same time. First Transaction **T1** will transfer **1** unit from account **Y** to account **X**. The second Transaction **T2** will perform an audit to ensure that the sum of balance over all the accounts doesn't change. Since we are moving the amount within the back account, hence the total amount shouldn't change.



The first Transaction **T1** moves **1** unit from account **X** to account **Y**. Hence in the first step of the Transaction, unit **1** is added to account **X** and after that unit -**1** is added to account **Y** (since we are moving **1** unit from account **Y**).

The second Transaction **T2** gets the account balance of **X** (**Get**(**X**)) and then gets the account balance of **Y** (**Get**(**Y**)) and further prints both the fetched account balances.

Serializability

By definition the results are **serializable** if there exists a serial order of the execution of the transactions that yields the same result as the actual executions.

Let's take the previous two Transactions **T1** and **T2**. There can be two ways in which these transactions can produce a serializable end result.

1. Transaction **T1** completes its execution and after that **T2** runs and further completes its execution.

```
Initial Configuration [X = 10, Y = 10]
     T1 -> Executes -> [X = 11, Y = 9]
     T2 -> Executes -> [prints -> X = 11, Y = 9]
```

2. Transaction **T2** completes its execution and after that **T1** runs and further completes its execution.

```
 Initial Configuration [X = 10, Y = 10]
     T2 -> Executes -> [prints -> X = 10, Y = 10]
     T1 -> Executes -> [X = 11, Y = 9]
```

The above two are the only legal ways of the execution of the transactions **T1** and **T2**.

Concurrency Control

In general while dealing with distributed transactions there is a need for concurrency control. In database transactions there are **Read** and **Write** operations which are required to be managed in the concurrent execution of the transactions. If we won't control the concurrent execution then it might lead to the existence of inconsistent data.

We will follow these two rules while executing a transaction:

1. Acquire locks before using any data item.
2. Hold on to the locks until the transaction is completely executed.

For the previous example of Transactions **T1** and **T2** we will discuss a way in which we will perform concurrency control and will also look upon the scenarios where things can go wrong if we won't take care of it.

Let's have a look at both the transactions once again. When **T1** starts its execution, it will first acquire a lock on data item **X** to add **1** unit into that account. Similarly when **T2** starts its execution it will also acquire a lock on data item **X** to fetch its account balance. Now when both the transactions execute in-parallel then both of them will race to get the lock over the data item **X**. Whoever gets the lock over that item first will go ahead and execute itself completely. The other transaction that failed to get the lock will wait for the other Transaction to finish and only then it will start its execution.

