Michelle Tan
Anna Rosenberg
Calista Kerins

Complex Transactions Concurrency Write Ups:

1. **add_recipe():**
   a. **Dirty Read**: In the add_recipe function, a dirty read could occur if one transaction reads data that has not been committed yet. For example, if transaction T1 inserts a new recipe but hasn't committed it yet, and transaction T2 queries for recipes, T2 may see the uncommitted recipe inserted by T1.
   b. **Non-Repeatable Read:** Non-repeatable read can occur when a transaction reads the same data multiple times but gets different values each time. In the add_recipe function, if another transaction updates the same recipe that is being read by the current transaction, the current transaction may get different values for the recipe between its reads.
2. **favorite_recipe():**
   a. **Dirty Read**: A dirty read can occur when multiple transactions are accessing and modifying the same data at the same time. In this function, if two or more users attempt to favorite the same recipe at the same time, there is a possibility of a dirty read. One transaction may read the data from the favorited_recipes table before another transaction has committed its changes, and this would lead to an inconsistent state where the same recipe is favorited by multiple users simultaneously.
   b. **Lost Update**: The lost update phenomenon can occur when two transactions attempt to update the same data at the same time. In this function, if multiple users simultaneously attempt to favorite the same recipe, a lost update can happen. One transaction may read the current number of favorites for the recipe, and before it updates the value, another transaction may also read and update the number of favorites. As a result, the first transaction's update may be lost, leading to a wrong count of favorites.
3. **unfavorite_recipe():**
   a. **Dirty Read**: Similar to the favorite_recipe function, a dirty read can occur if multiple transactions are accessing and modifying the same data at the same time. In this case, if two or more users attempt to unfavorite the same recipe concurrently, a dirty read may happen. One transaction may read the favorited_recipes table before another transaction commits its changes, where the recipe is unfavorited multiple times.
   b. **Lost Update:** The lost update phenomenon may occur when two transactions attempt to update the same data at the same time. In this function, if multiple users simultaneously try to unfavorite the same recipe, a lost update can happen. One transaction may read the current number of favorites for the recipe, and before it updates the value, another transaction may also read and update the number of favorites. As a result, the first transaction's update may be lost, resulting in an incorrect count of favorites.

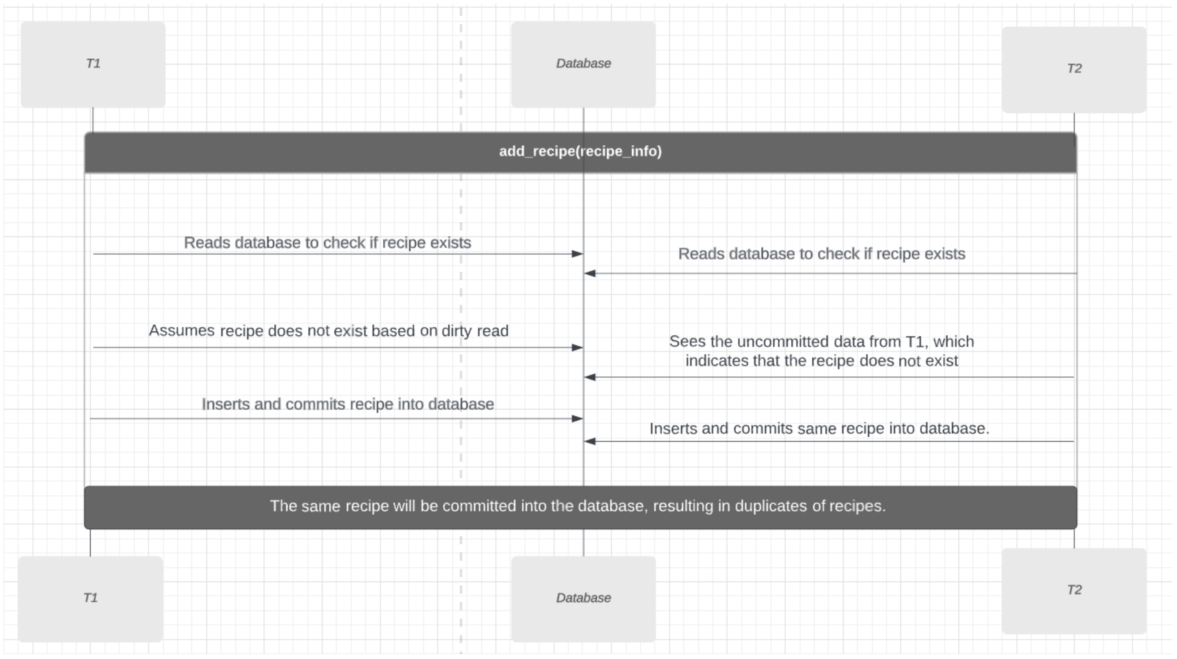
Ensure Isolation

I would implement the Serialization isolation level for these transactions because it prevents phenomena such as dirty reads, non-repeatable reads, and lost updates. It would prevent concurrent transactions from interfering with each other.

