A design recipe

A notable example of NoSQL design for «distributed transactions»
Design recipe: banking account

• Banks are serious business
• They need serious databases to store serious transactions and serious account information
• They can’t lose or create money
• A bank **must** be in balance **all the time**
Design recipe: banking example

Say you want to give $100 to your cousin Paul for Christmas.
You need to:

decrease your account balance by 100$

```json
{
  _id: "account_123456",
  account: "bank_account_001",
  balance: 900,
  timestamp: 1290678353.45,
  categories: ["bankTransfer"...],
  ...
}
```

increase Paul’s account balance by 100$

```json
{
  _id: "account_654321",
  account: "bank_account_002",
  balance: 1100,
  timestamp: 1290678353.46,
  categories: ["bankTransfer"...],
  ...
}
```
What if some kind of failure occurs between the two separate updates to the two accounts?

- Decrease your account balance by 100$.
- Increase Paul’s account balance by 100$.
Design recipe: banking example

• What if some kind of failure occurs between the two separate updates to the two accounts?

- Decrease your account balance by 100$.
- Increase Paul’s account balance by 100$.

Bank  
Message lost during transmission

Send
Design recipe: banking example

• What if some kind of failure occurs between the two separate updates to the two accounts?

• CouchDB cannot guarantee the bank balance.
  • A different strategy (design) must be adopted.
Banking recipe solution

• What if some kind of failure occurs between the two separate updates to the two accounts?
• A NoSQL database without 2-Phase Commit cannot guarantee the bank balance → a different strategy (design) must be adopted.

```
id: transaction001
from: "bank_account_001",
to: "bank_account_002",
qty: 100,
when: 1290678353.45,
...```
Design recipe: banking example

• How do we read the current account balance?
• Map
  function(transaction){
    emit(transaction.from, transaction.amount*-1);
    emit(transaction.to, transaction.amount);
  }
• Reduce
  function(key, values){
    return sum(values);
  }
• Result
  {rows: [ {key: "bank_account_001", value: 900} ]}
  {rows: [ {key: "bank_account_002", value: 1100} ]}

The reduce function receives:
• key= bank_account_001,
  values=[1000, -100]
• ...
• key= bank_account_002,
  values=[1000, 100]
• ...