Why holding the lock for the entire Transaction execution

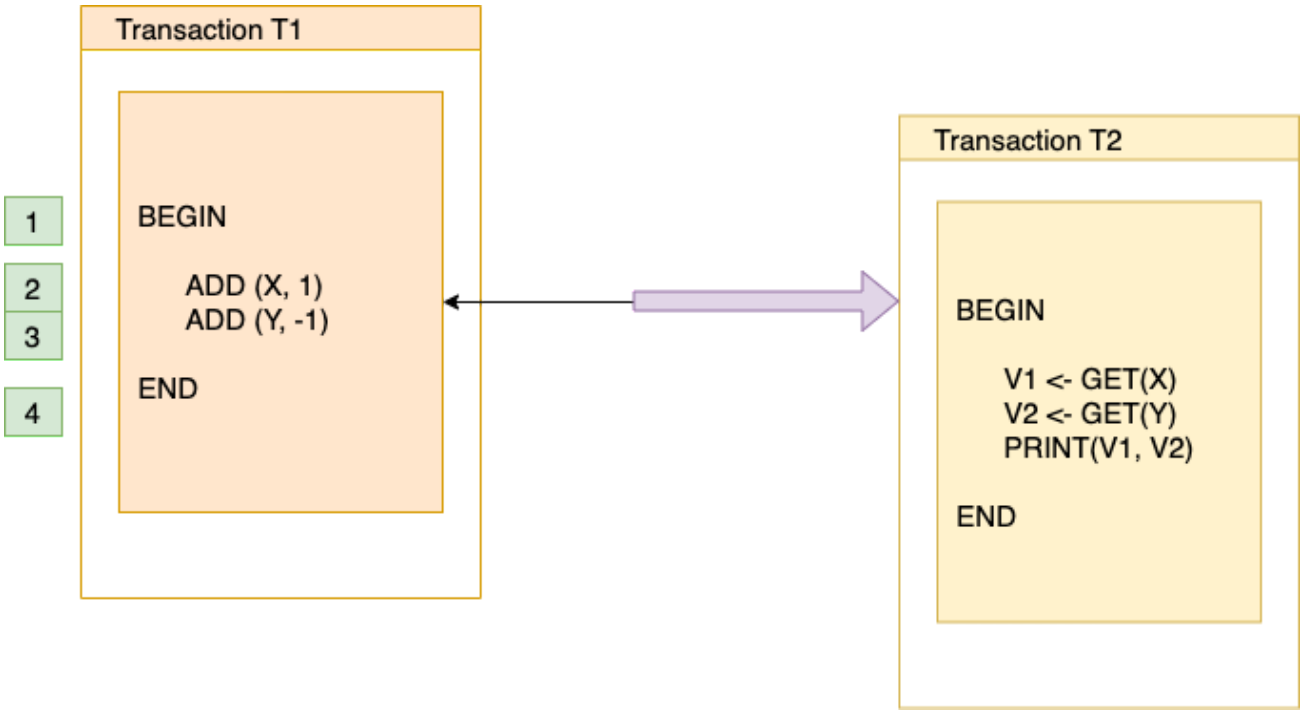
Here, a question might arise: Why does the transaction need to hold the lock over the data item till its entire execution? In our previous example, suppose transaction **T1** got the lock over the data-item **X** first, then it will hold the lock until it finishes its execution while transaction **T2** waits for that entire time. Can't it hold the lock just while it is dealing