(paper published at PLDI'21)

CoSplit

Practical Smart Contract Sharding with Static Program Analysis

George Pîrlea, Amrit Kumar, Ilya Sergey

ZILLIQA · NUS National University of Singapore · YaleNUSCollege

Smart Contract *Sharding* with **Static Program Analysis**

$\approx$

Formal reasoning for *scalability*,

not correctness

Smart Contract *Sharding* with **Static Program Analysis**

≈

1 **Formal reasoning** for *scalability*,

not correctness

Smart Contract *Sharding* with **Static Program Analysis**

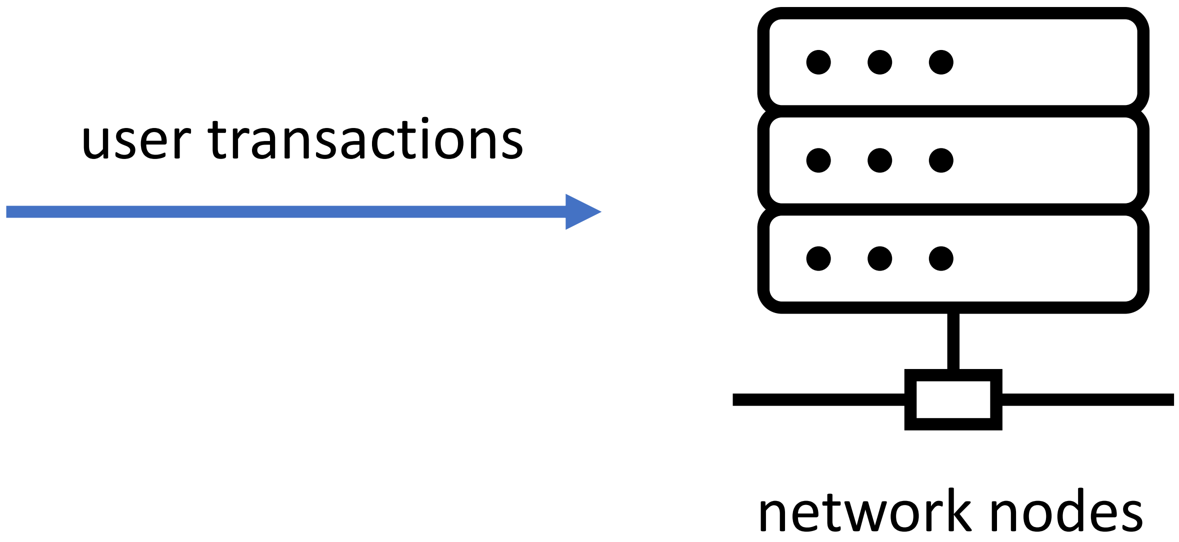
2

≈

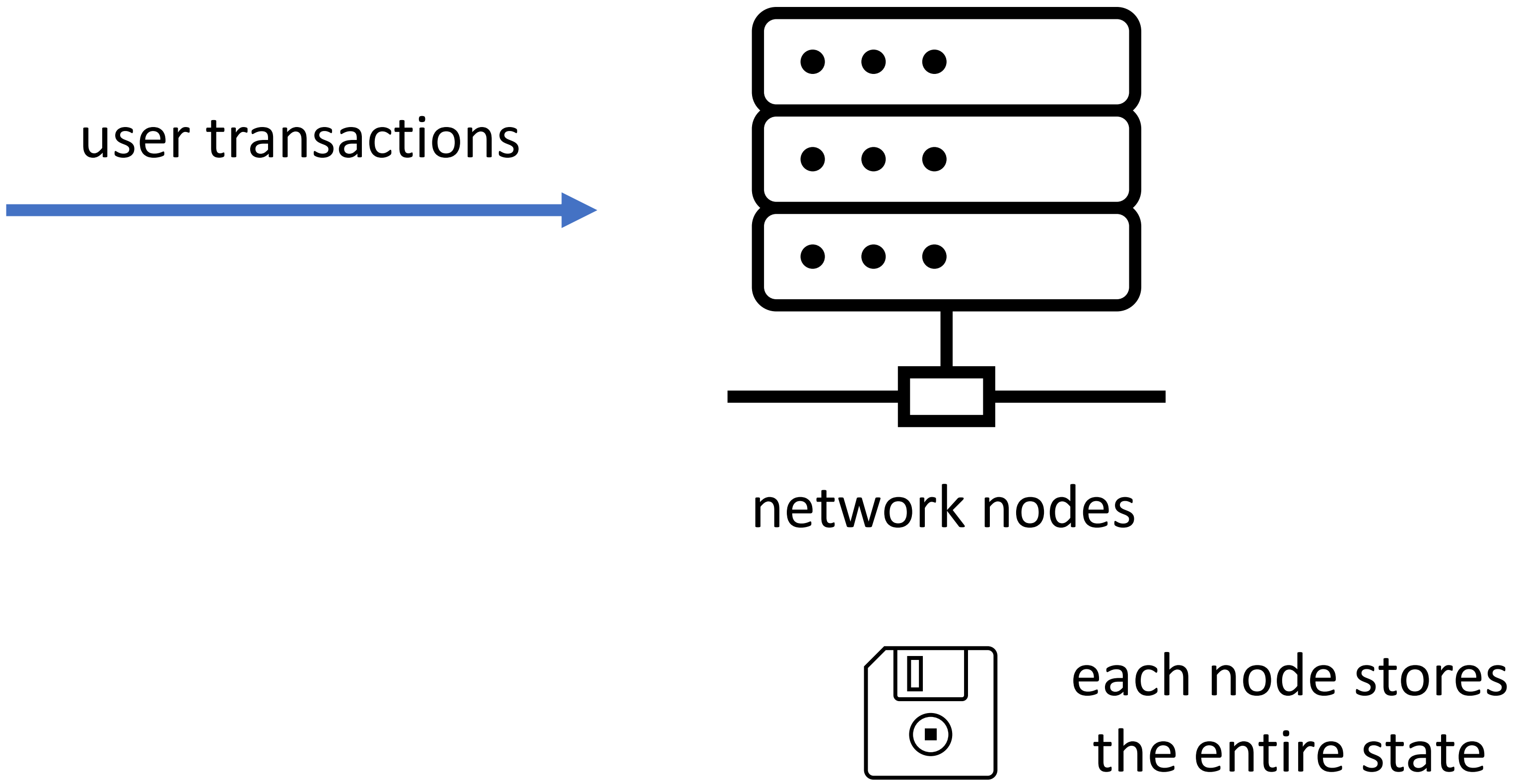
1 **Formal reasoning** for *scalability*,

not correctness

Scaling bottlenecks: state and execution



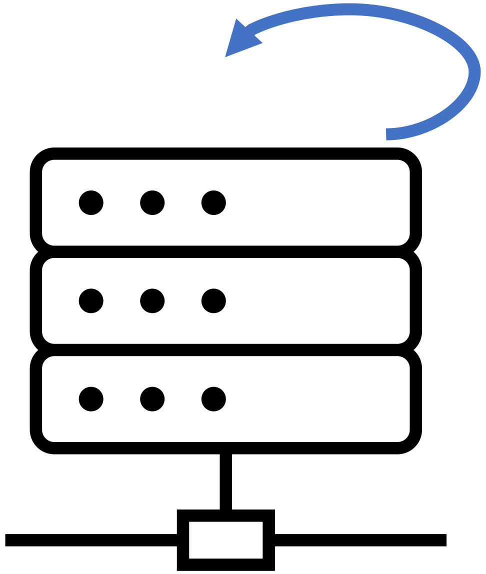
Scaling bottlenecks: state and execution



Scaling bottlenecks: state and execution

each node executes every transaction

user transactions

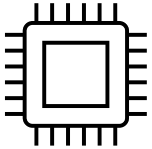


network nodes

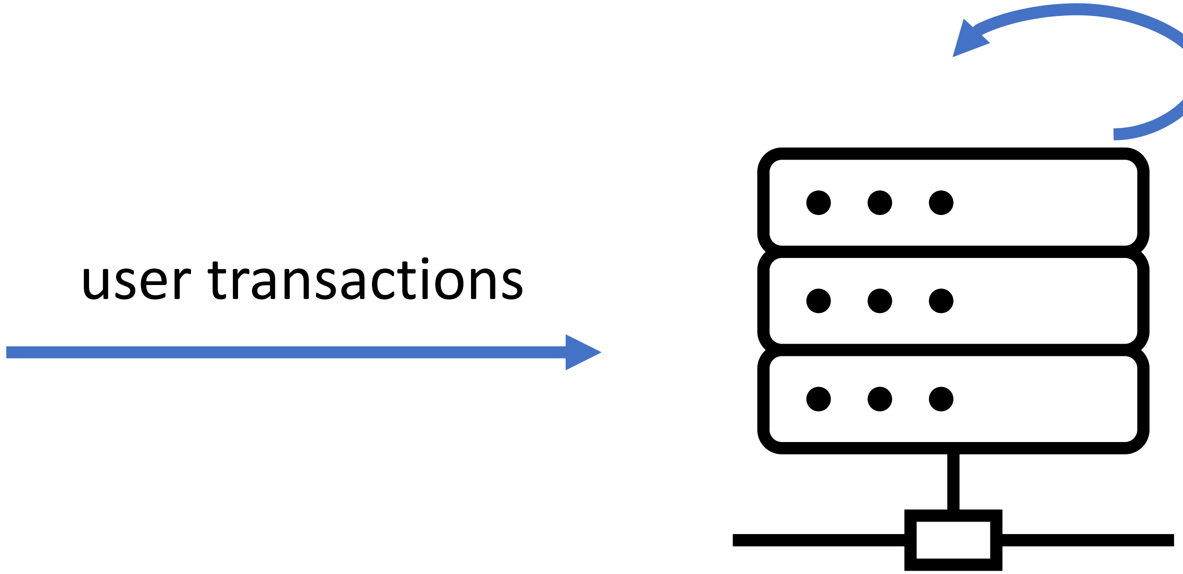
each node stores the entire state

Scaling bottlenecks: state and execution

each node executes every transaction

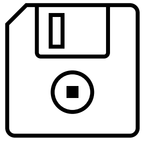


user transactions



network nodes

adding more nodes does not increase throughput OR allow more state to be stored



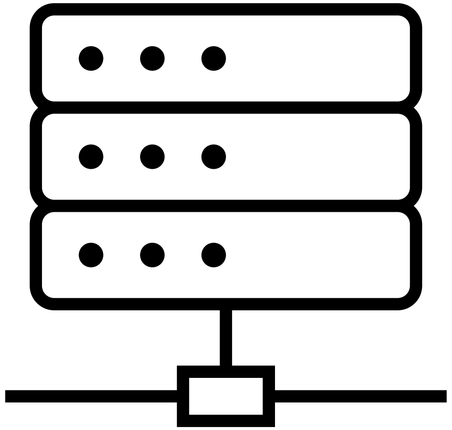
each node stores the entire state

[1] https://twitter.com/peter_szilagyi/status/1460202014919569410

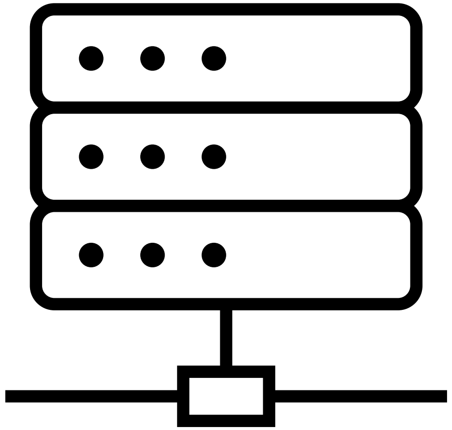
Scaling bottlenecks: state and execution

- Ethereum's head **state size** was ~130 GB as of Nov 2021[1]
  - For best performance, this needs to be kept in RAM
  - In practice, it is disk-based (NVME SSD) with caching in memory:
    - AFAIK, most of the time spent processing an Ethereum transaction is spent on disk I/O
- Increasing node hardware requirements decreases decentralisation
  - Solana validators already need >> 256 GB of RAM and 1 Gbps network
- In monolithic architectures, transaction **execution throughput** is limited by the capacity of the least performant node in the network

