INTERVIEW READY

Payment Tracking App - Low Level Design

Prioritized Requirements

- Users can **add expenses.**
- Users can **edit expenses.**
- Also, users can **settle expenses.**
- Allow users to **make groups** and add, edit and settle expenses in the group.

Requirements not a part of our design

- Comments for records.
- Activity log for each and every event.
- Authentication service.

Objects definition

- **User object**

```
struct User{
userID uid;
string ImageURI;
string bio;
}
```

- **Balance object**

  There are **2** types of people:

  i. People who will receive money from others. They have a positive balance.

  ii. People who will pay money to others. They have a negative balance.

  So our balance object should be able to handle both positive and negative values.

```
struct Balance{
string currency;
int amount; //For simplicity we are using integers here.
// We can also use float or double but in that case we also need to handle precision errors.
}
```

- **Expense object**

  Expense object must map each user to their balance.

```
  struct Expense{
  ExpenseID eid;
  bool isSettled;
  map<User,Balance>;
  GroupID gid; //This expense belongs to which group

  //Metadata
  string title;
  int timestamp;
  string imageURI;
}
```

- **Group object**

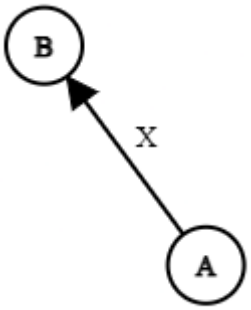
```
struct Group{
    GroupID gid;
    List<User> users;

    //metadata

    string ImageURI;
    string title;
    string description;
}
```

Behavior definition

- **Add Expense** We have user and balance object, and then we can persist it in database.
- **Edit Expense** Each expense object has a unique ID. We can use ID to change the mapping or other metadata.
- **Settle Expense** We make the `isSettled` flag true. We will use a balancing algorithm to settle expense.
- **Add, Settle and Edit expenses in group** Each expense object has groupID. So we can add, settle and edit expenses in group.

Balancing algorithm

- **Problem** Let us denote each user as a node and each payment as an edge in a graph.
  So we need to **minimize the number of edges in the graph**



It means A paid x amount to B.

- **Solution**
  Note : When we talk about balance we are talking about the sum of all transactions for a user.

  First let's divide the user into two categories
  - People who have positive balance.
  - People who have negative balance.

  If at point any user has 0 balance that means his/her expenses are settled, and we can remove that node from the graph.

  At each step we will pick the largest absolute value from each category and add an edge between them. Let's say we pick `A` from first category and `B` from second category. So will add an edge from B to A (because B will pay and A will receive) and then update the balances. If the new balance of any node becomes 0 then we will not consider the node again.

  To implement this we can use **heap data structure.**

  Pseudocode

```
struct Node{
    UserID uid;
    int finalBalance;
}

struct PaymentNode{
    UserID from;

    UserID to;
    int amount;
}

List<PaymentNode> makePaymentGraph(){

  Max_Heap<Node> firstCategory;
  Max_Heap<Node> secondCategory; //We are storing the absoulte value

  List<PaymentNode> graph;

  //The sum of balance of all users always results in 0 so if first heap is empty then second h
  loop(firstCategory.isNotEmpty()){
      receiver = firstCategory.top();
      sender = secondCategory.top();

      //Removing the element
      firstCategory.pop();
      secondCategory.pop();

      amountTransferred = min(sender.finalBalance , receiver.finalBalance);

      graph.insert(PaymentNode(sender.uid, receiver.uid, amountTransferred));
```

```
            sender.finalBalance -= amountTransfered;
            receiver.finalBalance -= amountTransfered;

            if(sender.finalBalance != 0)
                secondCategory.push(sender);

            if(receiver.finalBalance != 0)
                firstCategory.push(receiver);

        }

        return PaymentGraph;

    }
```

- **Edge case**
  Let us consider a case

A	B	C	D	E	F	G
80	25	-25	-20	-20	-20	-20

  If we use our approach then the payments will be
  - C will pay $25 to A
  - D will pay $20 to A
  - E will pay $20 to A
  - F will pay $20 to B
  - G will pay $15 to A
  - G will pay $5 to B

  So there are **6 transactions.**
  But it can be done in only **5** transactions
  - C will pay $25 to B
  - D will pay $20 to A
  - E will pay $20 to A
  - F will pay $20 to A
  - G will pay $20 to A