with the data-item **X** and release the lock the moment it finishes dealing with the data-item **X**? This will allow the Transaction **T2** to wait for a lesser

amount of time since **T1** will give up the lock over **X** early. Won't this lead to a better performance?

We can explain why our system doesn't do that with the help of two Cases.

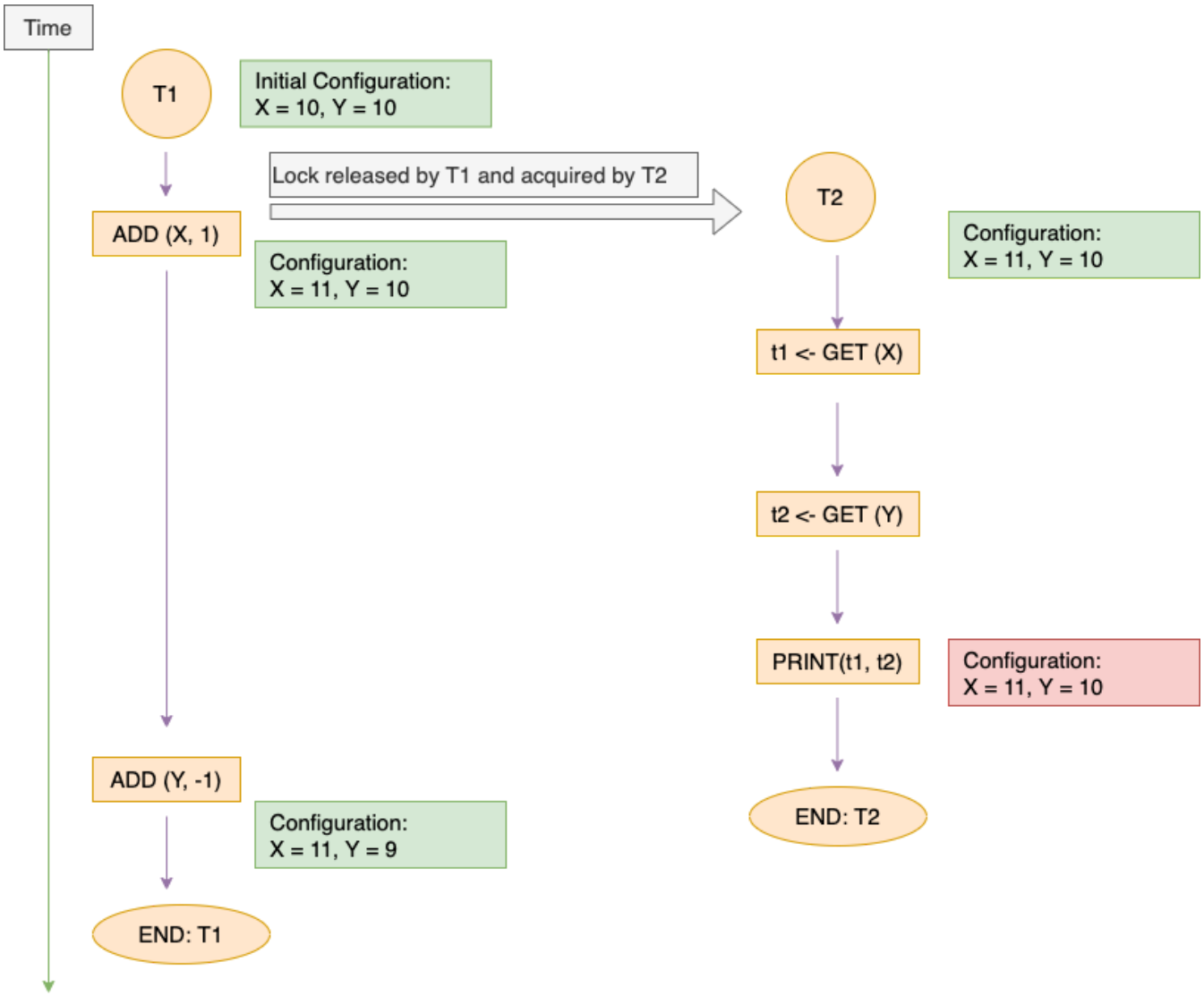
Case 1: Transactions reading a modified data item of a partially complete Transaction

