Written by Anonymous
on August 24, 2020

Venmo

Next challenge: design Venmo.

Identifying functionalities

Finally one I have used recently. Venmo lets you:

- Send money from people
- Request money from people
- Make transactions publicly or privately
- Friend people
- See a feed of global public transactions
- See a feed of friends' public transactions
- See a feed of your transactions
- Integrate to move money in and out
- I think it also lets you see your transaction history for a particular person, but I am not sure
- Some stuff with emoji

Pulling apart key functionalities, we have:

- know and update current balance

- send and received transaction requests
- see a log of public transactions among anyone
- see a log of public transactions with friends
- see a log of public transactions with you
- friend people/see if someone is a friend
- see a list of friends

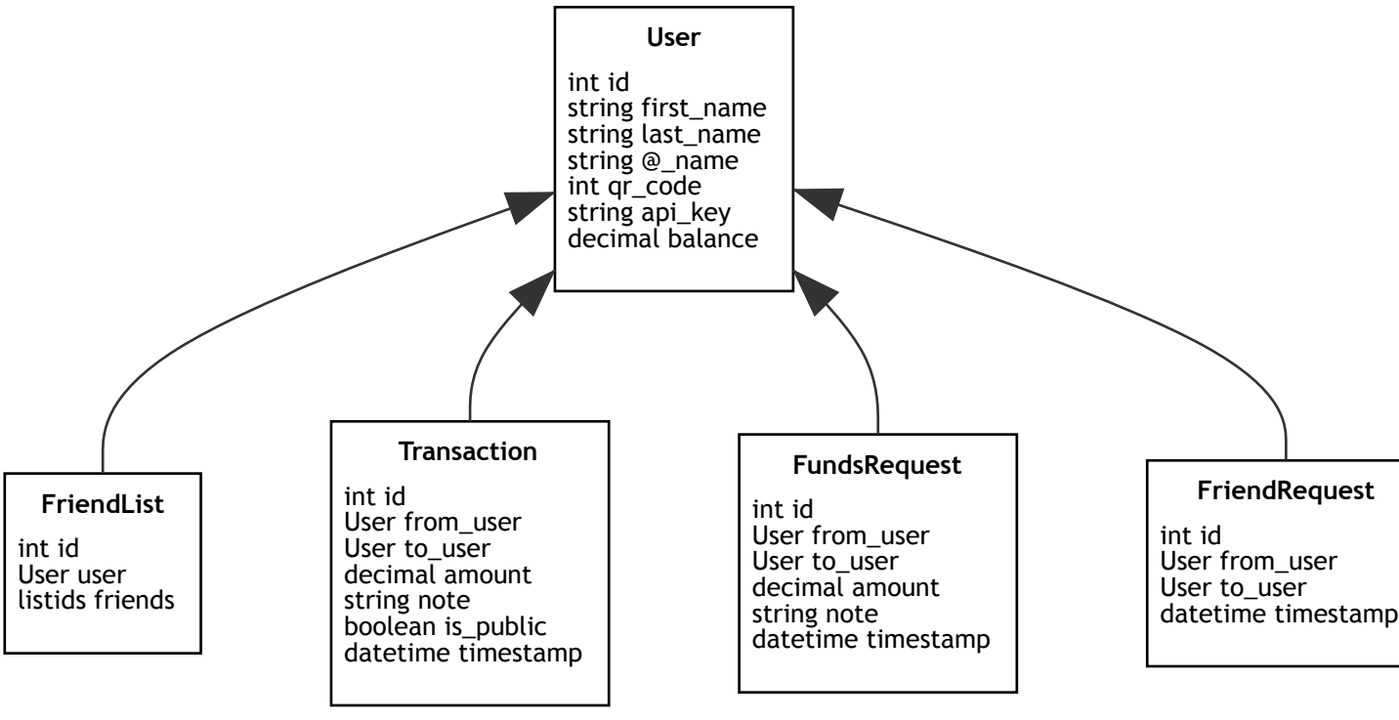
I will leave integration with banks and emoji things off the table for now.

High-level observations

Financial data immediately says “SQL” to me, for consistency guarantees and transactionalization. Updating balances and not allowing payments without capital is the canonical example used in transactions, so a store without consistency guarantees is probably not good enough for the balances.

The transaction history and transaction feeds don’t sound like they need consistency guarantees.

Entities and Relationships



Workflows

Just to start. There is more about feeds later.

1. Find a user by partial username search
2. Send a user money by name
3. Request money from a user by name
4. Issue friend request
5. Confirm friend request

Access patterns

Users making requests of eachother. No two-tiered system of access like Venmo. No special permissions other than "public is public, not-public is private to the transactors".

API Access

```
users_like(api_key, search_string)
send_money(api_key, to_user, amount, note,
is_public)
request_money(api_key, from_user, amount, note,
is_public)
confirm_request_money(api_key, request)
friend_request(api_key, new_friend)
confirm_friend_request_user(api_key, request)
```

Market Sizing

I will guesstimate that:

1. Venmo has 50M users, given that it is mostly used in the US and by young people.
2. About a tenth of venmo users use it on any given day.
3. Usage is spiky around mealtimes/late nights and US-concentrated.

Data Sizing

Memory

- 50M users * 12 bytes for balance, id, etc + some strings ~= 100 bytes per user ~= **5GB of user data** including balances
- Users might easily have 100 venmo transactions in a year, so **500GB of transaction storage**, 2.5TB in 5 years.
- Users might easily have 100 friend requests transactions, so **1GB of friend request storage**, x5 years
- Users have more friend requests than cash requests, in general, so <**1GB of data for cash requests**, x5 years
- Users might have 100s of friends, so ~400B * 50M ~= **20GB in friend storage**.