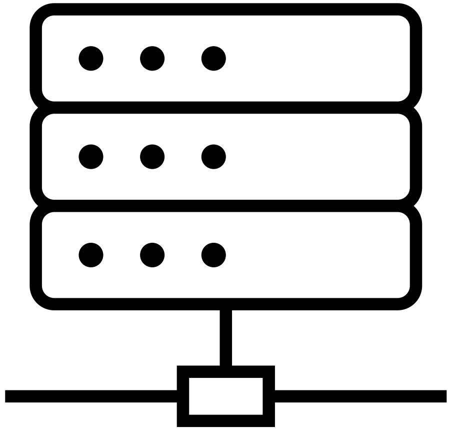
Sharding



shard 1 nodes



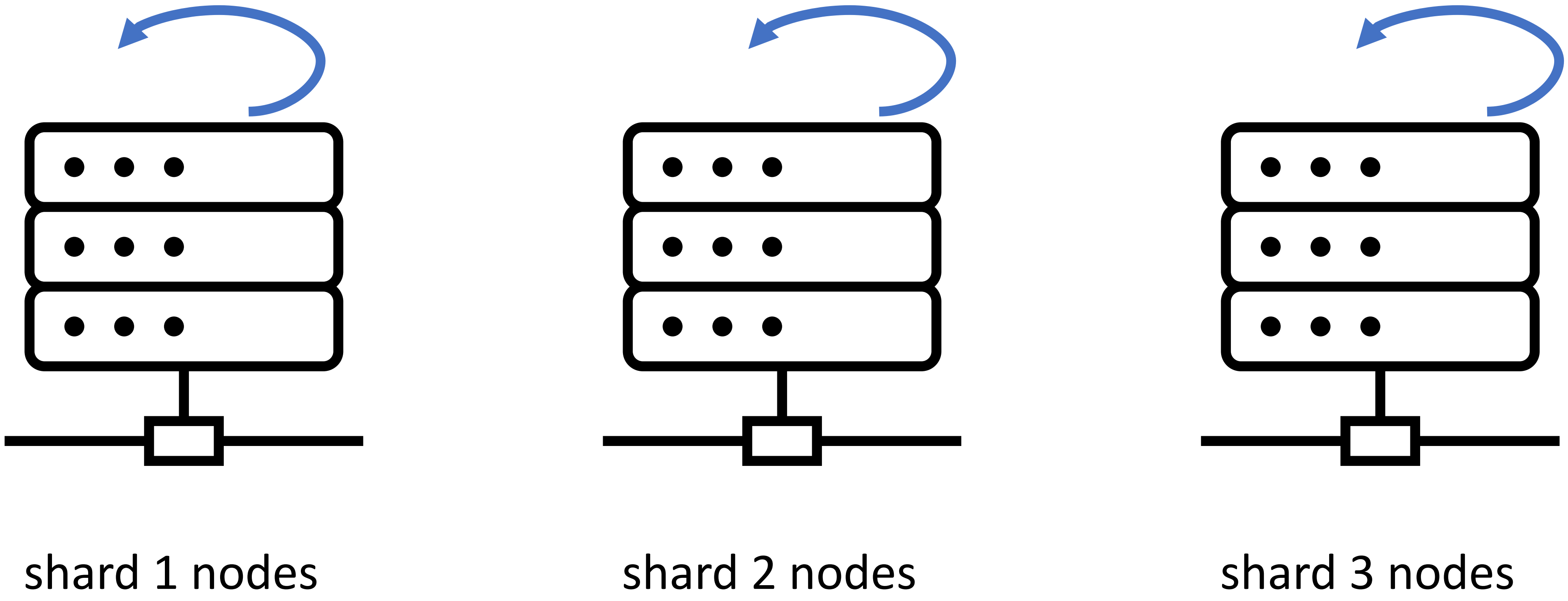
shard 2 nodes



shard 3 nodes

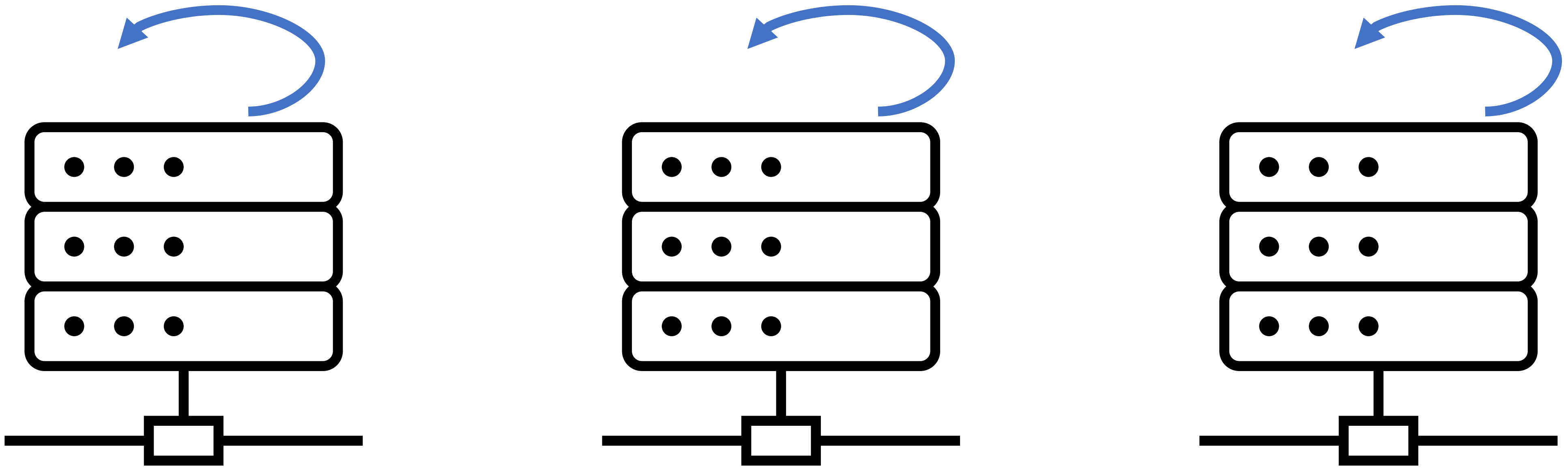
Sharding

each shard
executes a subset
of transactions



Sharding

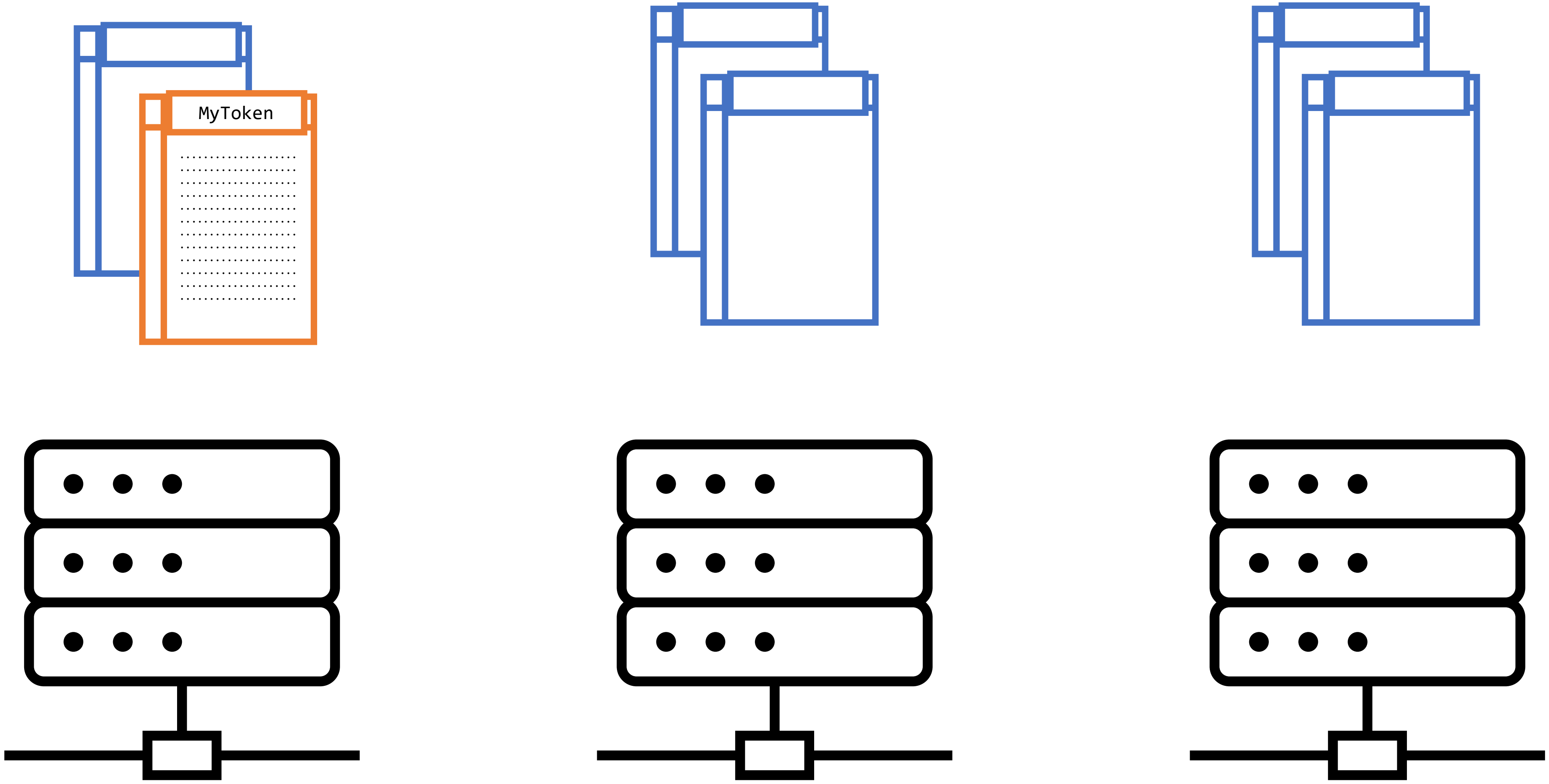
each shard
executes a subset
of transactions