Suppose a transaction acquired the lock on the data-item only till the time they were dealing with that item. In that case if **T1** got the lock over item **X** first, then it will also release the lock after dealing with it.



This will allow transaction **T2** to start its execution at the moment **T1** finishes executing the line **2**. The transaction **T2** can then read the modified value of data-item **X** by the partially executed transaction **T1**. And if **T2** went ahead of **T1**

and executed itself completely before **T1** executed line **3**, then this will result in an illegal end configuration.
The entire scenario will look somewhat like this.



Here when transaction **T2** was performing the audit it printed the account balances of **X** and **Y** as **11** and **10**. This is illegal since initially both the accounts had a balance of **10** units each and the sum should be **20** units which should be constant over the course of time. But here after the audit their sum comes up to be **21** units which is not correct.

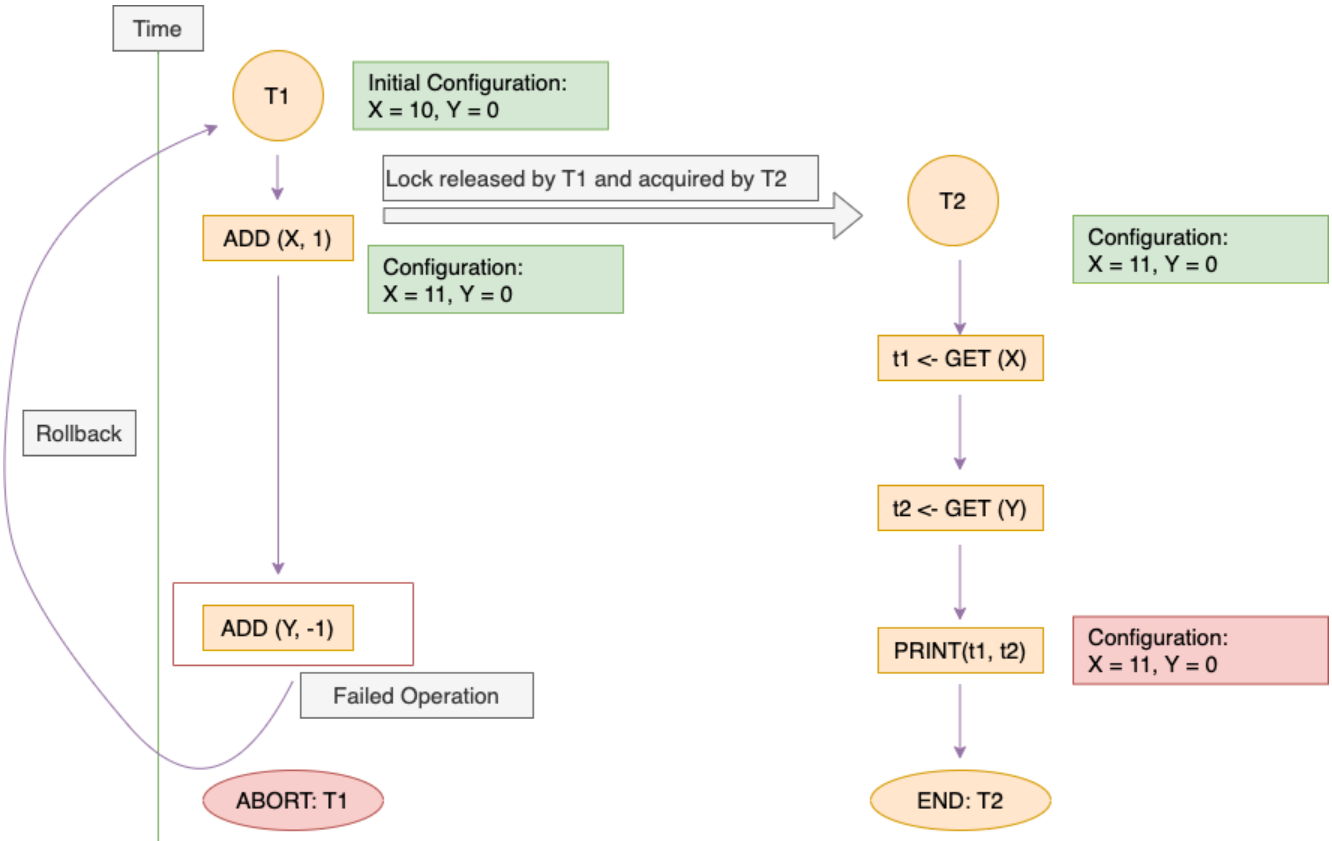
Case 2: Partially executed transaction got aborted due to some unfortunate reasons

Likewise in the previous case, suppose transaction **T1** got the lock over the data-item **X** first. It updated the data item **X** and released the lock immediately. And at that moment transaction **T2** acquired the lock over **X** and started its execution.

But transaction **T1** went ahead and failed due to some reasons. The possible reasons can be that the data item **Y** didn't exist or the data item **Y** had a value **0** and hence it can't further transfer **1** unit from itself to account **X**. In these cases transaction **T1** has no choice but to abort and rollback the changes. Hence transaction **T1** will rollback the value of data-item **X** back to **10**.

But in the meantime transaction **T2** will see the ghost value of **X** i.e. **11** and may print the illegal result.

This time the scenario will look like this.



Again when transaction **T2** was performing the audit it printed the account of balances of **X** and **Y** as **11** and **0**. This is illegal since initially both the accounts had a balance of **10** units and **0** units each and the sum should be **10** units which should be constant over the course of time. But here after the audit their sum comes up to be **11** units which is again not correct.

So, in both the previous cases the transaction **T2** printed an illegal state of the data-items. These are the obvious reasons due to which the transactions hold the lock over the data-item until their entire execution is completed.

Any of these situations wouldn't arise if the transaction **T1** would have held the lock over the data item until its complete execution.

Chapter 3

Transactions - Weak Isolation Levels

This chapter discusses the **Snapshot Isolation** scheme: one of the **Weak Isolation** schemes in detail and the problem of **Non-Repeatable** reads which this scheme solves for. We also discussed the concept of **Multi-version** databases in detail.

Introduction

In our previous chapter we discussed **Strong Isolation Levels** in detail. We understood the performance overheads involved and also discussed the **Read Committed** scheme in detail.