IO

- 50M requests/day is about 60 requests/second for writes, spiky.
- Transactions are seen more often than they are written because of the global and friend feeds.
- Guessing API keys are 16 bytes, sizes of API calls are

```
users_like(api_key, search_string)
# ~35B
send_money(api_key, to_user, amount, note,
is_public)                          # ~50B
request_money(api_key, from_user, amount, note,
is_public)                      # ~50B
confirm_request_money(api_key, request)
# ~20B
friend_request(api_key, new_friend)
# ~20B
confirm_friend_request_user(api_key, request)
# ~20B
```

5M money-send writes a day means ~250MB of IO a day.

The other quantities are smaller than this, so we might conservatively double to 500MB/day in writes. At a 10x read multiplier, that's 5GB/day in read-io.

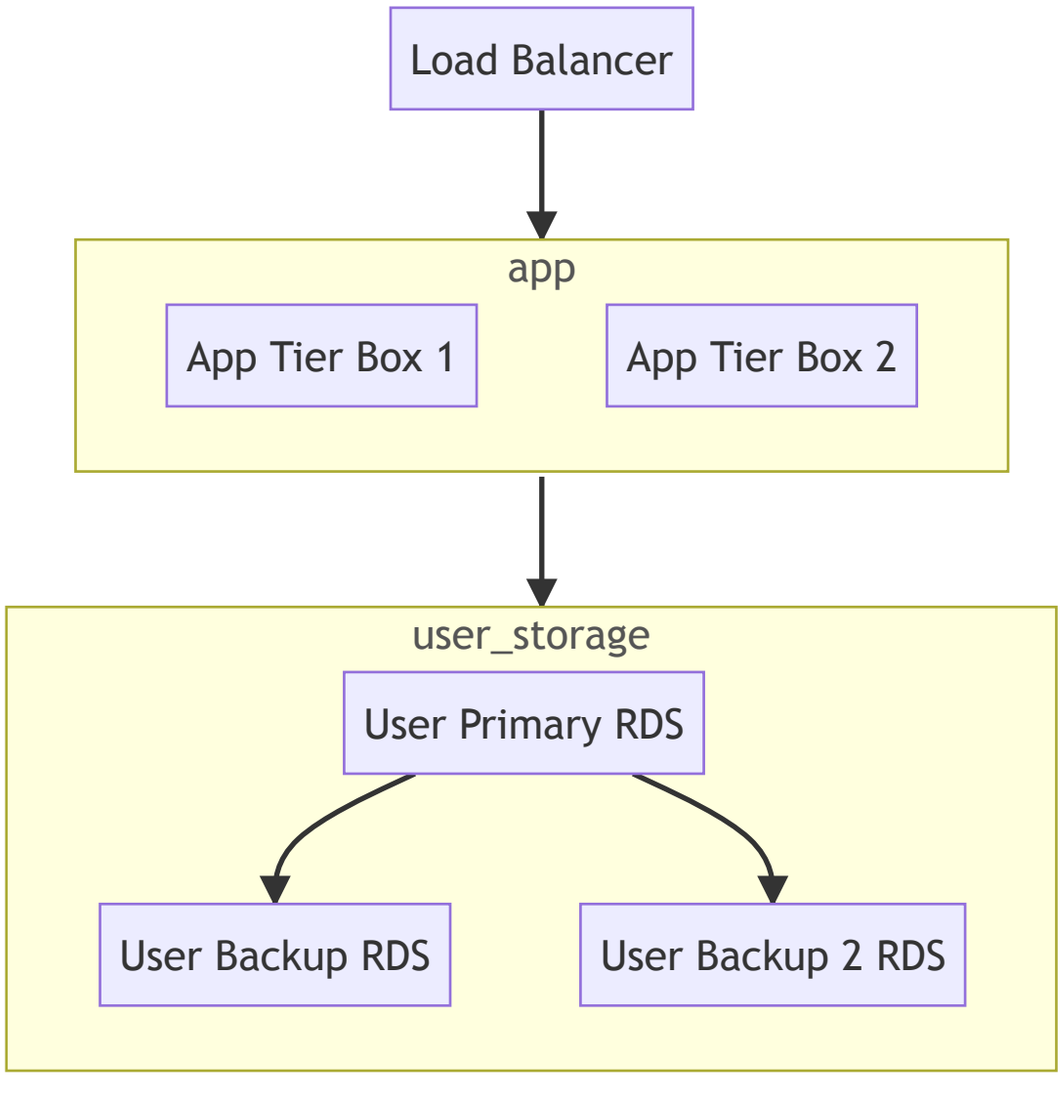
Spiky IO might conservatively quintuple the transaction volume during peak times, so 300 writes/second.

Architecture

Because of row-level locking, Postgres should not have contention for locks on balance-changing transactions very often, and they should clear quickly when they do.

We might consider breaking balances into a separate table just so user-data updating operations don't lock the tables when balance updates might need to come in.

Users, balances, and requests are all small, so can easily fit in an RDS Postgres box.



Transactions

500GB is too large for the largest RDS boxes, so we need scalable NoSQL storage.

Sharding on a hash of to_user will let us distribute the load (I don't think anyone issues requests fast enough to personally be a problem, and the hash smooths things to avoid clumping in a single partition). A local secondary index on timestamp lets us query for recency to show the personal receipt feed and global feed.

Questions I timed out on

Is it sufficient to query the transactions table twice to get the whole dataset, once for from_user and once for to_user?

I think the friend list is just a Redis store hashed on the friend-having account.