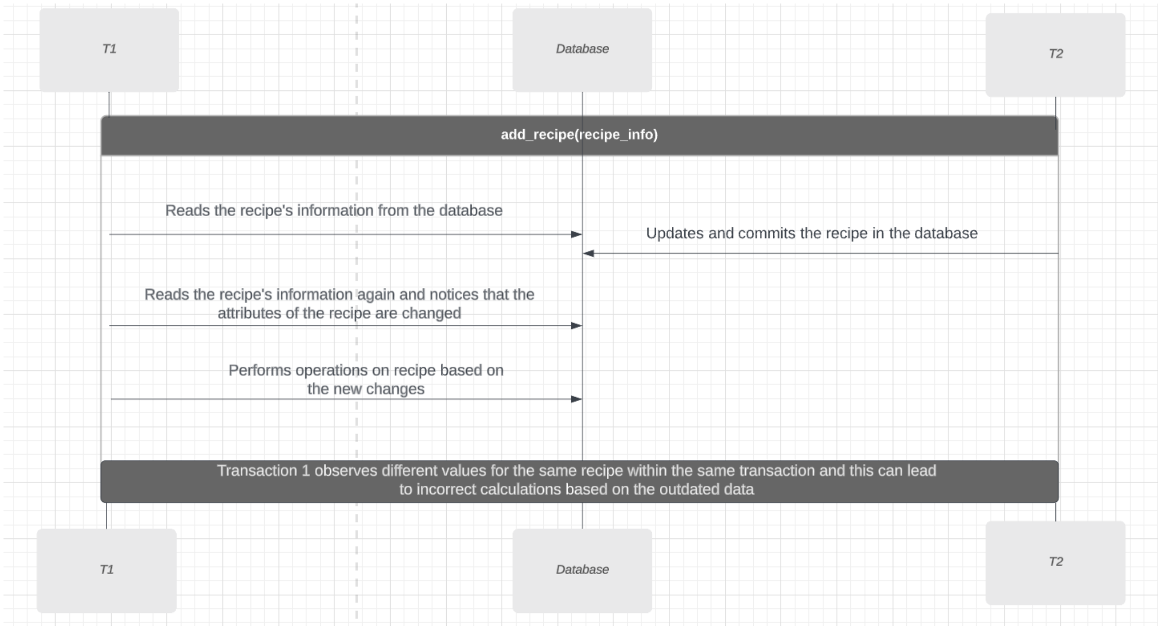
Sequence Diagrams

add_recipe():