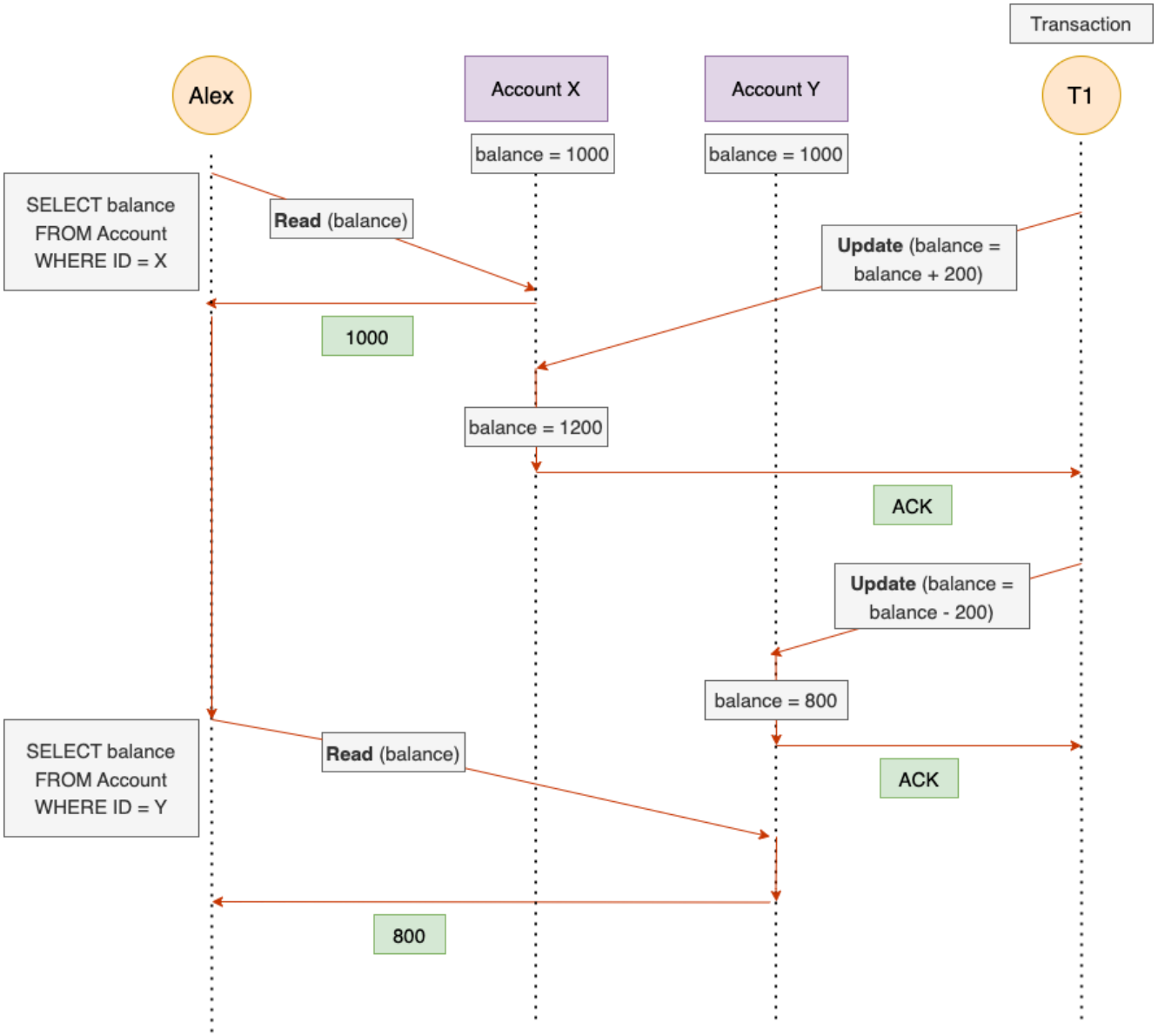
In this edition we will discuss an interesting **Weak Isolation** scheme called **Snapshot Isolation** and what are the problems they solve for.

Non Repeatable Read

Non Repeatable Read is yet another problem which could leave our Database in an in-consistent state. This problem can't be caught under the **Read Committed scheme**. Let's understand this issue with an example.

Suppose a person **Alex** has his money split into two accounts **X** and **Y**. Initially, Alex has an amount of **Rs.2000** which is equally divided into two accounts **X** and **Y** having a balance of **Rs.1000** each.

Now, Alex decides to transfer **Rs.200** from Account **Y** to **X**. A Transaction **T1** is initiated to carry on the transfer procedure while at the same time Alex reads the balance from both the accounts.