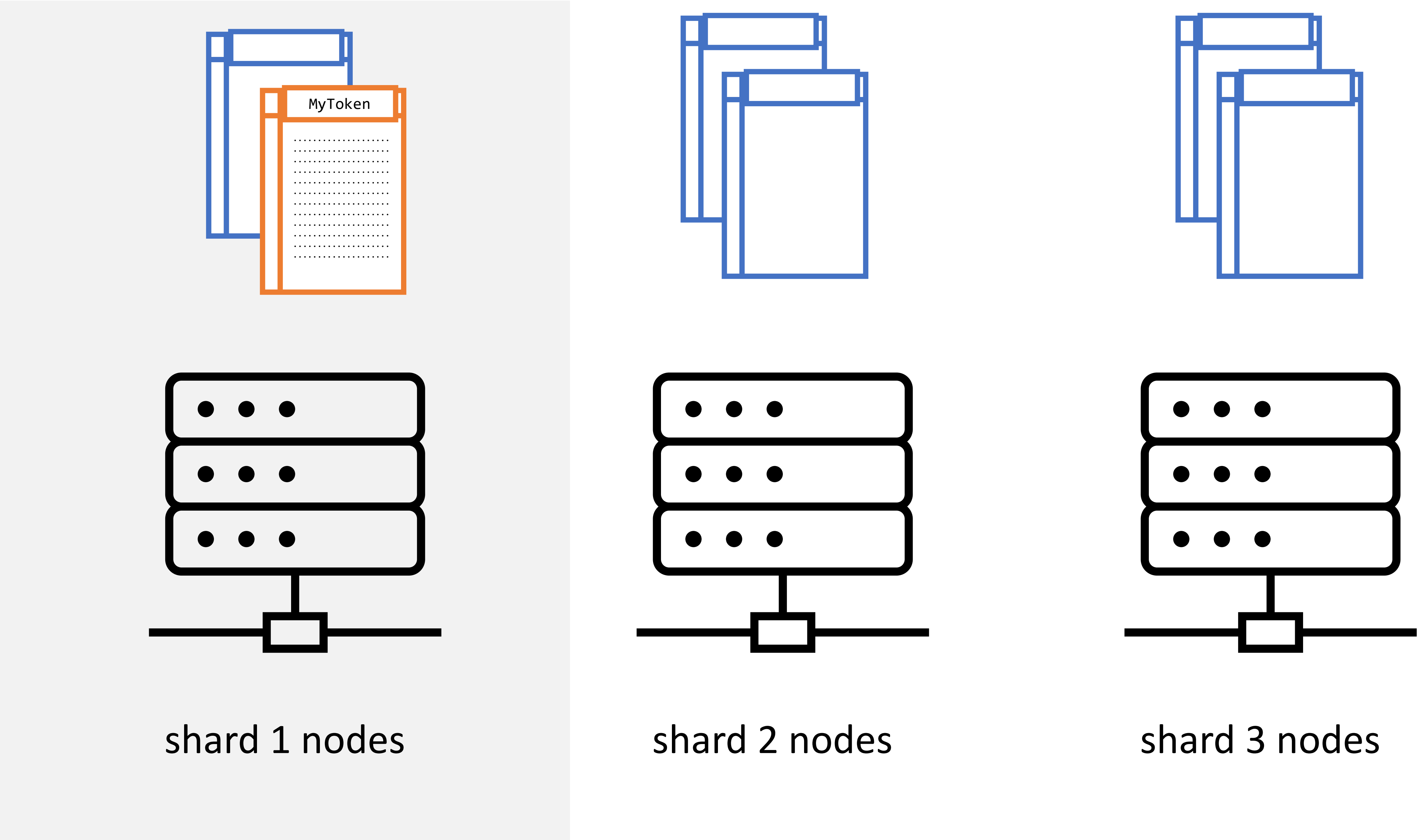
shard 1 nodes

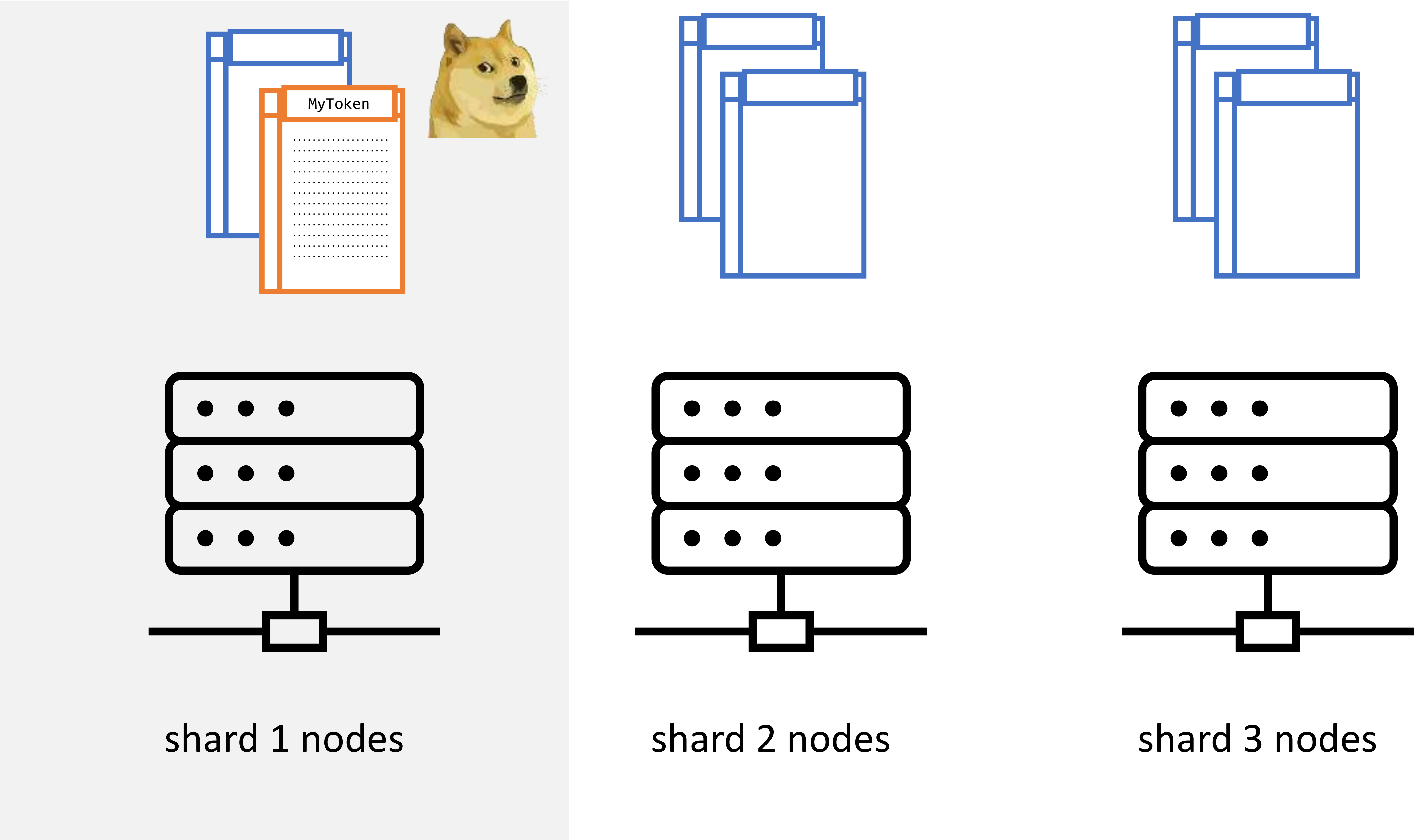
shard 2 nodes

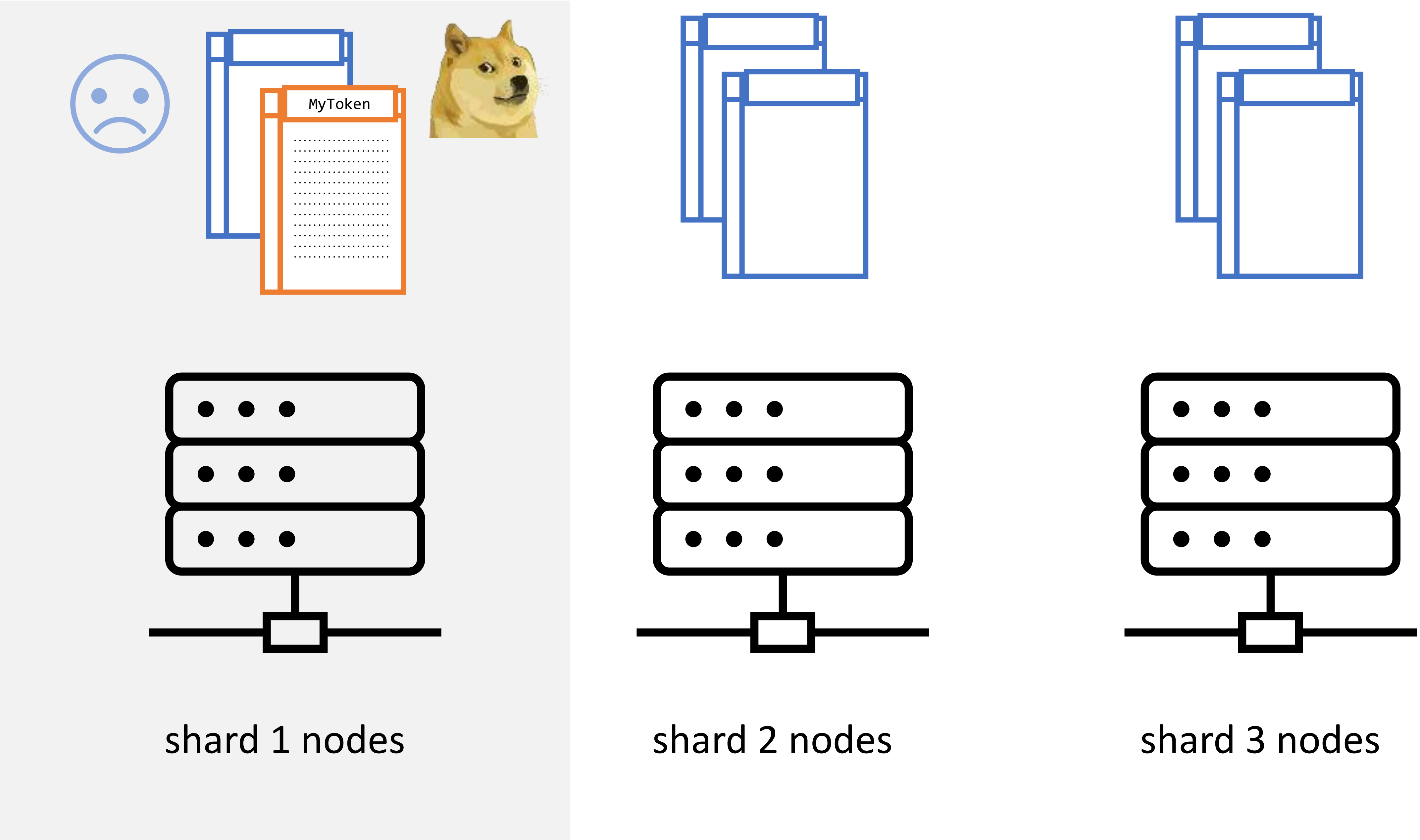
shard 3 nodes

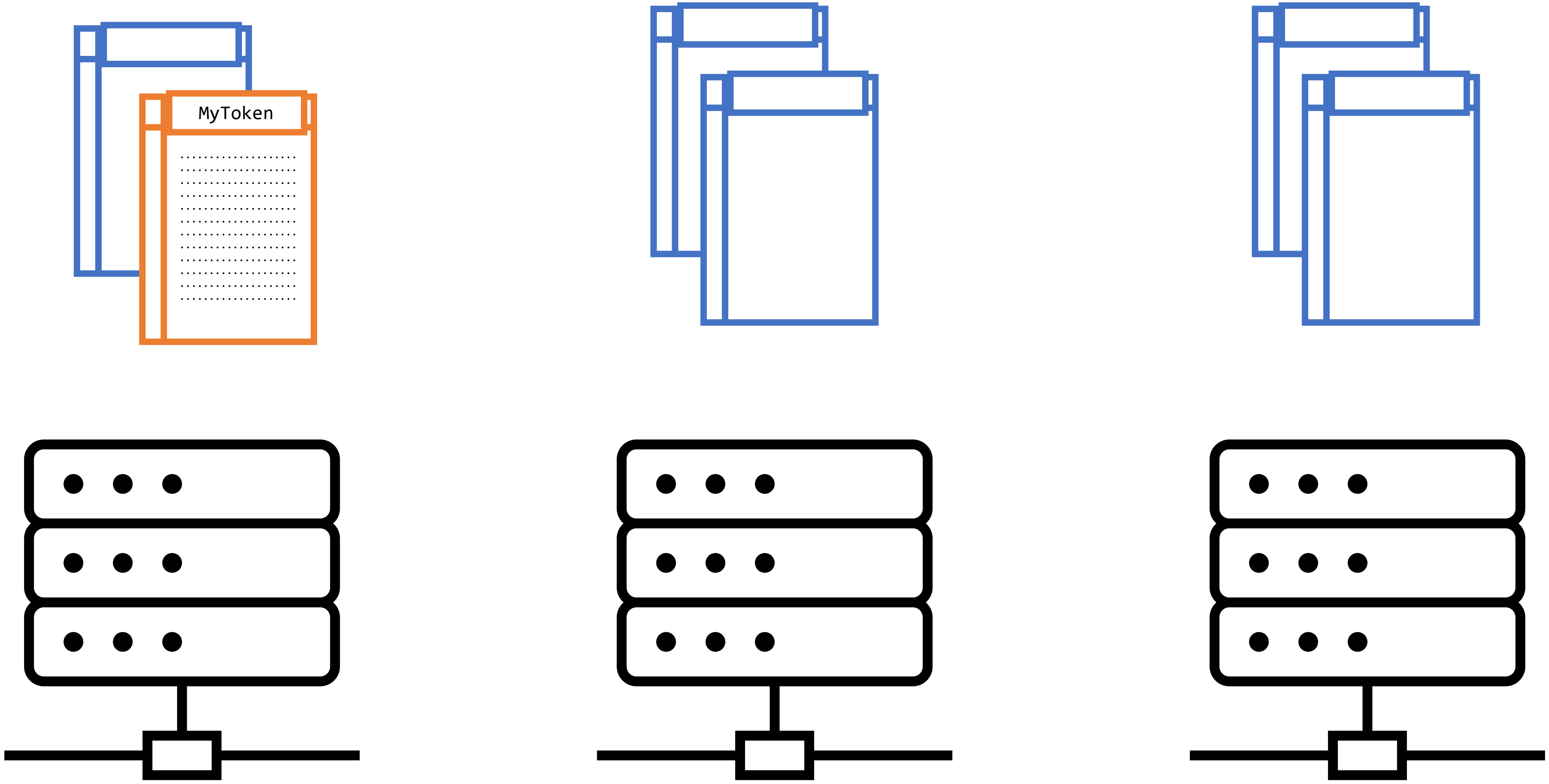


shard 1 nodes shard 2 nodes shard 3 nodes

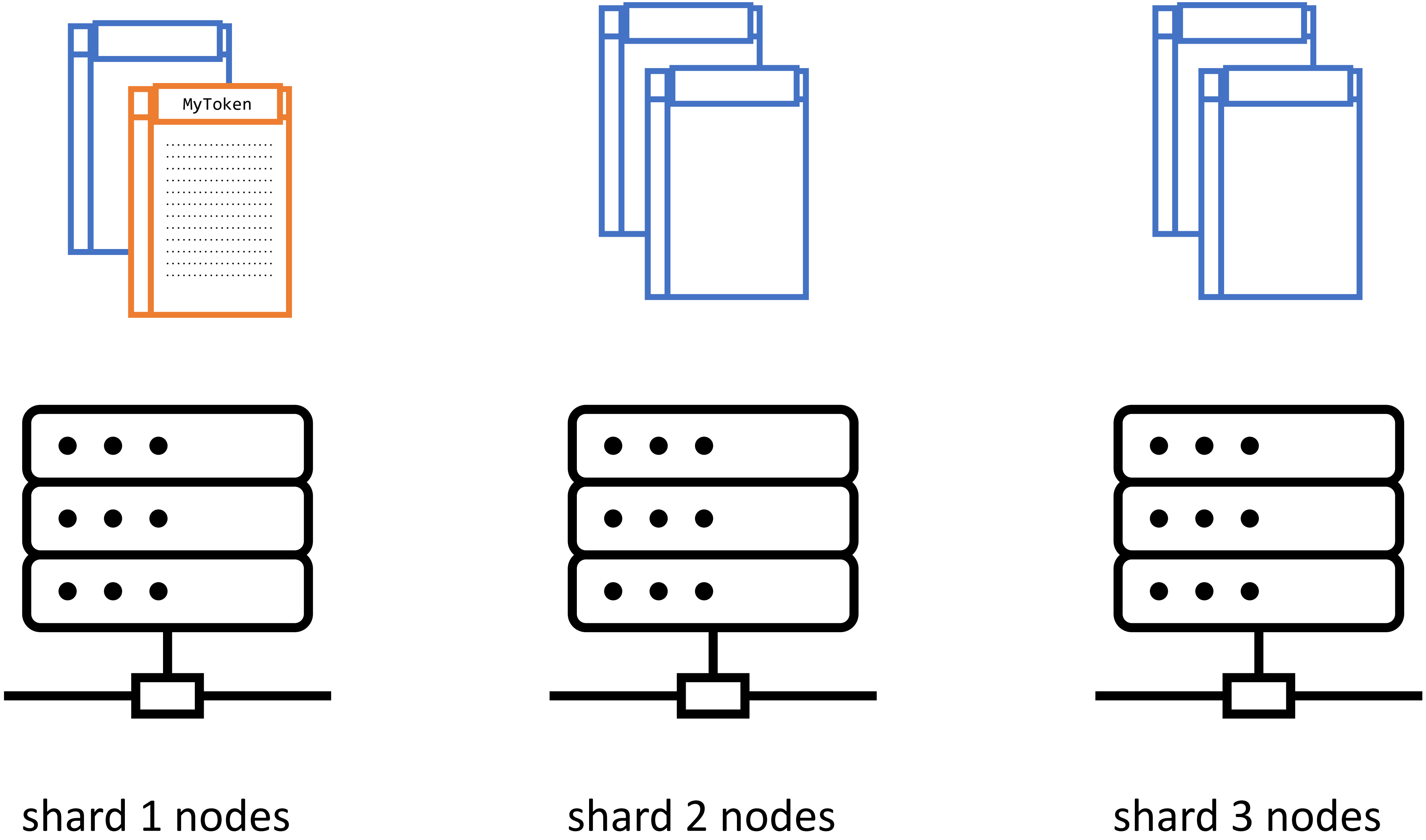

MyToken
shard 1 nodes
shard 2 nodes
shard 3 nodes

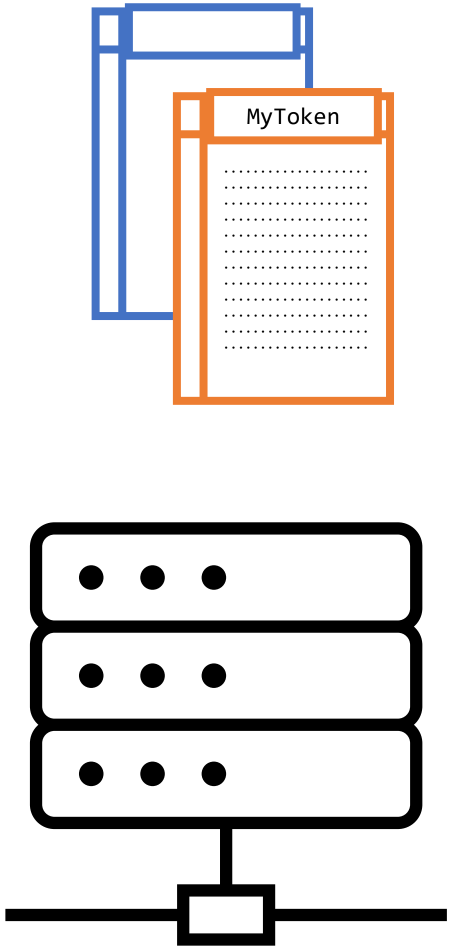

MyToken
shard 1 nodes
shard 2 nodes
shard 3 nodes


MyToken
shard 1 nodes
shard 2 nodes
shard 3 nodes

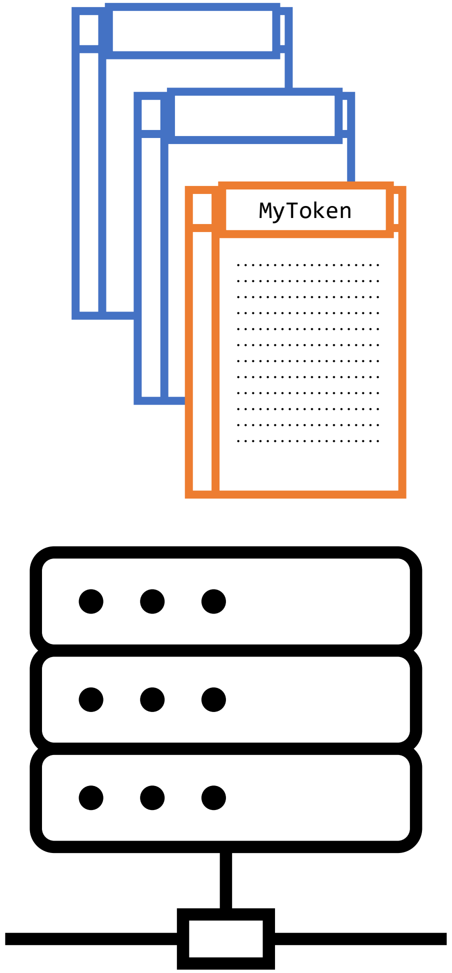


shard 1 nodes shard 2 nodes shard 3 nodes

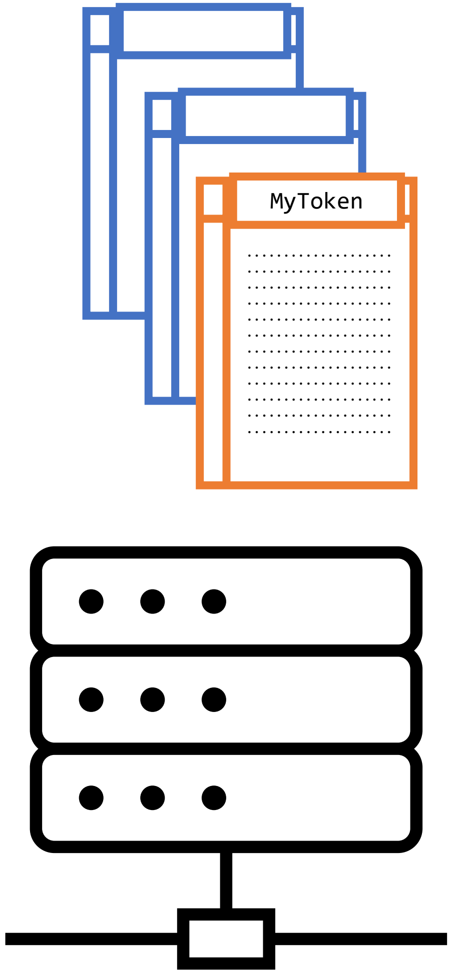

MyToken
shard 1 nodes
shard 2 nodes
shard 3 nodes



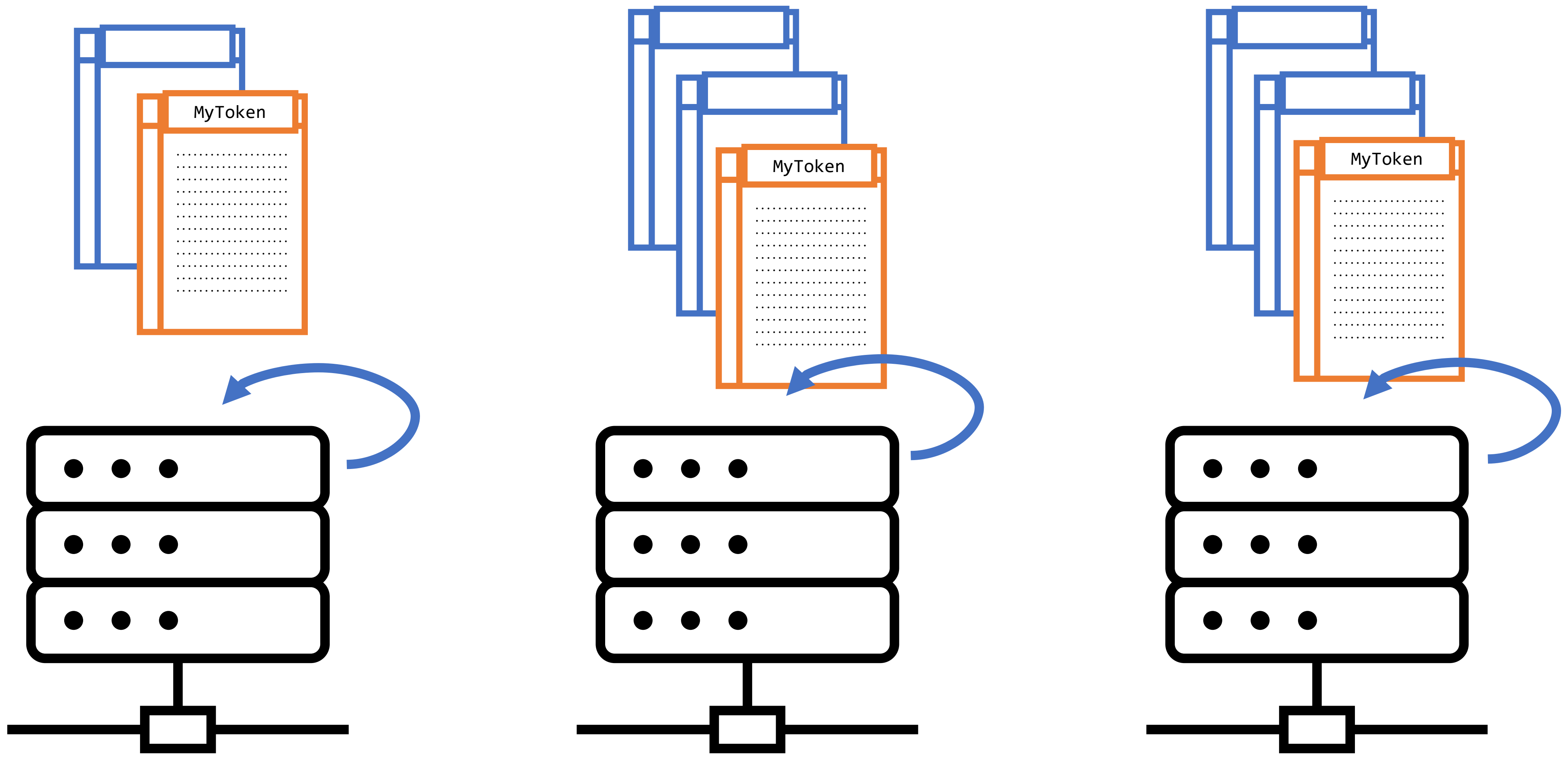
shard 1 nodes



shard 2 nodes



shard 3 nodes



shard 1 nodes shard 2 nodes shard 3 nodes

Look at the contract's code to learn how to shard it!