API Design

- **Expenses[] getGroupExpenses(GroupID gid)**
- **PaymentGraph getGroupPaymentGraph(GroupID gid)**
- User getUser(UserID uid)
- User[] getUsersInGroup(GroupID gid)
- ExpenseID addExpense(GroupID gid, UserID uid, int amount, string currency)
- void editExpense(ExpenseID eid, UserID uid, int amount, string currency)
- void settleExpense(ExpenseID eid)
- GroupID makeGroup(User[] users, string name, string imageURI, string description)
- Group getGroup(GroupID gid)
- Expense getExpense(ExpenseID eid)

Caching

We want to cache responses that are **required by many users** or are **expensive to calculate**. So will cache response of following APIs.

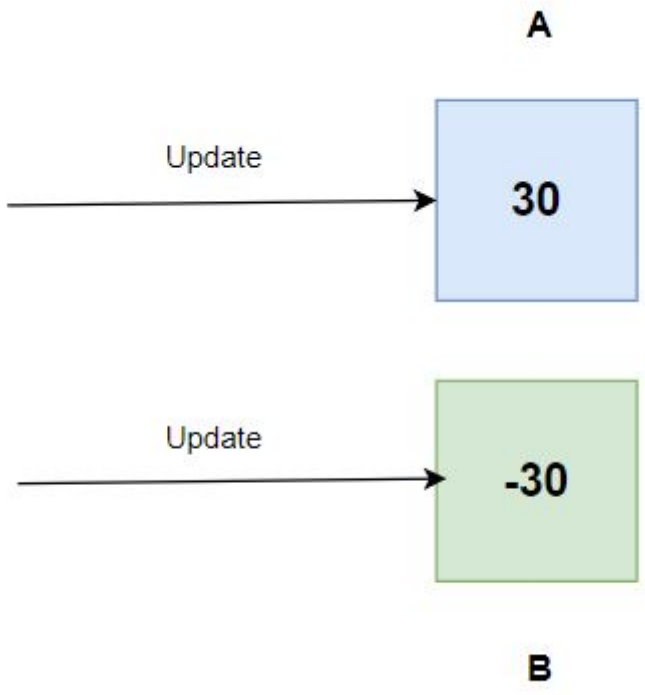
1. Expenses[] getGroupExpenses(GroupID gid)
2. PaymentGraph getGroupPaymentGraph(GroupID gid)

Data consistency

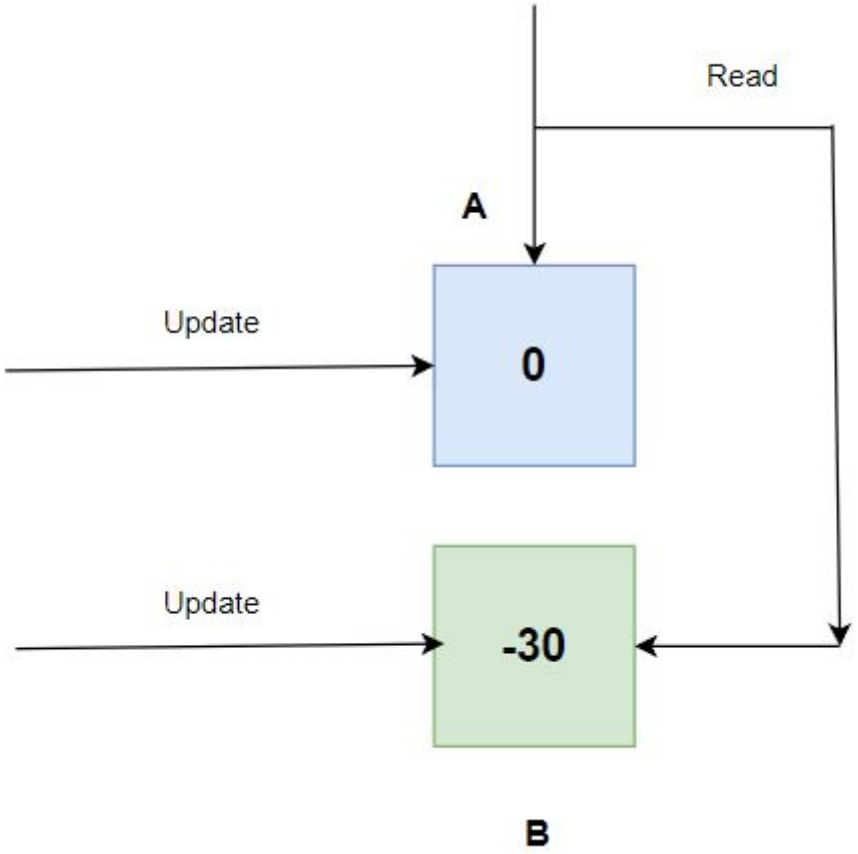
When performing simultaneous read and update operations it might happen that the data users get is incorrect. This is called **data inconsistency.**