The above diagram explains the entire procedure. Due to the bad timing Alex received an incorrect sum of money from both of his accounts (**X** & **Y**). This happened because he was inquiring about his account's balance at the same time when the transfer was happening from Account **Y** to **X**.

Alex read the balance from Account **X** before the amount was transferred from Account **Y** and hence read the balance amount of **Rs.1000**. Afterwards, Alex read

the balance from Amount **Y** post the transfer procedure completed and hence received the amount of **Rs.800**.

Summing up the balance from both the Accounts result in the total amount of **Rs. 1800**, whereas the total amount was supposed to be **Rs.2000**. Alex might be wondering where his **Rs.200** went? Although this is a temporary glitch and will get solved when Alex re-tries to fetch the balance from Account **X** and will receive an amount of **Rs.1200** this time.

This is the **Non Repeatable Read** problem which was explained in the previous scenario. It is also known as the **Read Skew** problem.

Note: The **Read Skew** problem can not be caught by the **Read Committed** scheme. Since the reads in this scenario happened when the data was in a committed state. While the **Read Committed** scheme avoids only those reads which are performed on a data-item in an in-consistent state.

In the previous example the problem faced by Alex was temporary but Read Skew can also lead to some permanent issues in the following scenario.

Backups: While taking a backup, we copy all the values from a database. During the backup process, the writes are allowed to be performed on the database. In the above scenario some older-versions of data-items might get backed-up while some newer versions of the data-item can get backed-up in combination. Suppose if we later restore the database from that backup, then our database might be left in an in-consistent state. The problem Alex had in his scenario will get preserved in the database when we decided to apply the back-up.

There is one solution to this problem which is called **Snapshot Isolation** - A weaker isolation scheme. Let's understand this in our next section.

Snapshot Isolation Scheme

Snapshot Isolation scheme works on the principle:

> "Every transaction reads from a consistent snapshot of the database."

Every transaction can only see the version of data that was committed at the time the transaction was initiated. Suppose after the start of a transaction any update was made to the data-item, then the transaction won't be able to see those. This avoids the situation of **Non-Repeatable Reads** from happening in the first place.

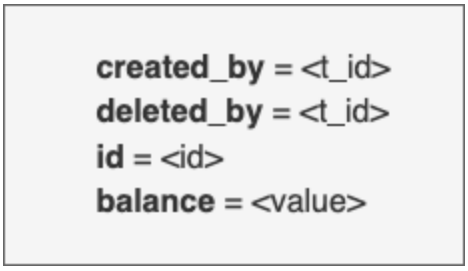
Multi-Version Database

In order to understand the Snapshot Isolation scheme, we need to first understand the concept of **Multi-version Databases**. Under this, a database maintains several different committed versions of the data-items. The reason is to support the Snapshot Isolation scheme, since multiple transactions might need to see the database at different points in time.

Since the database maintains multiple versions of data-items in parallel, this technique is known as **Multi-Version Concurrency Control (MVCC)**.