Smart Contract Sharding

with Static Program Analysis

problem we are working on

Smart Contract Sharding

with Static Program Analysis

problem we are working on

Smart Contract Sharding

with Static Program Analysis

technique we are applying

problem we are working on

Smart Contract Sharding

with Static Program Analysis

Ownership

technique we are applying

problem we are working on

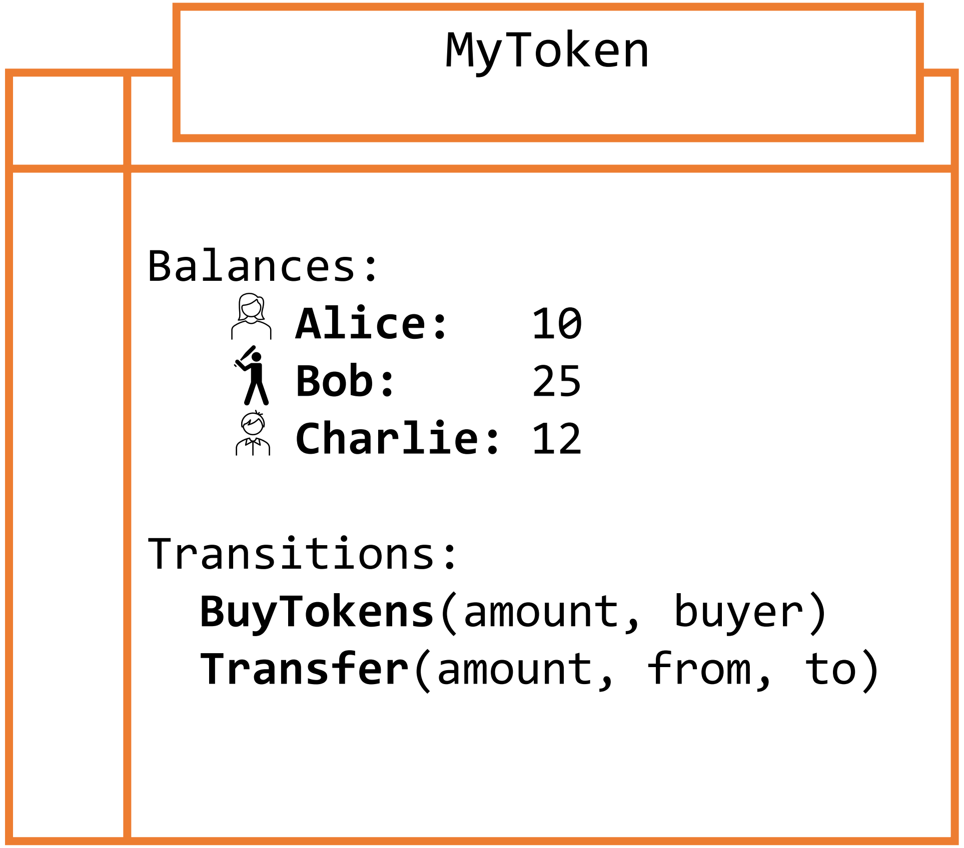
Smart Contract Sharding

with Static Program Analysis

Ownership

Commutativity

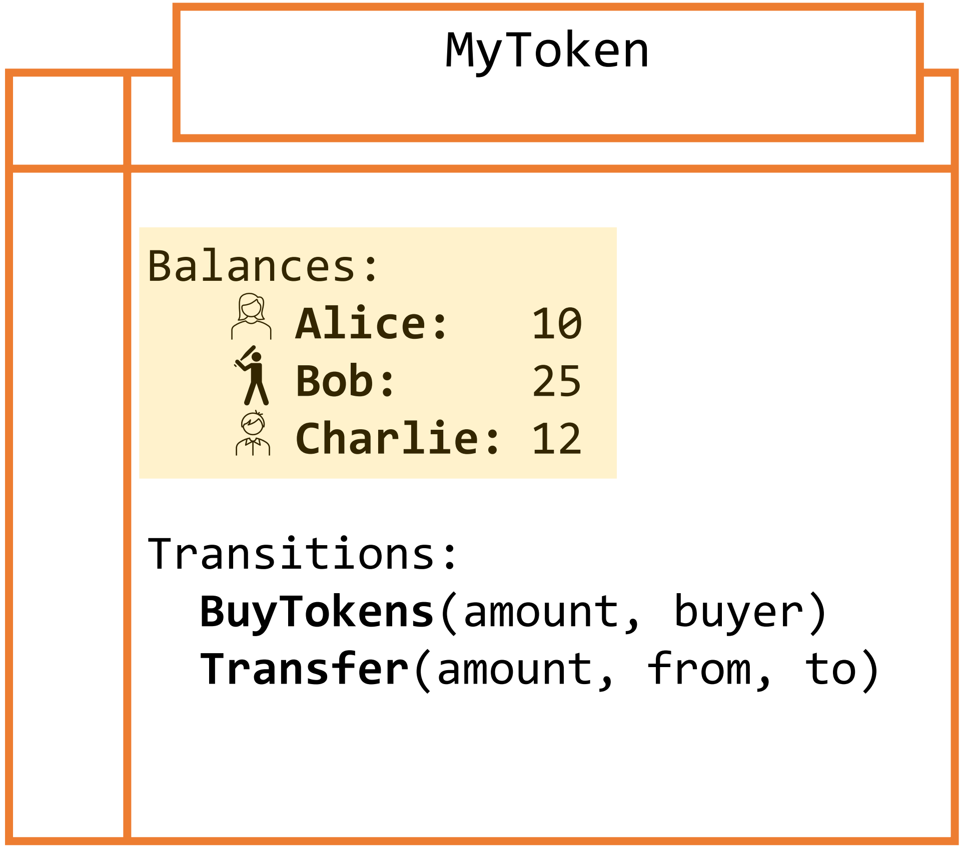
technique we are applying

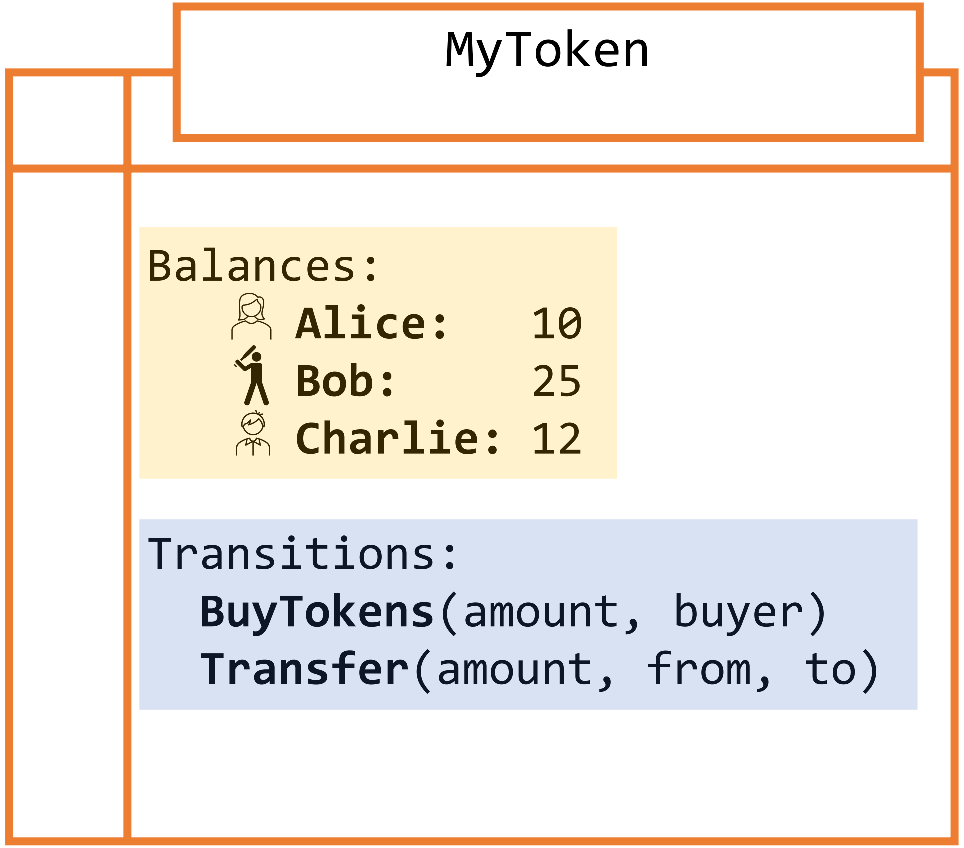

MyToken
Balances:
Alice: 10
Bob: 25
Charlie: 12
Transitions:
BuyTokens(amount, buyer)
Transfer(amount, from, to)

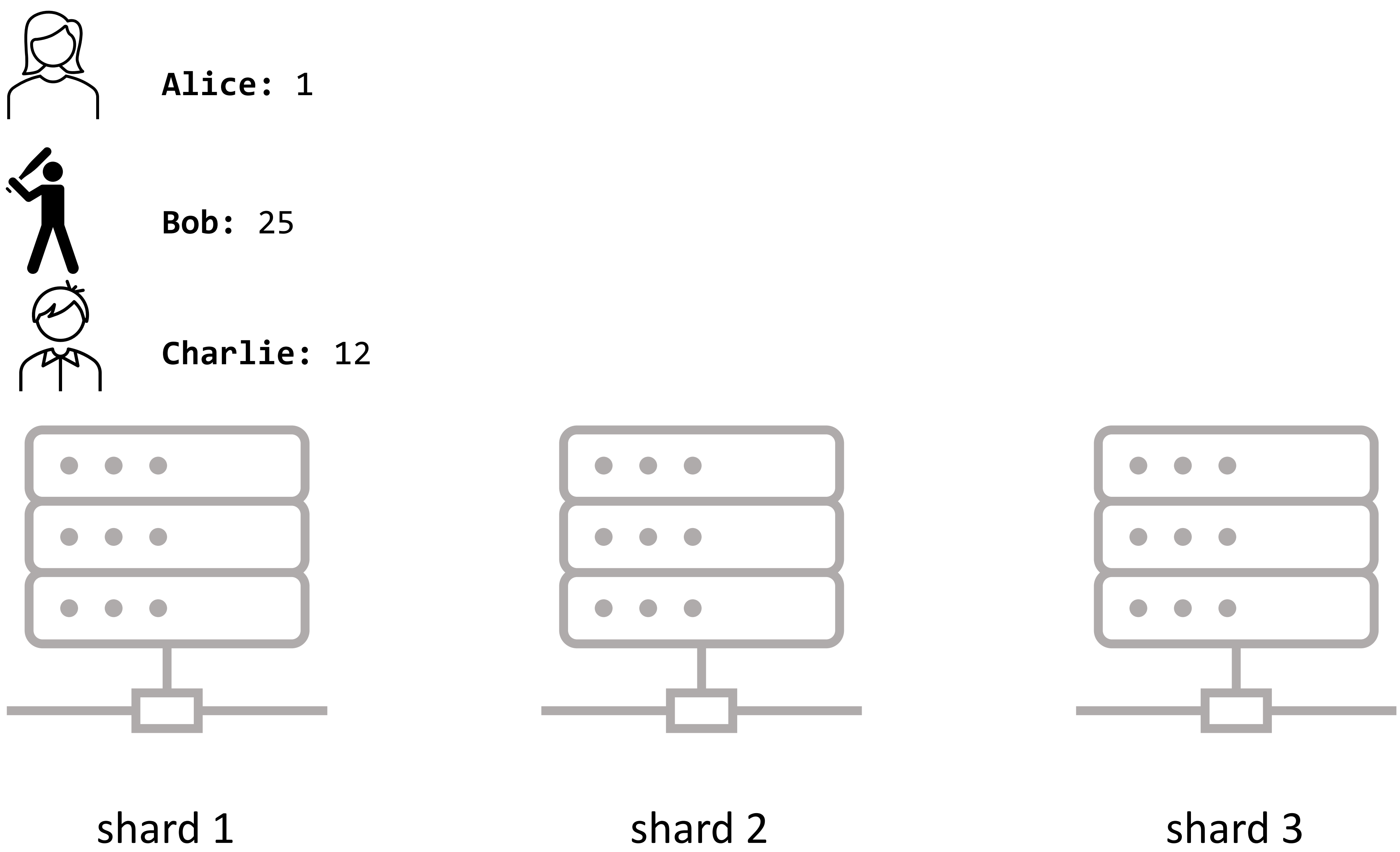
MyToken

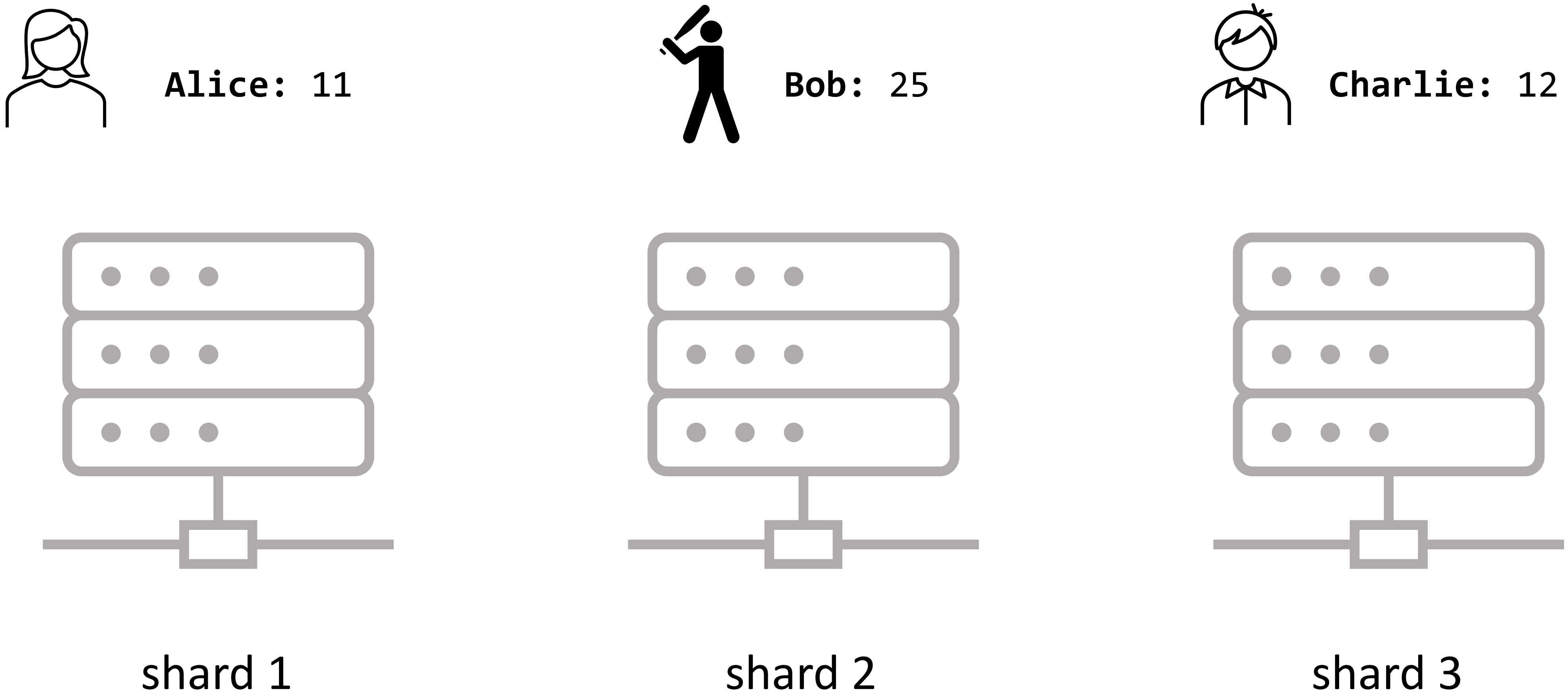
```
Balances:
    Alice:   10
    Bob:     25
    Charlie: 12

Transitions:
  BuyTokens(amount, buyer)
  Transfer(amount, from, to)
```

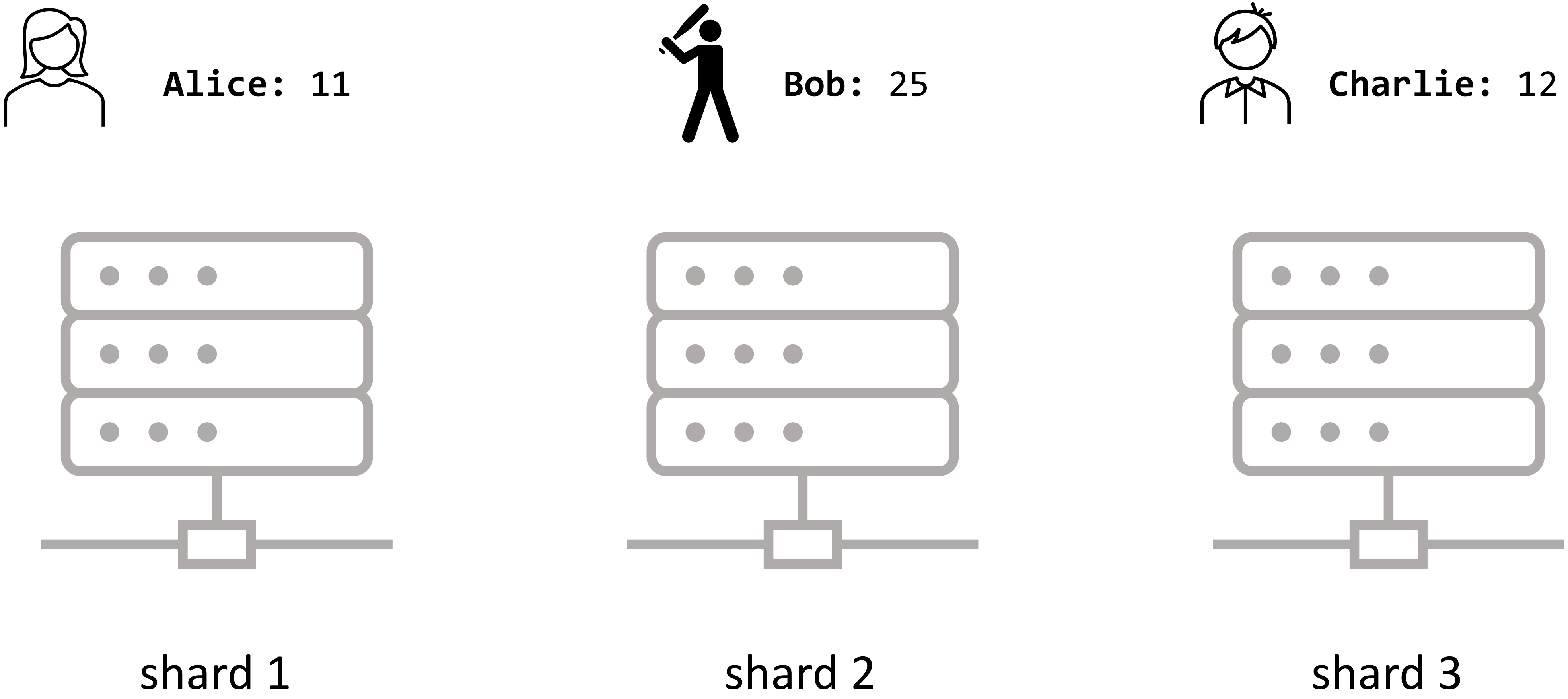

MyToken
Balances:
Alice: 10
Bob: 25
Charlie: 12
Transitions:
BuyTokens(amount, buyer)
Transfer(amount, from, to)


MyToken
Balances:
Alice: 10
Bob: 25
Charlie: 12
Transitions:
BuyTokens(amount, buyer)
Transfer(amount, from, to)