1. **Dirty Read**

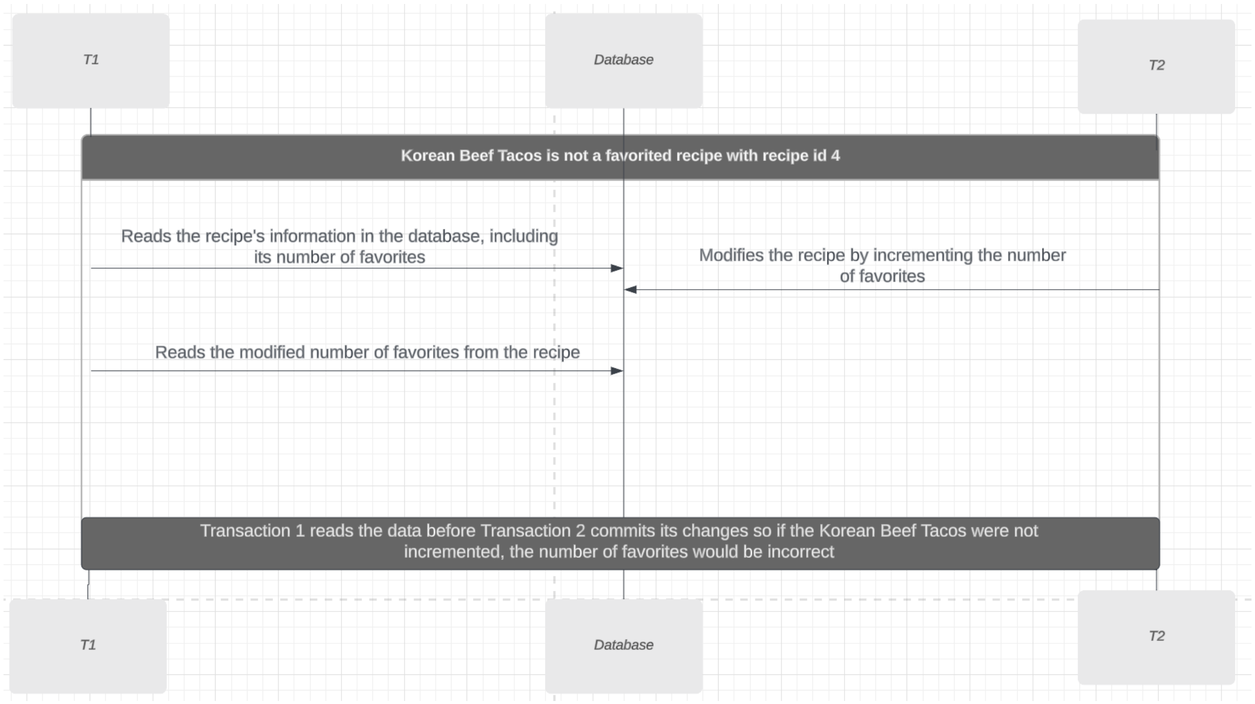

T1
Database
T2
add_recipe(recipe_info)
Reads database to check if recipe exists
Reads database to check if recipe exists
Assumes recipe does not exist based on dirty read
Sees the uncommitted data from T1, which indicates that the recipe does not exist
Inserts and commits recipe into database
Inserts and commits same recipe into database.
The same recipe will be committed into the database, resulting in duplicates of recipes.
T1
Database
T2