Implementing Snapshot Isolation scheme

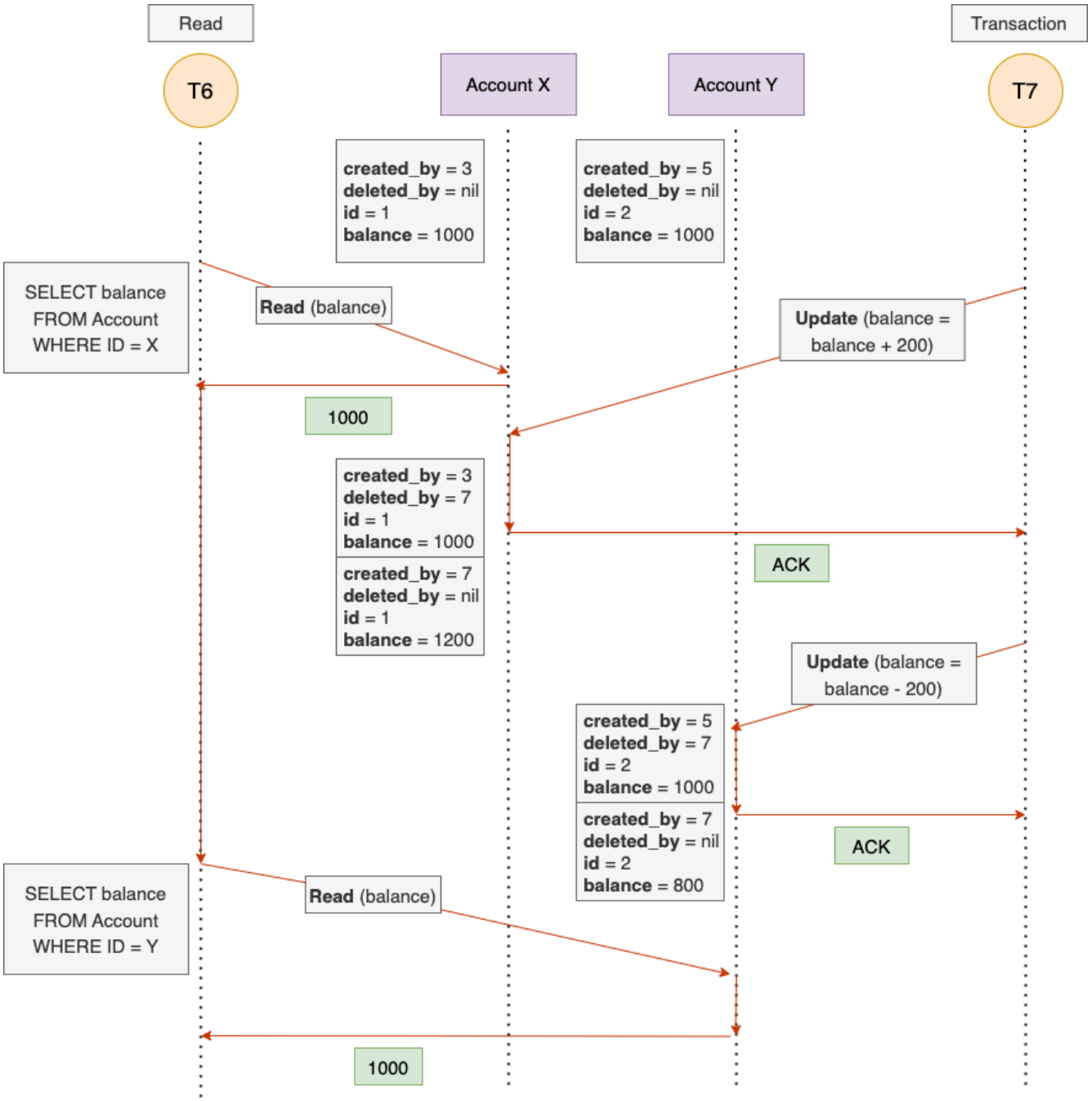
In this scheme the database preserves multiple copies of the same data-item describing their values in various points of time. Let's understand what a single data-item looks like.



The **created_by** field in the above data-item object stores the Transaction ID that created the version of the data-item.
The **deleted_by** field in the above data-item object stores the Transaction ID that deleted the version of the data-item. When a data-item is deleted, it is not actually removed from the data-store immediately whereas the **deleted_by** field is set and is removed at some point later in time when it's safe to be deleted (not required by any active transaction).

In this scheme an **Update** operation is translated into Delete followed by a Create operation. When the value of a data-item is updated from **X** to **Y**, then the previous version having **X** is set to be deleted and a newer version having value **Y** is created.

Let's understand how this scheme avoids the previous **Non-Repeatable Read** problem faced by Alex while transferring some amount of money from one account to another.



Since now we have multiple versions of the data-items stored in the database, Alex reads the correct balances from his Accounts. While performing a read from Account **Y**, he reads a balance of **Rs.1000** and not **Rs.800**, since the later version of the data-item was created by transaction **ID 7** and won't be visible to the transaction reading the data-item with **ID 6**.

These are the visibility rules followed by the Snapshot Isolation scheme to decide which version of the data-item can be seen by the transactions:

- **Rule 1**: Any writes made by the transaction with a later transaction ID (which started after the current transaction) are ignored, regardless of whether those transactions have been committed.

- **Rule 2**: The writes performed by aborted transactions are ignored.

Thank You! ❤️

Hope this handbook helped you in clearing out the concept of **Transactions** and **Concurrency Control** in **Distributed Systems**.

Follow me on Linkedin
Do subscribe to my engineering newsletter “Systems That Scale”