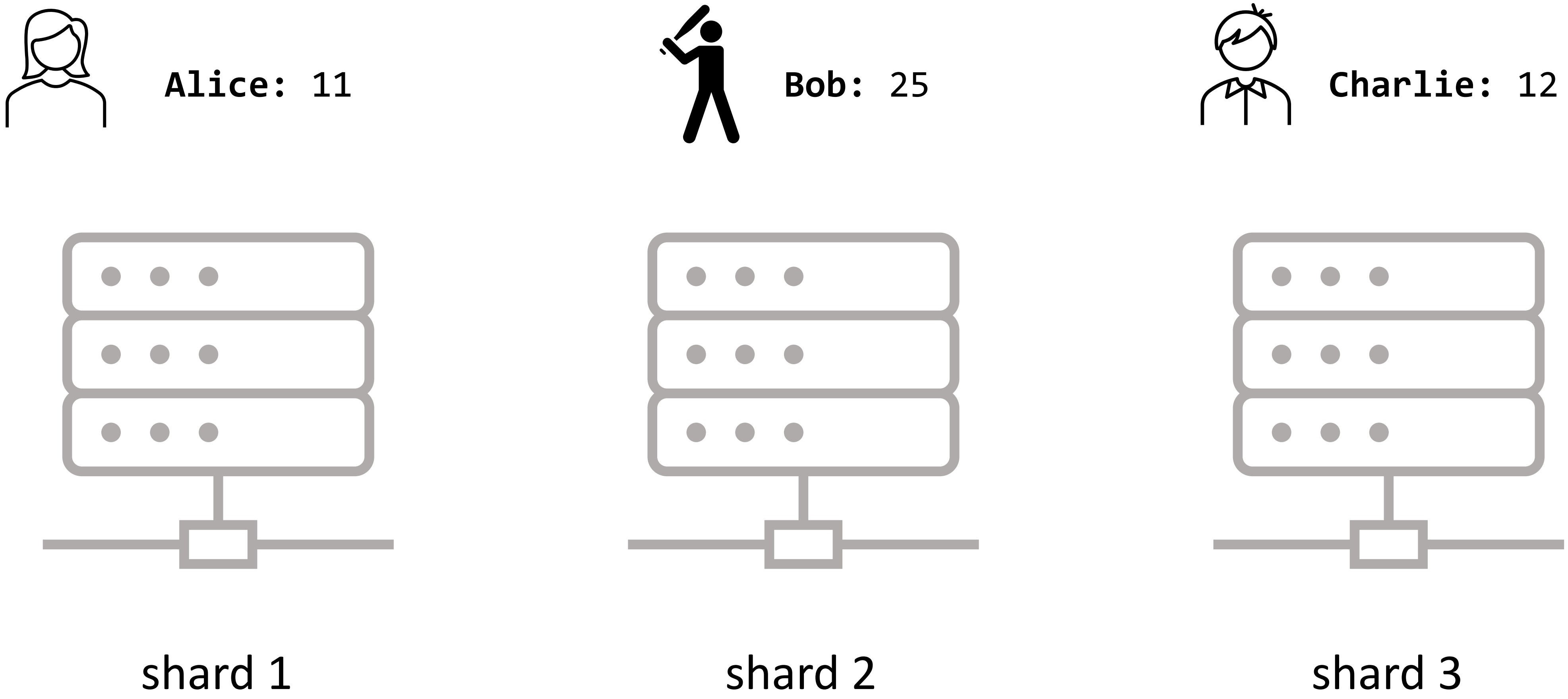

Alice: 1
Bob: 25
Charlie: 12
shard 1
shard 2
shard 3

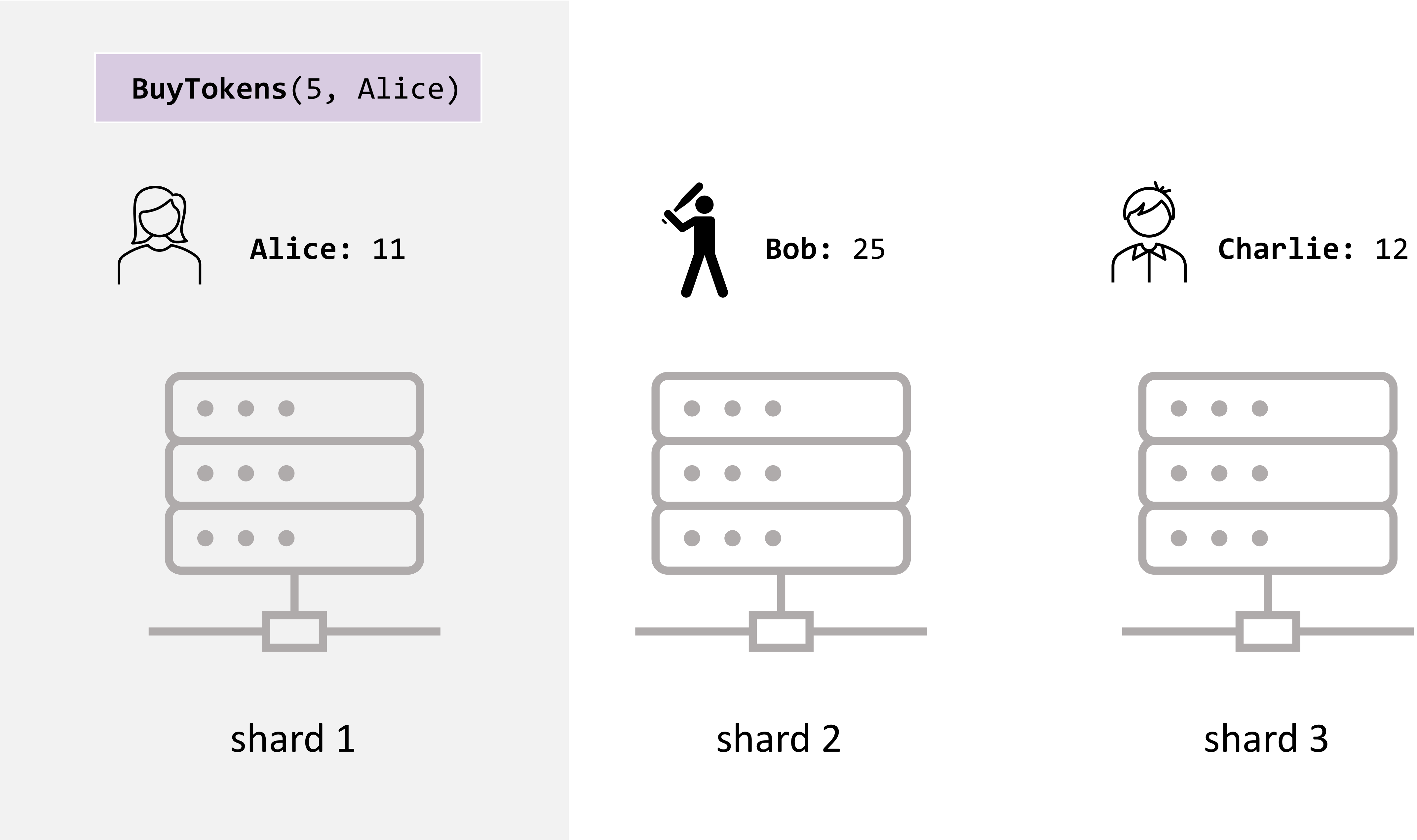

Alice: 11
Bob: 25
Charlie: 12
shard 1
shard 2
shard 3

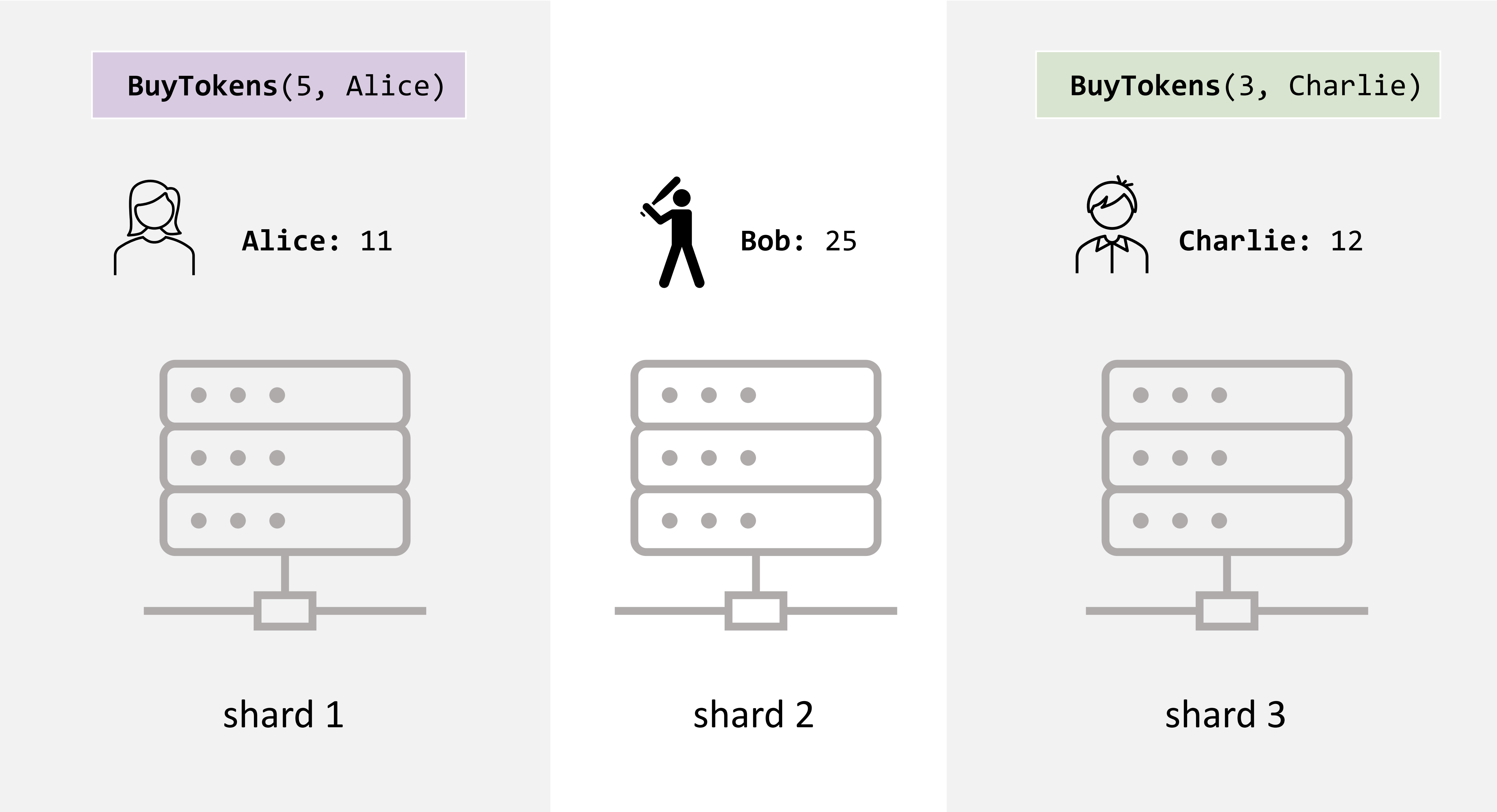
BuyTokens(amount, buyer)