2. **Non-Repeatable Read**

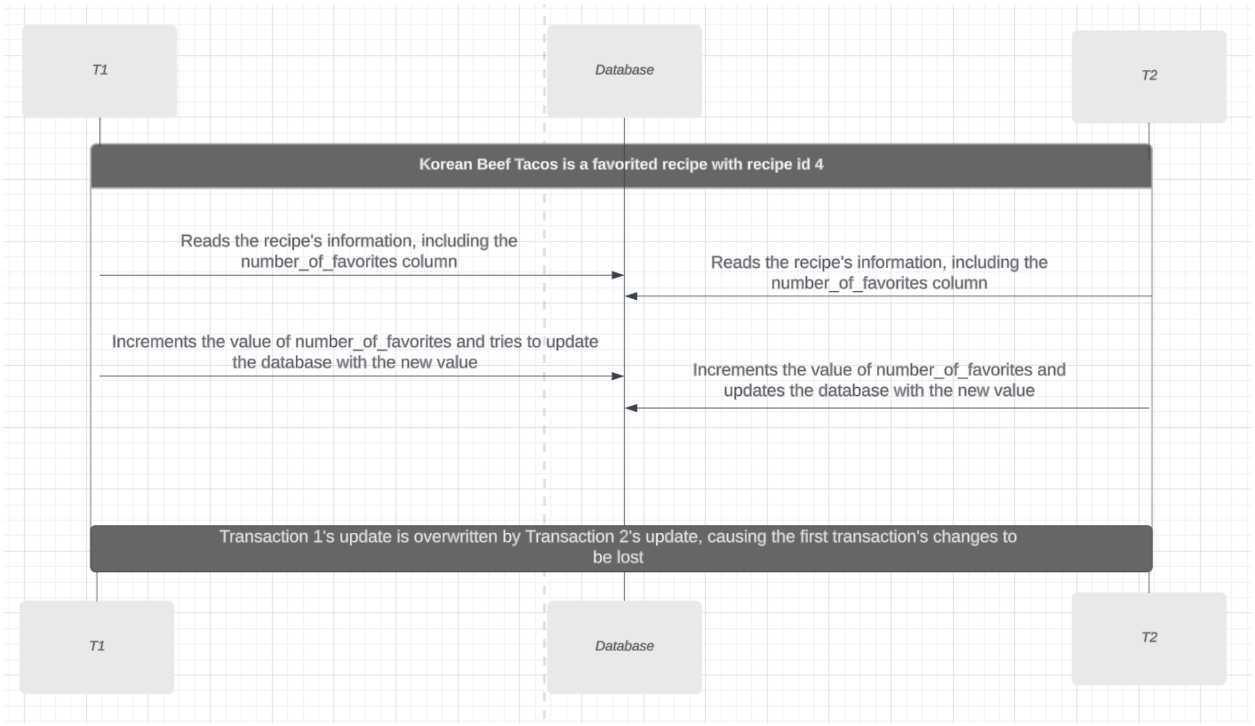

T1
Database
T2
add_recipe(recipe_info)
Reads the recipe's information from the database
Updates and commits the recipe in the database
Reads the recipe's information again and notices that the attributes of the recipe are changed
Performs operations on recipe based on the new changes
Transaction 1 observes different values for the same recipe within the same transaction and this can lead to incorrect calculations based on the outdated data
T1
Database
T2

favorite_recipe()

1. **Dirty Read**

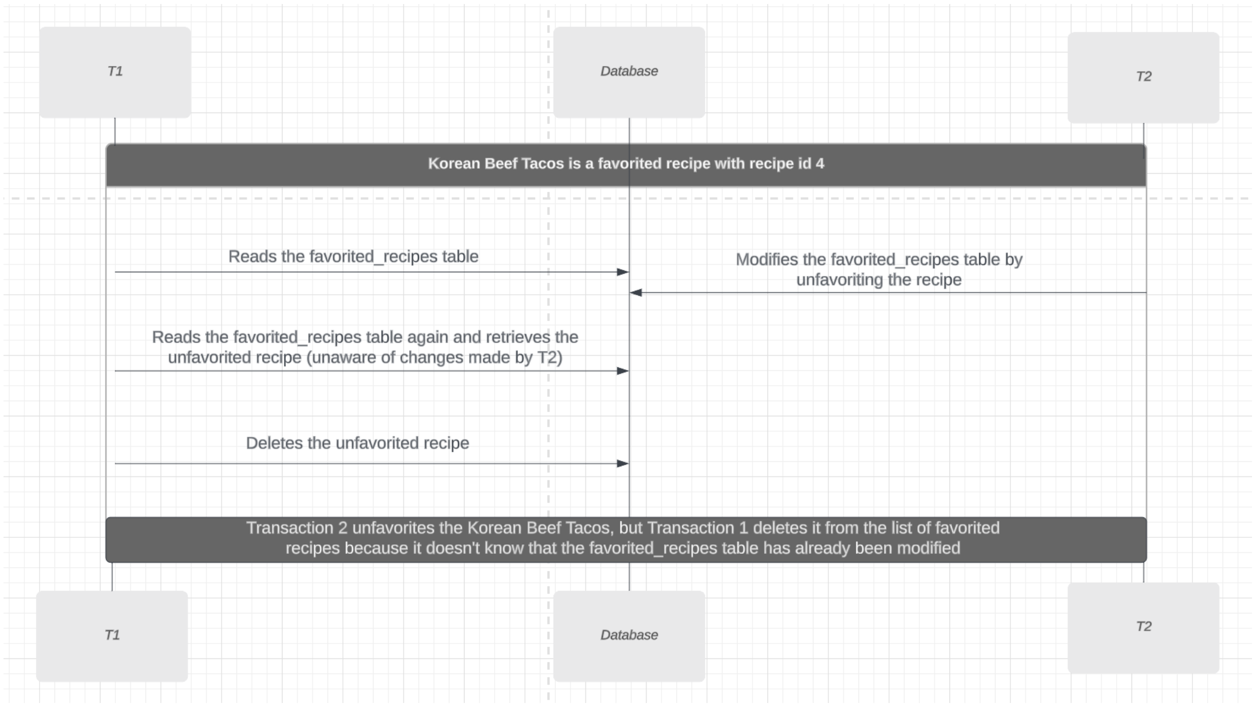

T1
Database
T2
Korean Beef Tacos is not a favorited recipe with recipe id 4
Reads the recipe's information in the database, including its number of favorites
Modifies the recipe by incrementing the number of favorites
Reads the modified number of favorites from the recipe
Transaction 1 reads the data before Transaction 2 commits its changes so if the Korean Beef Tacos were not incremented, the number of favorites would be incorrect
T1
Database
T2