Let's understand this using an example:

We have two users A and B. A received $30 from B. So we perform an update expression



Now the update operation on A was completed but not for B. And at the exact moment there was a read request from user.

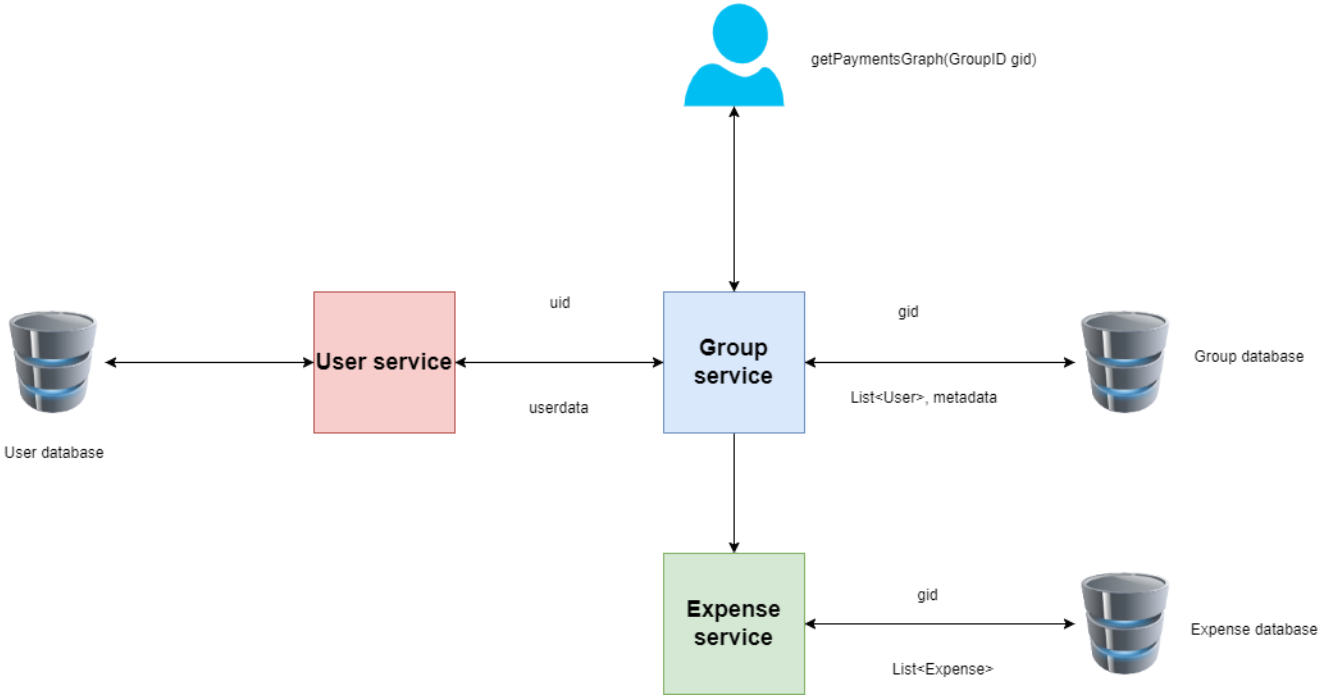


The sum of all balances should be 0 but in this case it is -30. So the data is inconsistent.

There are **2 ways we can solve this problem**

- **Put read lock when updating expenses** While updating the expense objects we will not allow users to read data. It will make sure that the data users get will be consistent.
- **Make objects immutable** So whenever there is an edit request we will create a new object and update data in new object. Once the update is completed we will point the reference to the new object. It might happen that users get old data however we can be sure that data will be consistent. This will make our system **eventually consistent**.

Diagram



That's it for now!

You can check out more designs on our video course at InterviewReady.