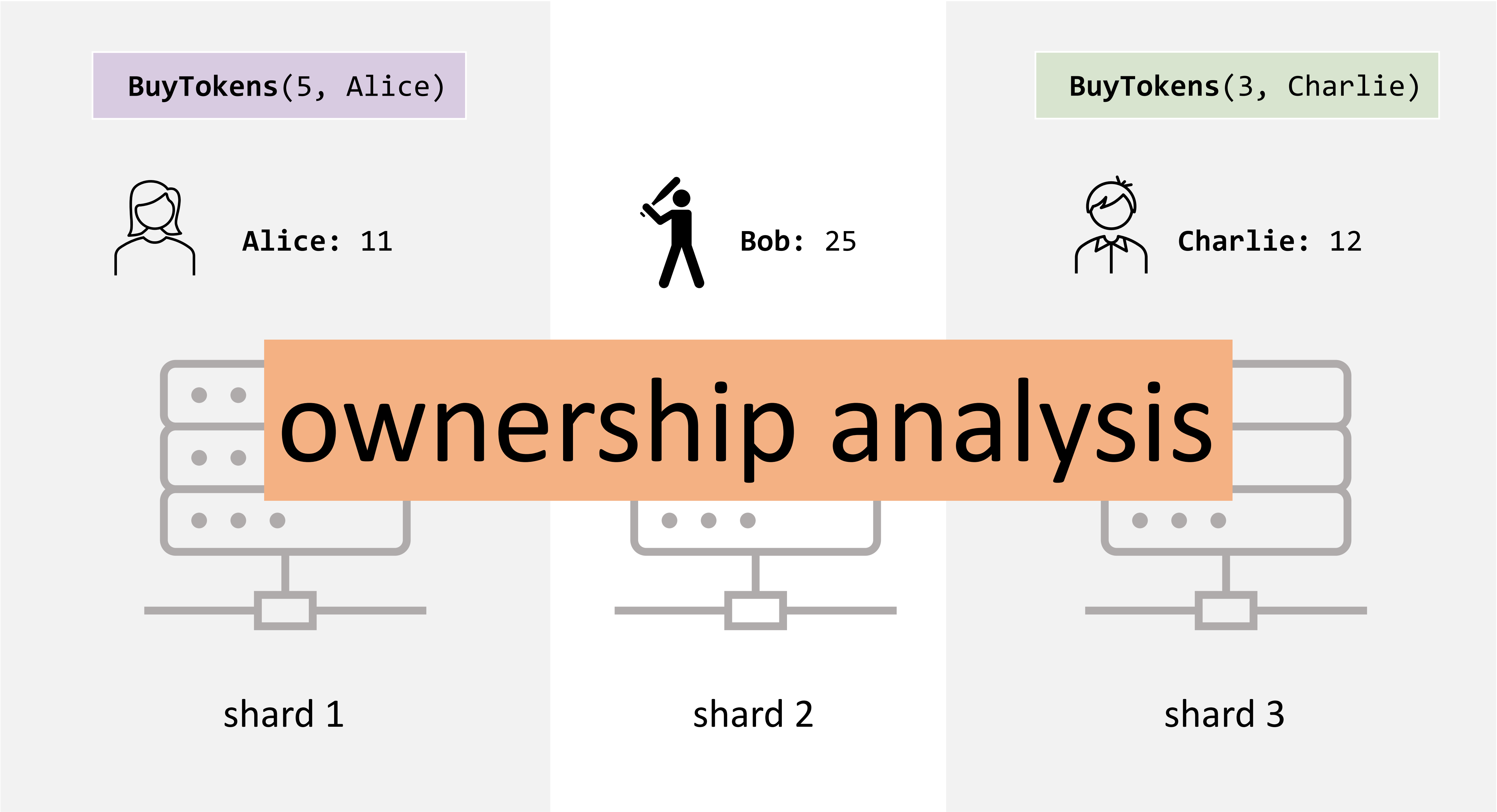
`BuyTokens(5, Alice)`

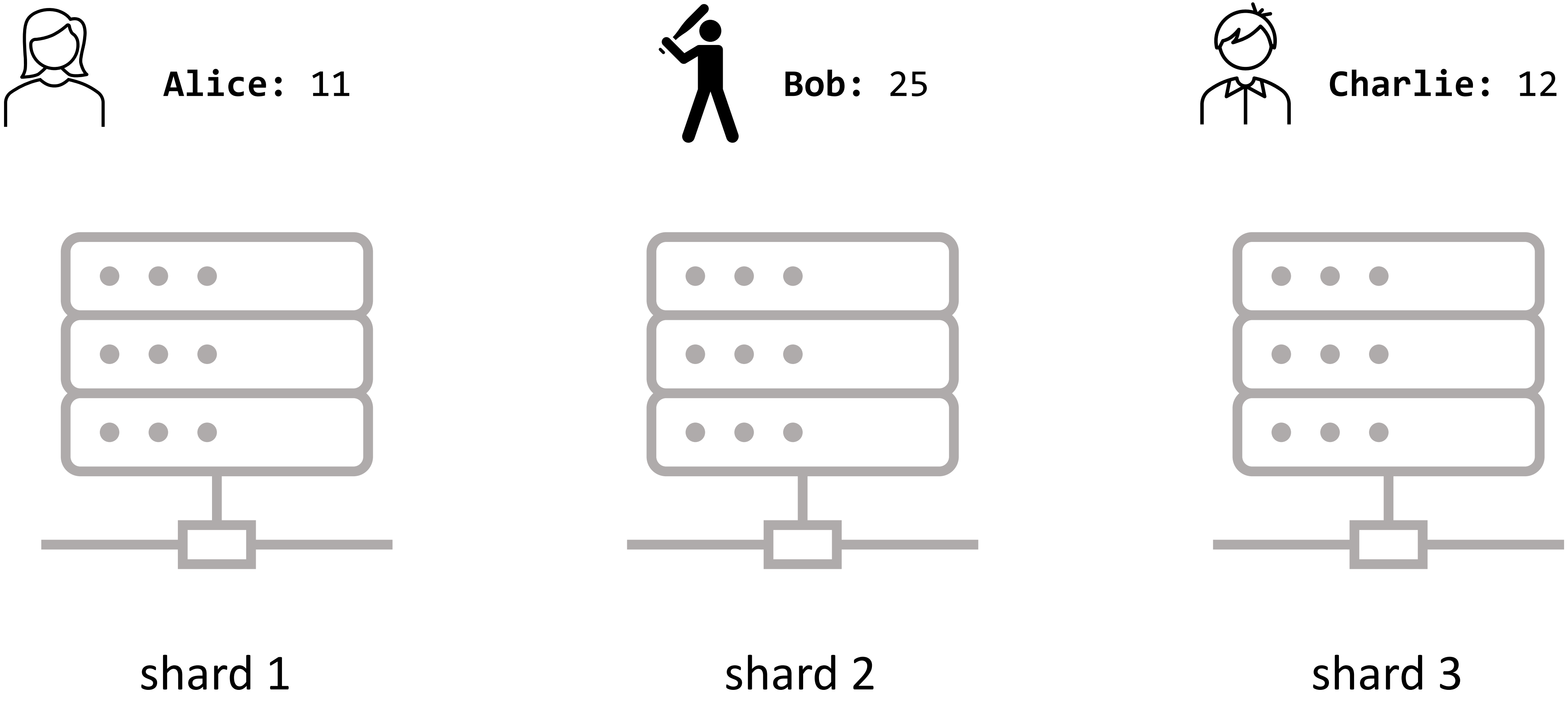


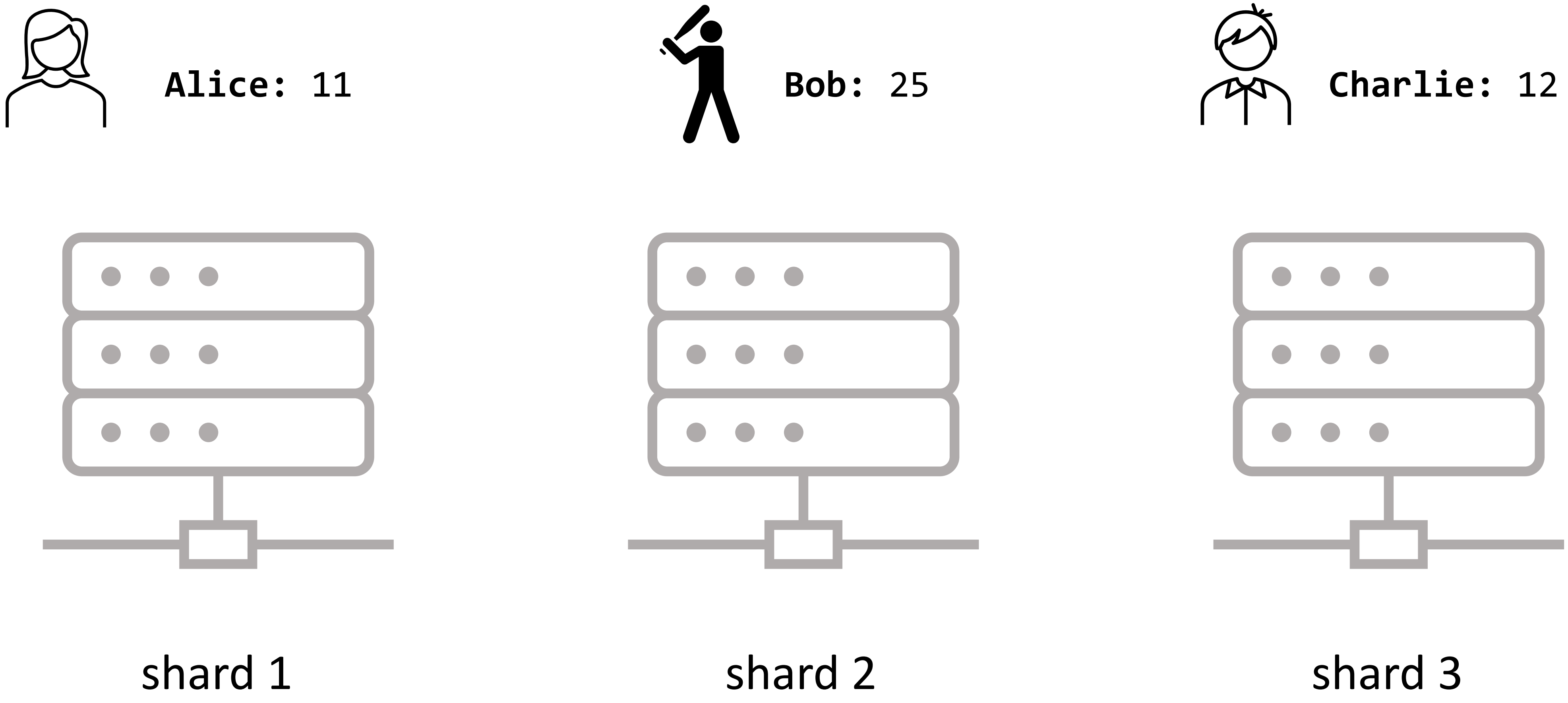

BuyTokens(5, Alice)
Alice: 11
Bob: 25
Charlie: 12
shard 1
shard 2
shard 3

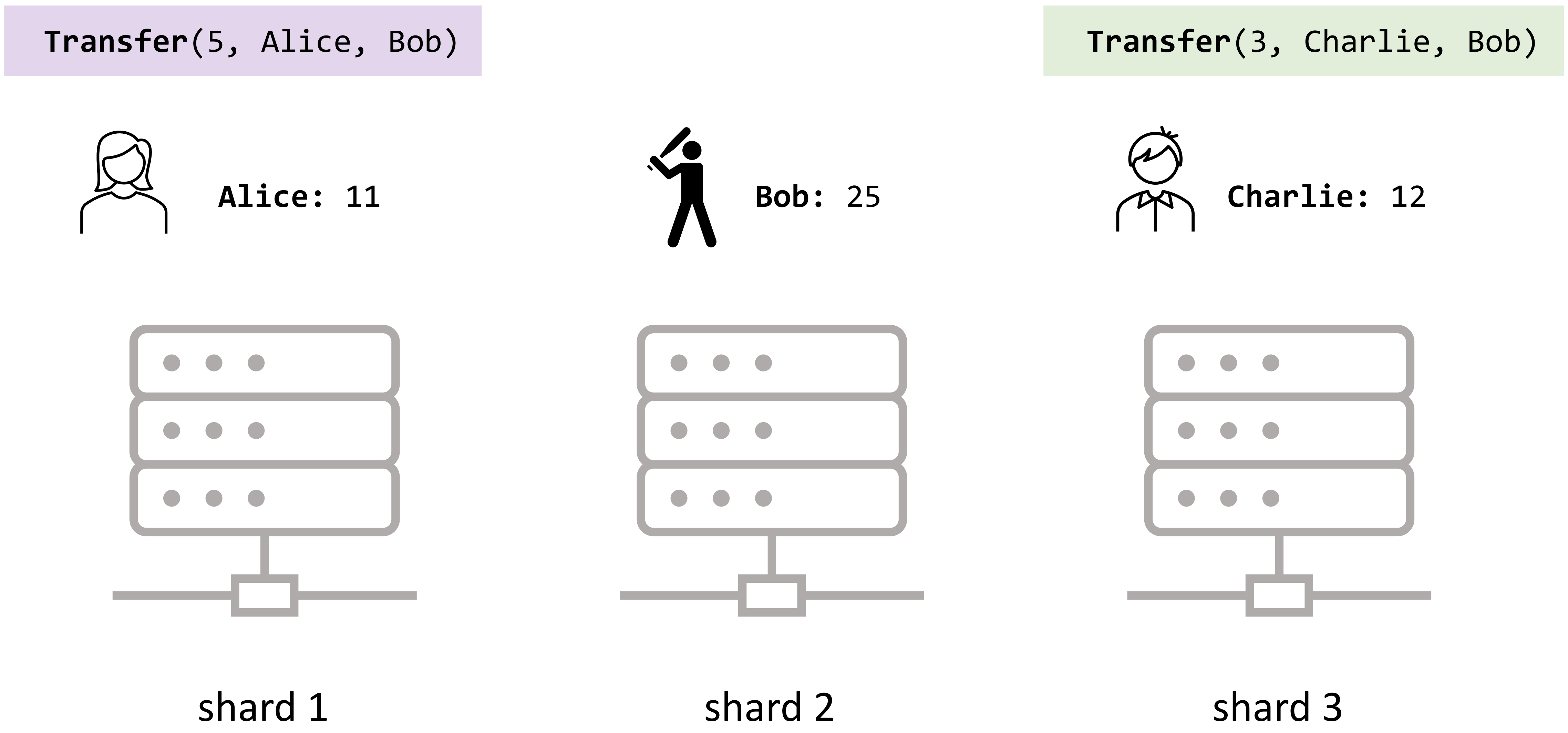


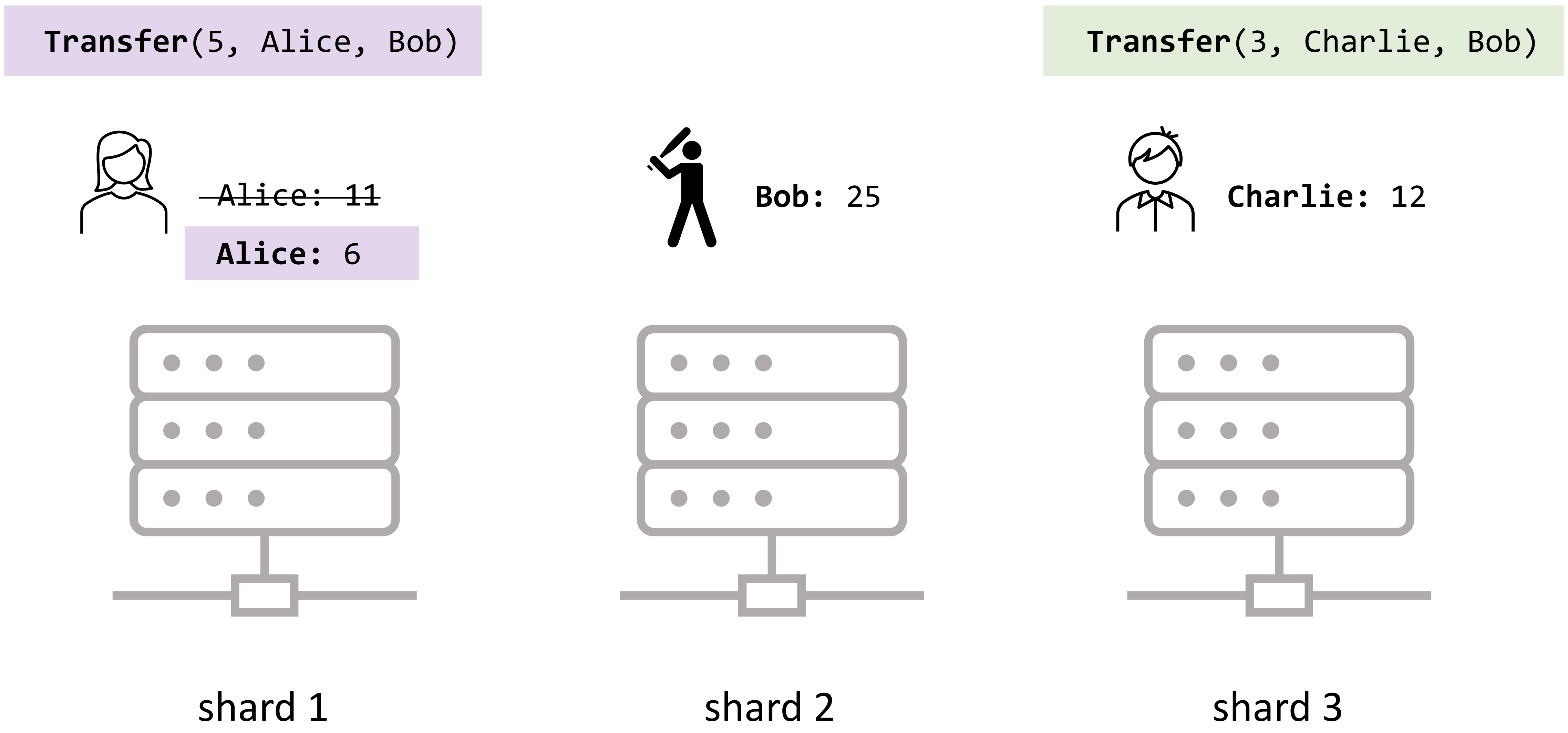
BuyTokens(5, Alice)
Alice: 11
shard 1
Bob: 25
shard 2
BuyTokens(3, Charlie)
Charlie: 12
shard 3

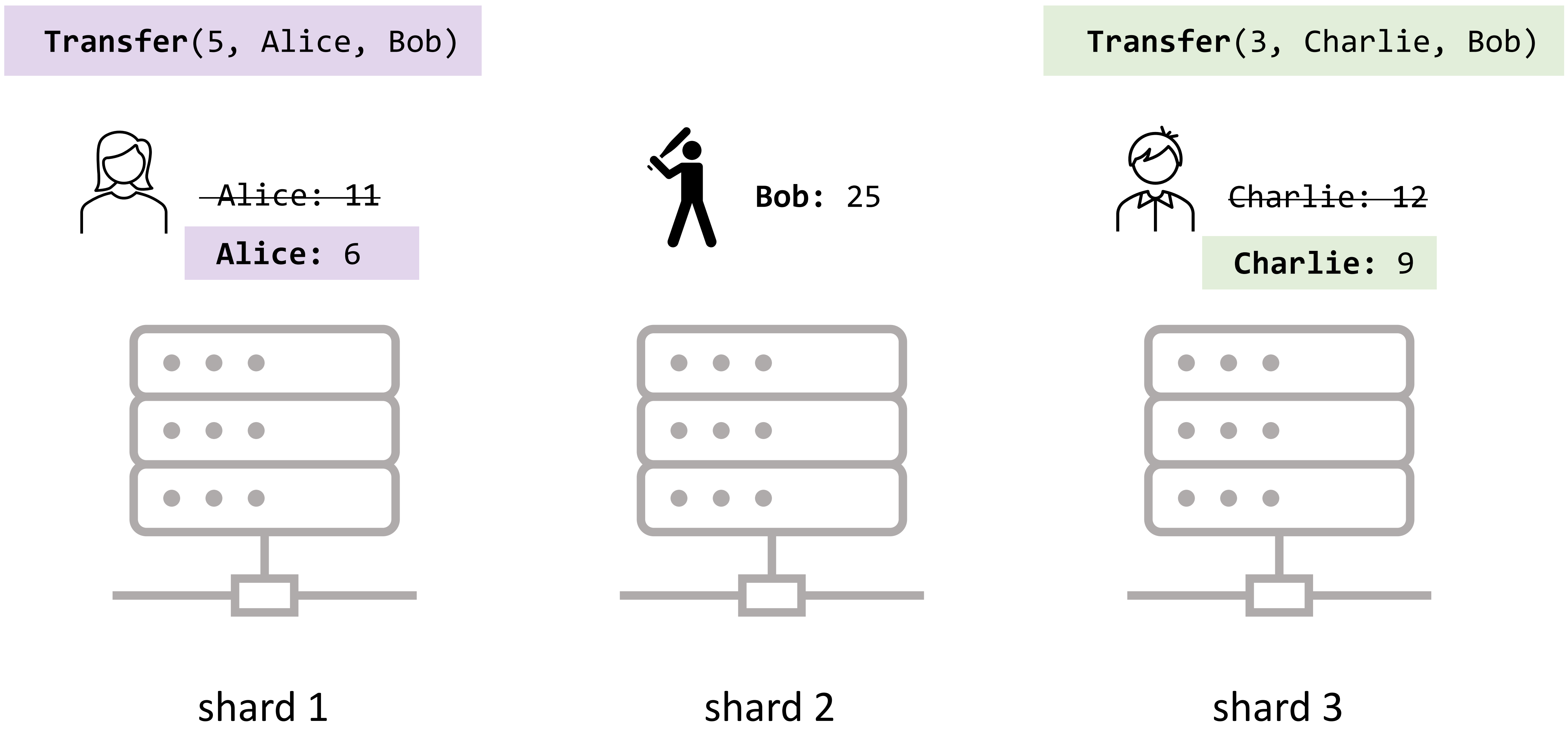

BuyTokens(5, Alice)
BuyTokens(3, Charlie)
Alice: 11
Bob: 25
Charlie: 12
ownership analysis
shard 1
shard 2
shard 3