2. **Lost Updates**

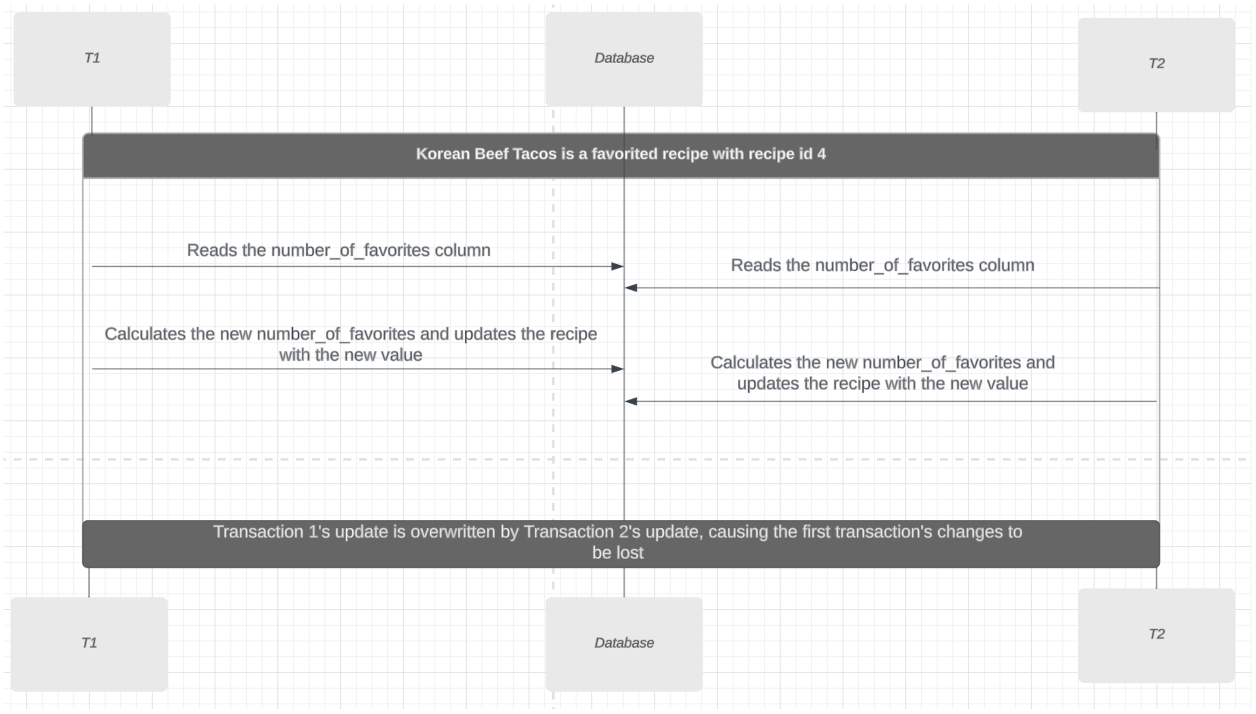

T1
Database
T2
Korean Beef Tacos is a favorited recipe with recipe id 4
Reads the recipe's information, including the number_of_favorites column
Reads the recipe's information, including the number_of_favorites column
Increments the value of number_of_favorites and tries to update the database with the new value
Increments the value of number_of_favorites and updates the database with the new value
Transaction 1's update is overwritten by Transaction 2's update, causing the first transaction's changes to be lost
T1
Database
T2

unfavorite_recipe()

1. **Dirty Read**


T1
Database
T2
Korean Beef Tacos is a favorited recipe with recipe id 4
Reads the favorited_recipes table
Modifies the favorited_recipes table by unfavoriting the recipe
Reads the favorited_recipes table again and retrieves the unfavorited recipe (unaware of changes made by T2)
Deletes the unfavorited recipe
Transaction 2 unfavorites the Korean Beef Tacos, but Transaction 1 deletes it from the list of favorited recipes because it doesn't know that the favorited_recipes table has already been modified
T1
Database
T2

2. **Lost Updates**


T1
Database
T2
Korean Beef Tacos is a favorited recipe with recipe id 4
Reads the number_of_favorites column
Reads the number_of_favorites column
Calculates the new number_of_favorites and updates the recipe with the new value
Calculates the new number_of_favorites and updates the recipe with the new value
Transaction 1's update is overwritten by Transaction 2's update, causing the first transaction's changes to be lost
T1
Database
T2