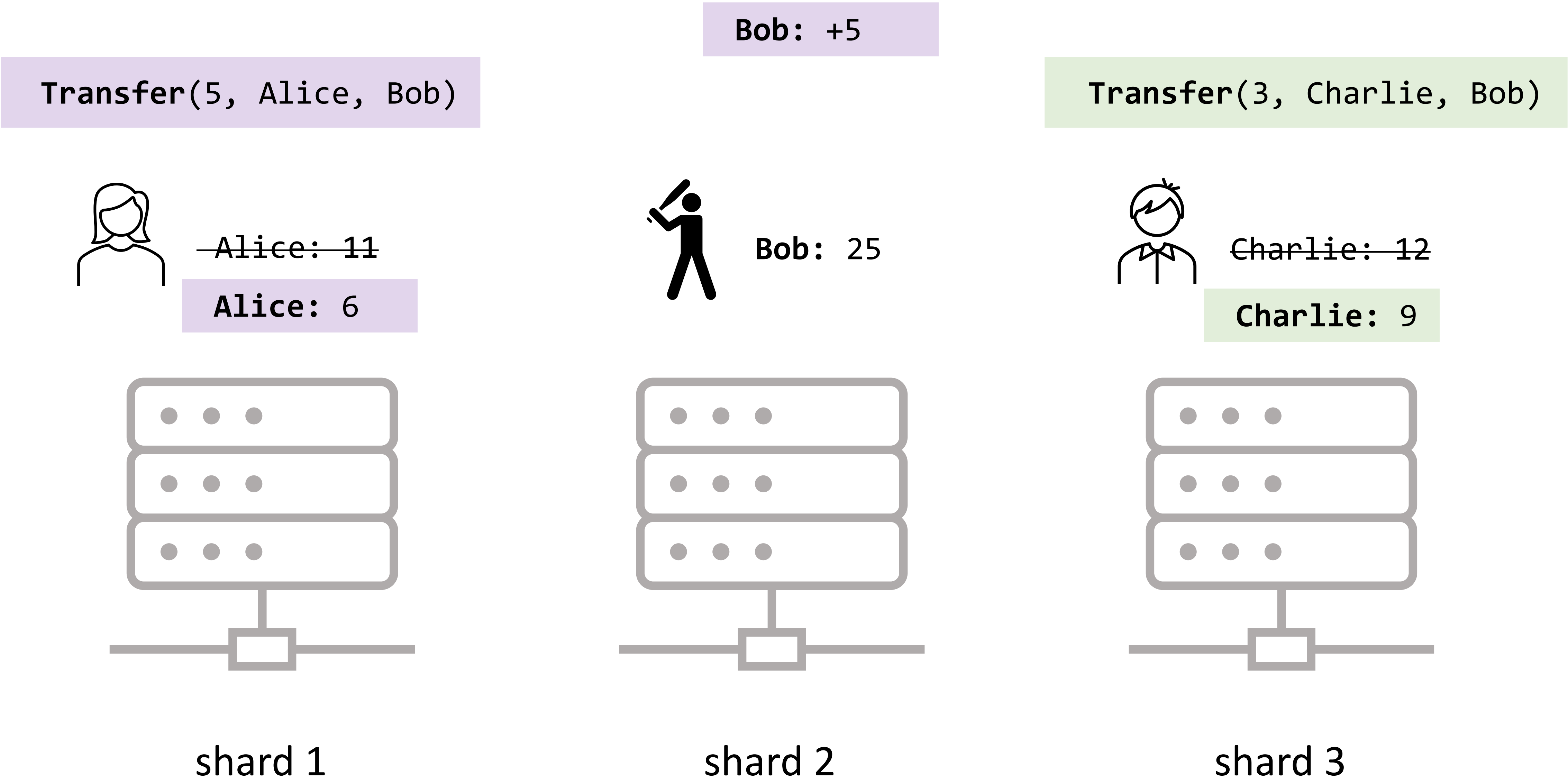

Alice: 11
Bob: 25
Charlie: 12
shard 1
shard 2
shard 3

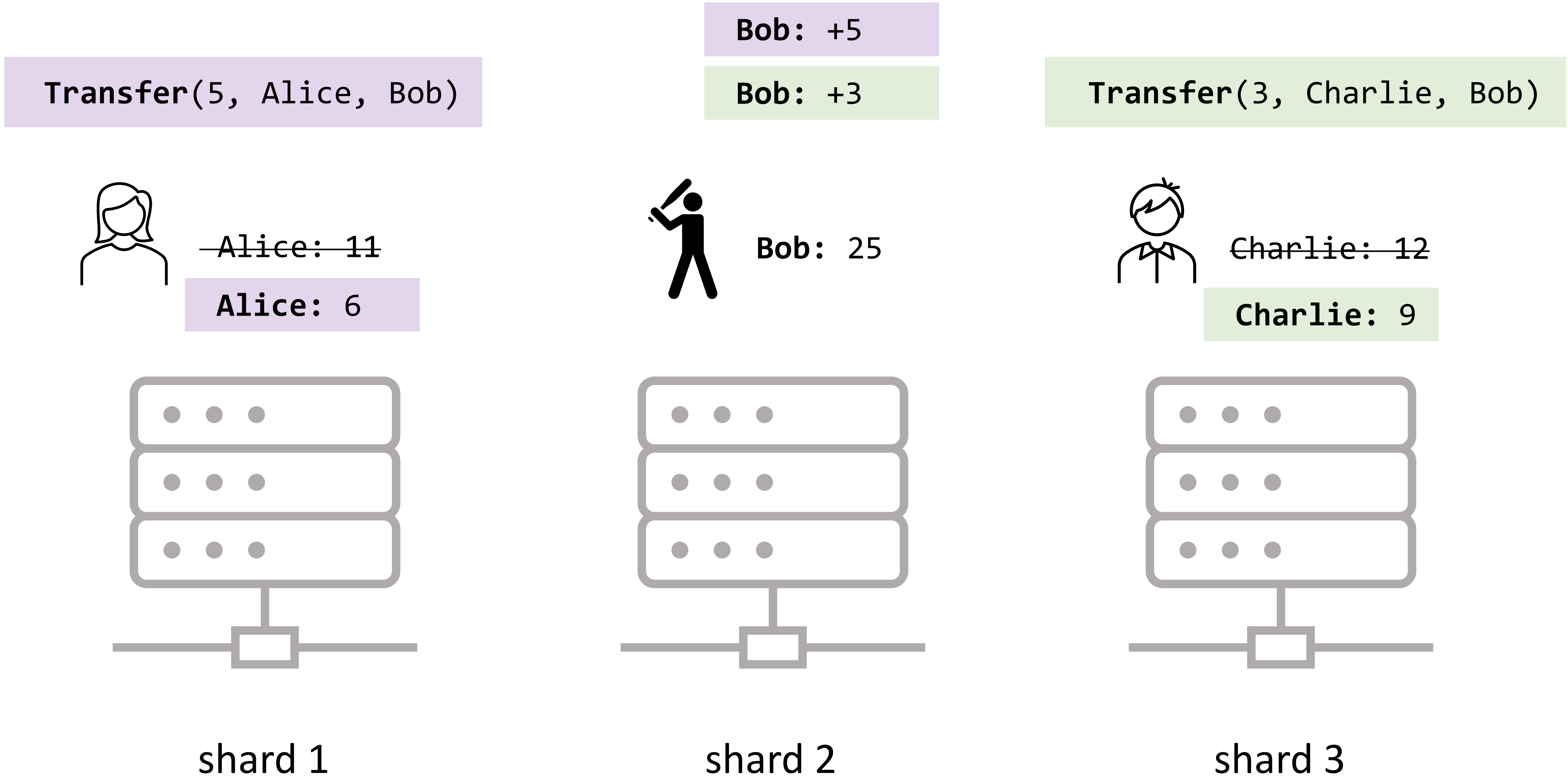

Alice: 11
Bob: 25
Charlie: 12
shard 1
shard 2
shard 3

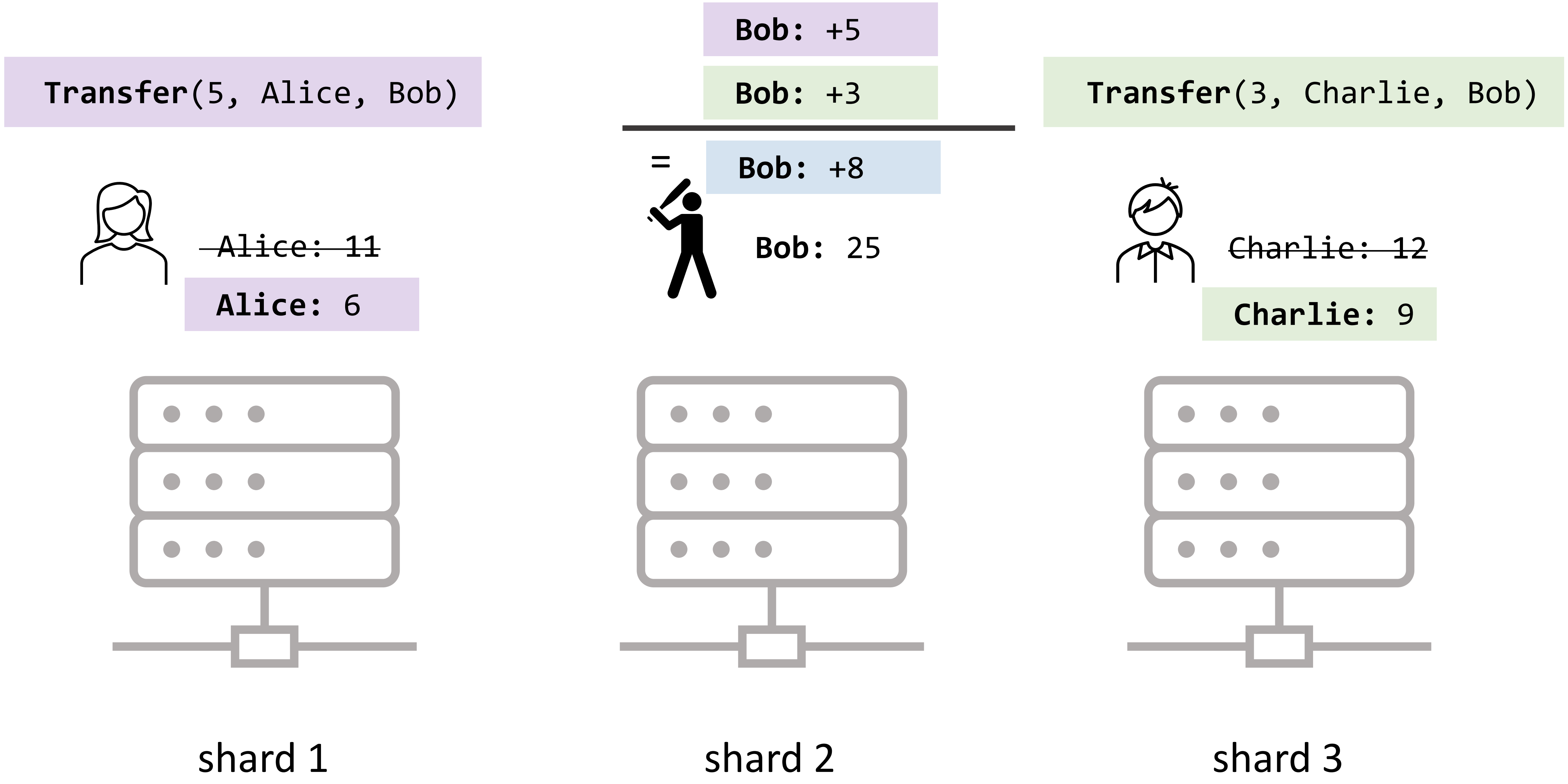

Transfer(5, Alice, Bob)
Transfer(3, Charlie, Bob)
Alice: 11
Bob: 25
Charlie: 12
shard 1
shard 2
shard 3

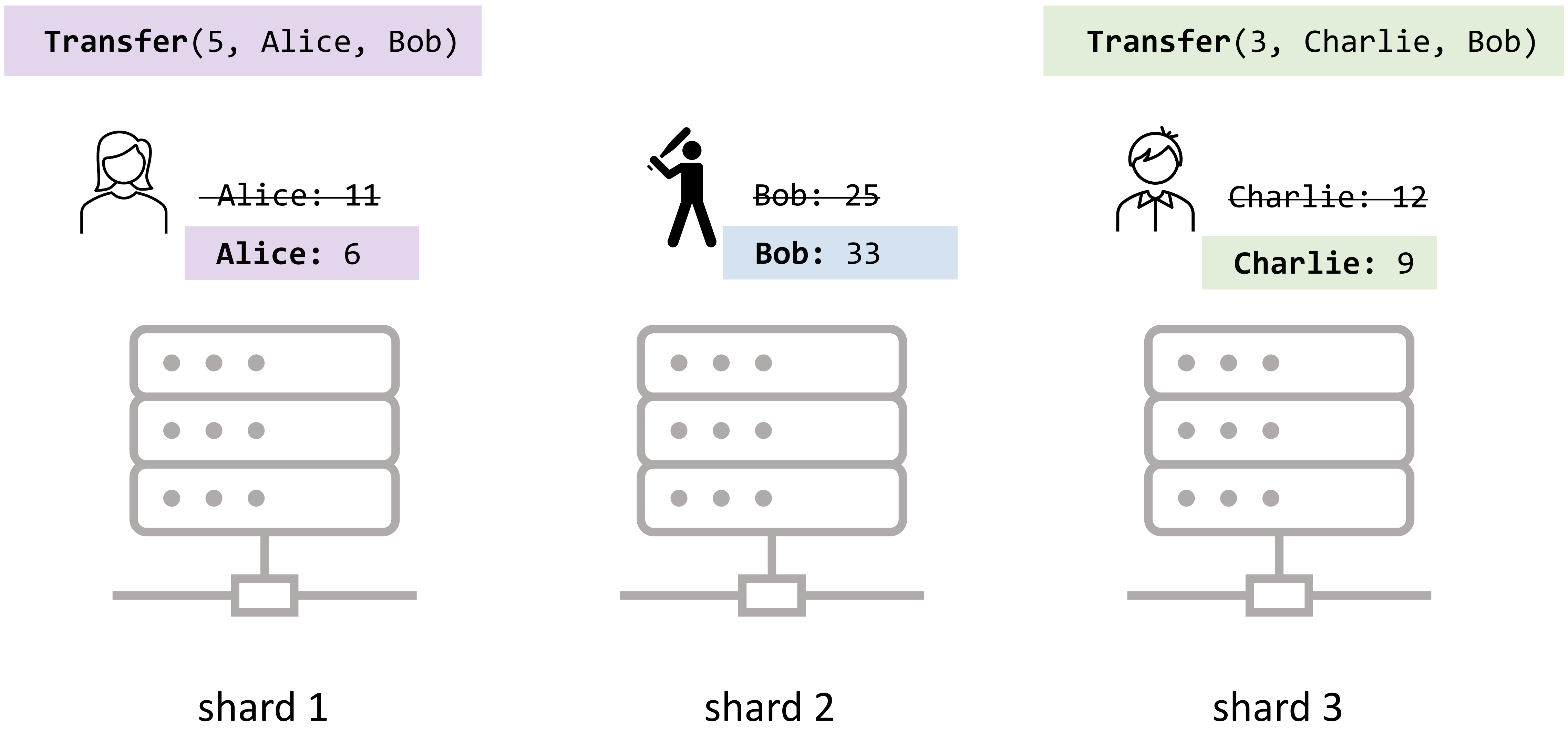

Transfer(5, Alice, Bob)
Transfer(3, Charlie, Bob)
~~Alice: 11~~
Alice: 6
Bob: 25
Charlie: 12
shard 1
shard 2
shard 3

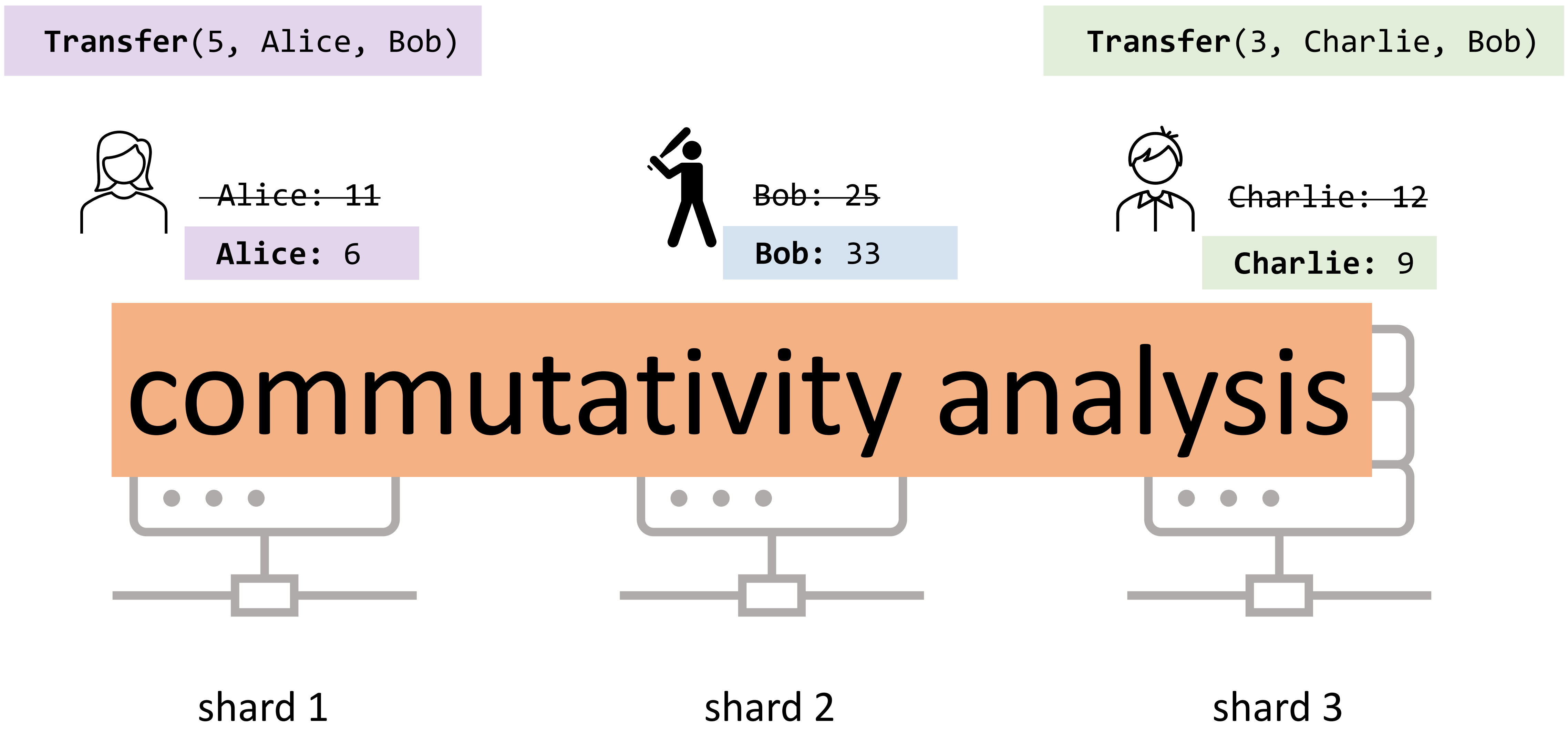

Transfer(5, Alice, Bob)
Transfer(3, Charlie, Bob)
~~Alice: 11~~
Alice: 6
Bob: 25
~~Charlie: 12~~
Charlie: 9
shard 1
shard 2
shard 3


Bob: +5
Transfer(5, Alice, Bob)
Transfer(3, Charlie, Bob)
~~Alice: 11~~
Alice: 6
Bob: 25
~~Charlie: 12~~
Charlie: 9
shard 1
shard 2
shard 3


Transfer(5, Alice, Bob)
~~Alice: 11~~
Alice: 6
Bob: +5
Bob: +3
Bob: 25
Transfer(3, Charlie, Bob)
~~Charlie: 12~~
Charlie: 9
shard 1
shard 2
shard 3


Transfer(5, Alice, Bob)
Bob: +5
Bob: +3
=
Bob: +8
Transfer(3, Charlie, Bob)
~~Alice: 11~~
Alice: 6
Bob: 25
~~Charlie: 12~~
Charlie: 9
shard 1
shard 2
shard 3


Transfer(5, Alice, Bob)
Transfer(3, Charlie, Bob)
~~Alice: 11~~
Alice: 6
~~Bob: 25~~
Bob: 33
~~Charlie: 12~~
Charlie: 9
shard 1
shard 2
shard 3


Transfer(5, Alice, Bob)
Transfer(3, Charlie, Bob)
~~Alice: 11~~
Alice: 6
~~Bob: 25~~
Bob: 33
~~Charlie: 12~~
Charlie: 9
commutativity analysis
shard 1
shard 2
shard 3

CoSplit

Practical Smart Contract Sharding with Static Program Analysis

ZILLIQA + SCILLA

SCILLA

smart contracts
as
communicating automata

Two key features:

- Clearly **separates computation from communication**
  - message-passing rather than method calls for contract interaction
- Strict distinction between **pure and effectful computations**
  - Scilla has a small imperative fragment with conditionals but without loops
  - Only pure (non-effectful) recursion is allowed

static analysis can be quite precise

```
field balances: Map Address Uint
```

```
transition Transfer(to: Address, amount: Uint)
  from_bal <- balances[_sender];
  match from_bal with
  | Some bal =>
    match amount ≤ bal with
    | True =>
      new_from_bal = builtin sub bal amount;
      balances[_sender] := new_from_bal;
      to_bal <- balances[to];
      new_to_bal = match to_bal with
      | Some bal => builtin add bal amount
      | None => amount
      end;
      balances[to] := new_to_bal
```

Static analysis for transition effects

- *Ownership:* produce an **effect summary** for every transition
  - Effects include: reads, writes, accepting funds, sending messages, conditioning on values derived from mutable fields
  - The effect summary *over-approximates* the behaviour of the transition

- *Commutativity:* **linearity-aware flows-to analysis**
  - Inspired by GHC’s cardinality analysis (POPL’14)
  - Expressed as a type system for “contribution types”
    - compositional, but sometimes gives uninformative types

Constant	x, y	constant contract field or transition parameter
Mutable	f	mutable field or map-field access via parameter
Contrib. src.	$cs ::=$	$x \mid \mathsf{f}$
Cardinality	$card ::=$	$\mathsf{None} \mid \mathsf{Linear} \mid \mathsf{NonLinear}$
Operation	$op ::=$	$+ \mid - \mid * \mid \ldots$
Abstr. expr.	$e ::=$	$\top \mid \overline{(cs, card, \overline{op})}$
Effect	$\varepsilon ::=$	$\mathsf{Read}(\mathsf{f}) \mid \mathsf{Write}(\mathsf{f}, e) \mid \mathsf{AcceptFunds} \mid$
		$\mathsf{Condition}(e) \mid \mathsf{Event}(e) \mid \mathsf{SendMsg}(e) \mid \top$

```
transition Transfer(to: Address, amount: Uint)
  from_bal <- balances[_sender];
  match from_bal with
  | Some bal =>
    match amount ≤ bal with
    | True =>
      new_from_bal = builtin sub bal amount;
      balances[_sender] := new_from_bal;
      to_bal <- balances[to];
      new_to_bal = match to_bal with
      | Some bal => builtin add bal amount
      | None => amount
      end;
      balances[to] := new_to_bal
```

```
transition Transfer(to: Address, amount: Uint)
  from_bal <- balances[_sender];                              Read(balances[_sender])
  match from_bal with
  | Some bal =>
    match amount ≤ bal with
    | True =>
      new_from_bal = builtin sub bal amount;
      balances[_sender] := new_from_bal;
      to_bal <- balances[to];
      new_to_bal = match to_bal with
      | Some bal => builtin add bal amount
      | None => amount
      end;
      balances[to] := new_to_bal
```

```
transition Transfer(to: Address, amount: Uint)
  from_bal <- balances[_sender];
  match from_bal with
  | Some bal =>
    match amount ≤ bal with
    | True =>
      new_from_bal = builtin sub bal amount;
      balances[_sender] := new_from_bal;
      to_bal <- balances[to];
      new_to_bal = match to_bal with
      | Some bal => builtin add bal amount
      | None => amount
      end;
      balances[to] := new_to_bal
```

Read(balances[_sender])

Condition(balances[_sender])

```
transition Transfer(to: Address, amount: Uint)
  from_bal <- balances[_sender];
  match from_bal with
  | Some bal =>
    match amount ≤ bal with
    | True =>
      new_from_bal = builtin sub bal amount;
      balances[_sender] := new_from_bal;
      to_bal <- balances[to];
      new_to_bal = match to_bal with
      | Some bal => builtin add bal amount
      | None => amount
      end;
      balances[to] := new_to_bal
```

Read(balances[_sender])

Condition(balances[_sender])

(balances[_sender], Linear, Ø)

```
transition Transfer(to: Address, amount: Uint)
  from_bal <- balances[_sender];
  match from_bal with
  | Some bal =>
    match amount ≤ bal with
    | True =>
      new_from_bal = builtin sub bal amount;
      balances[_sender] := new_from_bal;
      to_bal <- balances[to];
      new_to_bal = match to_bal with
      | Some bal => builtin add bal amount
      | None => amount
      end;
      balances[to] := new_to_bal
```

Read(balances[_sender])

Condition(balances[_sender])

(balances[_sender], Linear, Ø)

Condition(balances[_sender], amount)

```
transition Transfer(to: Address, amount: Uint)
  from_bal <- balances[_sender];                    Read(balances[_sender])
  match from_bal with                               Condition(balances[_sender])
  | Some bal =>                                     (balances[_sender], Linear, Ø)
    match amount ≤ bal with                         Condition(balances[_sender], amount)
    | True =>                                       {(balances[_sender], Linear, sub),
      new_from_bal = builtin sub bal amount;        (amount, Linear, sub)}
      balances[_sender] := new_from_bal;
      to_bal <- balances[to];
      new_to_bal = match to_bal with
      | Some bal => builtin add bal amount
      | None => amount
      end;
      balances[to] := new_to_bal
```

```
transition Transfer(to: Address, amount: Uint)
  from_bal <- balances[_sender];
  match from_bal with
  | Some bal =>
    match amount ≤ bal with
    | True =>
      new_from_bal = builtin sub bal amount;
      balances[_sender] := new_from_bal;
      to_bal <- balances[to];
      new_to_bal = match to_bal with
      | Some bal => builtin add bal amount
      | None => amount
      end;
      balances[to] := new_to_bal
```

Read(balances[_sender])

Condition(balances[_sender])

(balances[_sender], Linear, Ø)

Condition(balances[_sender], amount)

{(balances[_sender], Linear, sub),
(amount, Linear, sub)}

Write(balances[_sender],
{(balances[_sender], Linear, sub),
(amount, Linear, sub)})

```
transition Transfer(to: Address, amount: Uint)
  from_bal <- balances[_sender];
  match from_bal with
  | Some bal =>
    match amount ≤ bal with
    | True =>
      new_from_bal = builtin sub bal amount;
      balances[_sender] := new_from_bal;
      to_bal <- balances[to];
      new_to_bal = match to_bal with
      | Some bal => builtin add bal amount
      | None => amount
      end;
      balances[to] := new_to_bal
```

Read(balances[_sender])

Condition(balances[_sender])

(balances[_sender], Linear, Ø)

Condition(balances[_sender], amount)

{(balances[_sender], Linear, sub),
(amount, Linear, sub)}

Write(balances[_sender],
{(balances[_sender], Linear, sub),
(amount, Linear, sub)})

Read(balances[to])

```
transition Transfer(to: Address, amount: Uint)
  from_bal <- balances[_sender];
  match from_bal with
  | Some bal =>
    match amount ≤ bal with
    | True =>
      new_from_bal = builtin sub bal amount;
      balances[_sender] := new_from_bal;
      to_bal <- balances[to];
      new_to_bal = match to_bal with
      | Some bal => builtin add bal amount
      | None => amount
      end;
      balances[to] := new_to_bal
```

Read(balances[_sender])

Condition(balances[_sender])

(balances[_sender], Linear, Ø)

Condition(balances[_sender], amount)

{(balances[_sender], Linear, sub),
(amount, Linear, sub)}

Write(balances[_sender],
{(balances[_sender], Linear, sub),
(amount, Linear, sub)})

Read(balances[to])

Write(balances[to],
{(balances[to], Linear, add),
(amount, Linear, add)})

$$\text{Constraint} \qquad oc ::= \mathsf{Owns}(\mathsf{f}) \mid \mathsf{UserAddr}(x) \mid \mathsf{NoAliases}(\langle x, y \rangle) \mid \mathsf{SenderShard} \mid \mathsf{ContractShard} \mid \bot$$

Constraint	oc ::=	$\mathsf{Owns}(\mathsf{f}) \mid \mathsf{UserAddr}(x) \mid \mathsf{NoAliases}(\langle x, y \rangle) \mid$ $\mathsf{SenderShard} \mid \mathsf{ContractShard} \mid \perp$
Join	$\uplus_{\mathsf{f}}$ ::=	$\mathsf{OwnOverwrite} \mid \mathsf{IntMerge}$

Read(balances[_sender])

Condition(balances[_sender])

Condition(balances[_sender], amount)

Write(balances[_sender],
{(balances[_sender], Linear, sub),
(amount, Linear, sub)})

Read(balances[to])

Write(balances[to],
{(balances[to], Linear, add),
(amount, Linear, add)})

Constraint $oc ::= \mathsf{Owns}(\mathsf{f}) \mid \mathsf{UserAddr}(x) \mid \mathsf{NoAliases}(\langle x, y \rangle) \mid \mathsf{SenderShard} \mid \mathsf{ContractShard} \mid \perp$

Join $\uplus_{\mathsf{f}} ::= \mathsf{OwnOverwrite} \mid \mathsf{IntMerge}$

Owns(balances[_sender])

Read(balances[_sender])

Condition(balances[_sender])

Condition(balances[_sender], amount)

Write(balances[_sender],
{(balances[_sender], Linear, sub),
(amount, Linear, sub)})

Read(balances[to])

Write(balances[to],
{(balances[to], Linear, add),
(amount, Linear, add)})

Constraint $\quad oc ::= \mathsf{Owns}(\mathsf{f}) \mid \mathsf{UserAddr}(x) \mid \mathsf{NoAliases}(\langle x,y \rangle) \mid \mathsf{SenderShard} \mid \mathsf{ContractShard} \mid \bot$

Join $\quad \uplus_{\mathsf{f}} ::= \mathsf{OwnOverwrite} \mid \mathsf{IntMerge}$

Owns(balances[_sender])

NoAliases(<_sender, to>)

Read(balances[_sender])

Condition(balances[_sender])

Condition(balances[_sender], amount)

Write(balances[_sender],
{(balances[_sender], Linear, sub),
(amount, Linear, sub)})

Read(balances[to])

Write(balances[to],
{(balances[to], Linear, add),
(amount, Linear, add)})

Constraint $oc ::= \mathsf{Owns}(\mathsf{f}) \mid \mathsf{UserAddr}(x) \mid \mathsf{NoAliases}(\langle x, y \rangle) \mid \mathsf{SenderShard} \mid \mathsf{ContractShard} \mid \bot$

Join $\uplus_{\mathsf{f}} ::= \mathsf{OwnOverwrite} \mid \mathsf{IntMerge}$

Owns(balances[_sender])

NoAliases(<_sender, to>)

OwnOverwrite join for owned contributions
IntMerge join for un-owned contributions

Read(balances[_sender])

Condition(balances[_sender])

Condition(balances[_sender], amount)

Write(balances[_sender],
{(balances[_sender], Linear, sub),
(amount, Linear, sub)})

Read(balances[to])

Write(balances[to],
{(balances[to], Linear, add),
(amount, Linear, add)})

Constraint $oc ::= \mathsf{Owns}(\mathsf{f}) \mid \mathsf{UserAddr}(x) \mid \mathsf{NoAliases}(\langle x, y \rangle) \mid \mathsf{SenderShard} \mid \mathsf{ContractShard} \mid \bot$

Join $\uplus_{\mathsf{f}} ::= \mathsf{OwnOverwrite} \mid \mathsf{IntMerge}$

Read(balances[_sender])

Condition(balances[_sender])

Weak reads

Condition(balances[_sender], amount)

Write(balances[_sender],
{(balances[_sender], Linear, sub),
(amount, Linear, sub)})

Owns(balances[_sender])

NoAliases(<_sender, to>)

OwnOverwrite join for owned contributions
IntMerge join for un-owned contributions

Read(balances[to])

Write(balances[to],
{(balances[to], Linear, add),
(amount, Linear, add)})

Integrating **CoSplit** with a sharded blockchain

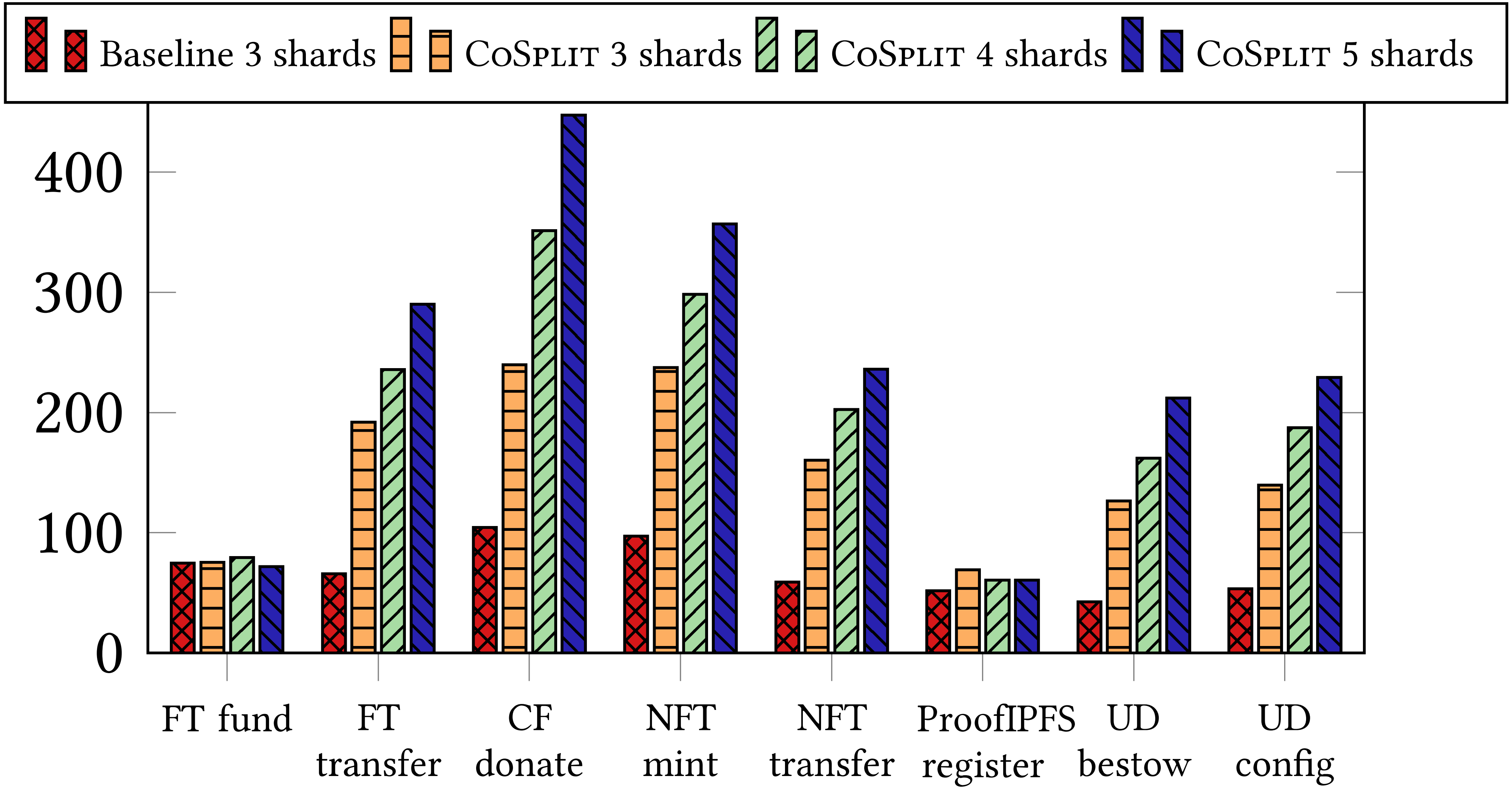
1. Run the static analysis when the contract is first deployed
2. Store the resulting *sharding signature*

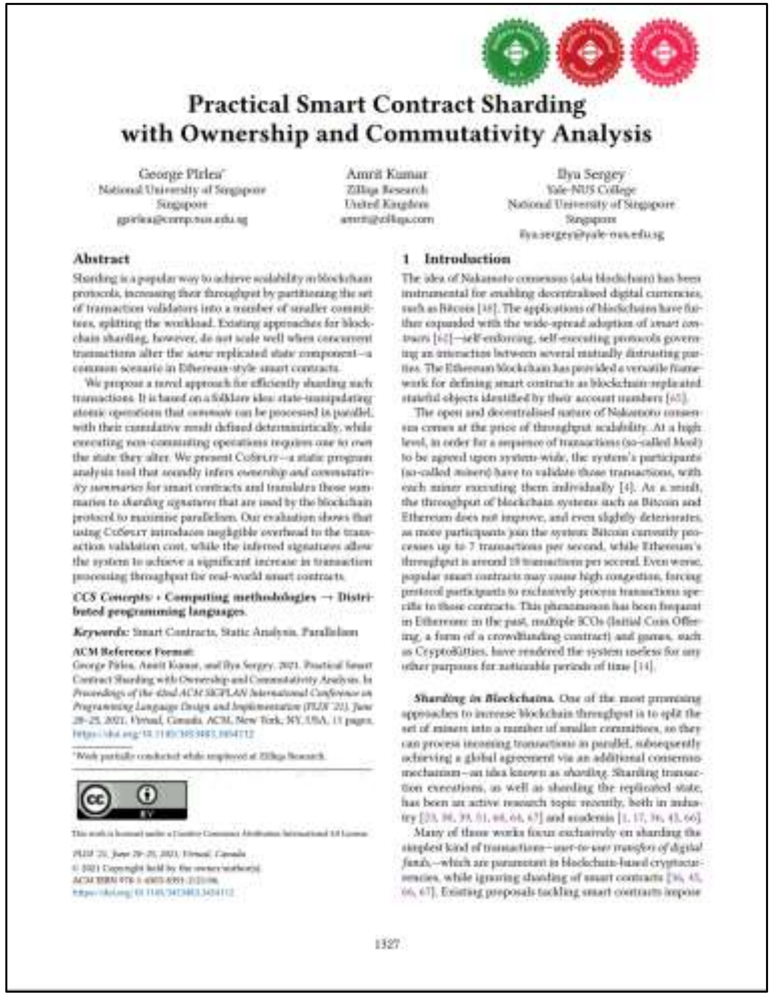
   = set of transition constraints + join instructions for each field in the contract

3. When processing a transaction, solve the constraints to determine which shard(s) the transaction can be processed by
   - if the constraints have no solution, must process sequentially/cross-shard

4. After parallel processing, merge (join) state contributions from shards before sequential transactions are processed

a parallelising compiler
for blockchains


Baseline 3 shards
CoSplit 3 shards
CoSplit 4 shards
CoSplit 5 shards
0
100
200
300
400
FT fund
FT transfer
CF donate
NFT mint
NFT transfer
ProofIPFS register
UD bestow
UD config



Practical Smart Contract Sharding with Ownership and Commutativity Analysis

George Pîrlea* — National University of Singapore, Singapore

Amrit Kumar — Zilliqa Research, United Kingdom

Ilya Sergey — Yale-NUS College, National University of Singapore, Singapore

Abstract

1 Introduction

1327

tinyurl.com/cosplit-paper

I’m happy to take questions.