

(paper published at PLDI'21)



CoSplit

Practical Smart Contract Sharding with Static Program Analysis

George Pîrlea, Amrit Kumar, Ilya Sergey



Smart Contract *Sharding* with Static Program Analysis

≈

Formal reasoning for *scalability*,
not correctness

Smart Contract *Sharding* with Static Program Analysis

≈

① Formal reasoning for *scalability*,
not correctness

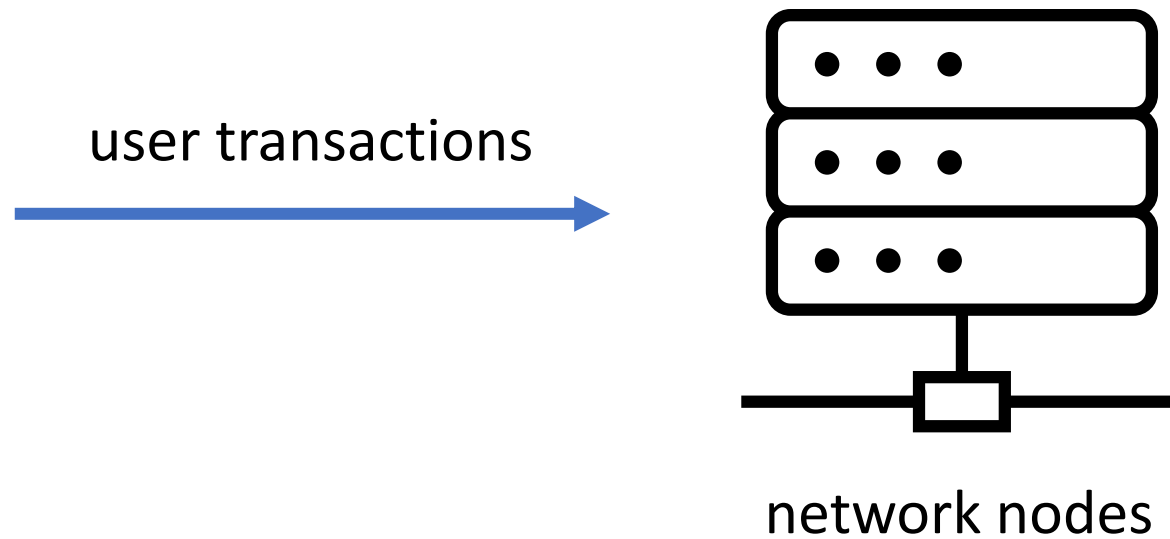
Smart Contract *Sharding* with Static Program Analysis

②

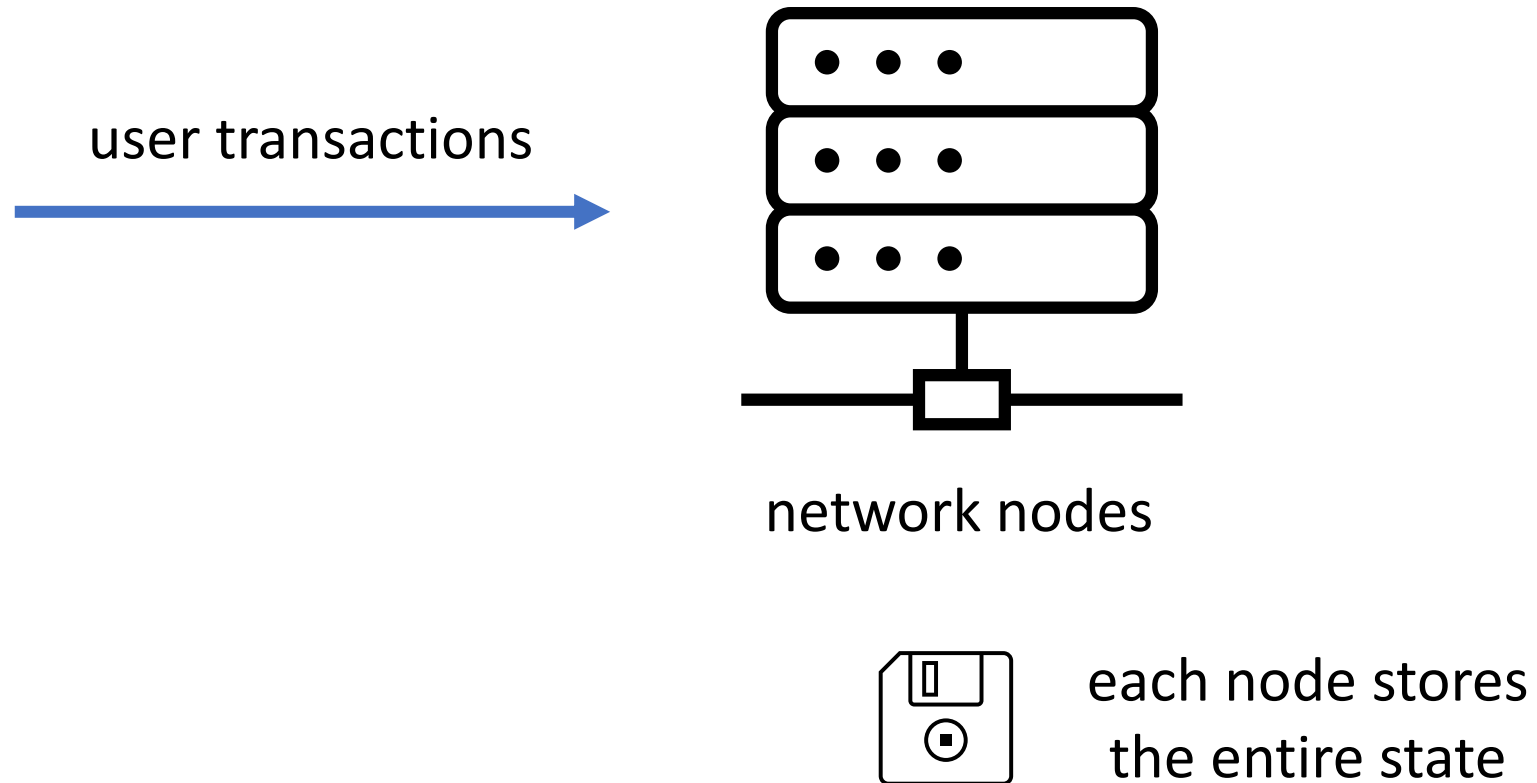
≈

① Formal reasoning for *scalability*,
not correctness

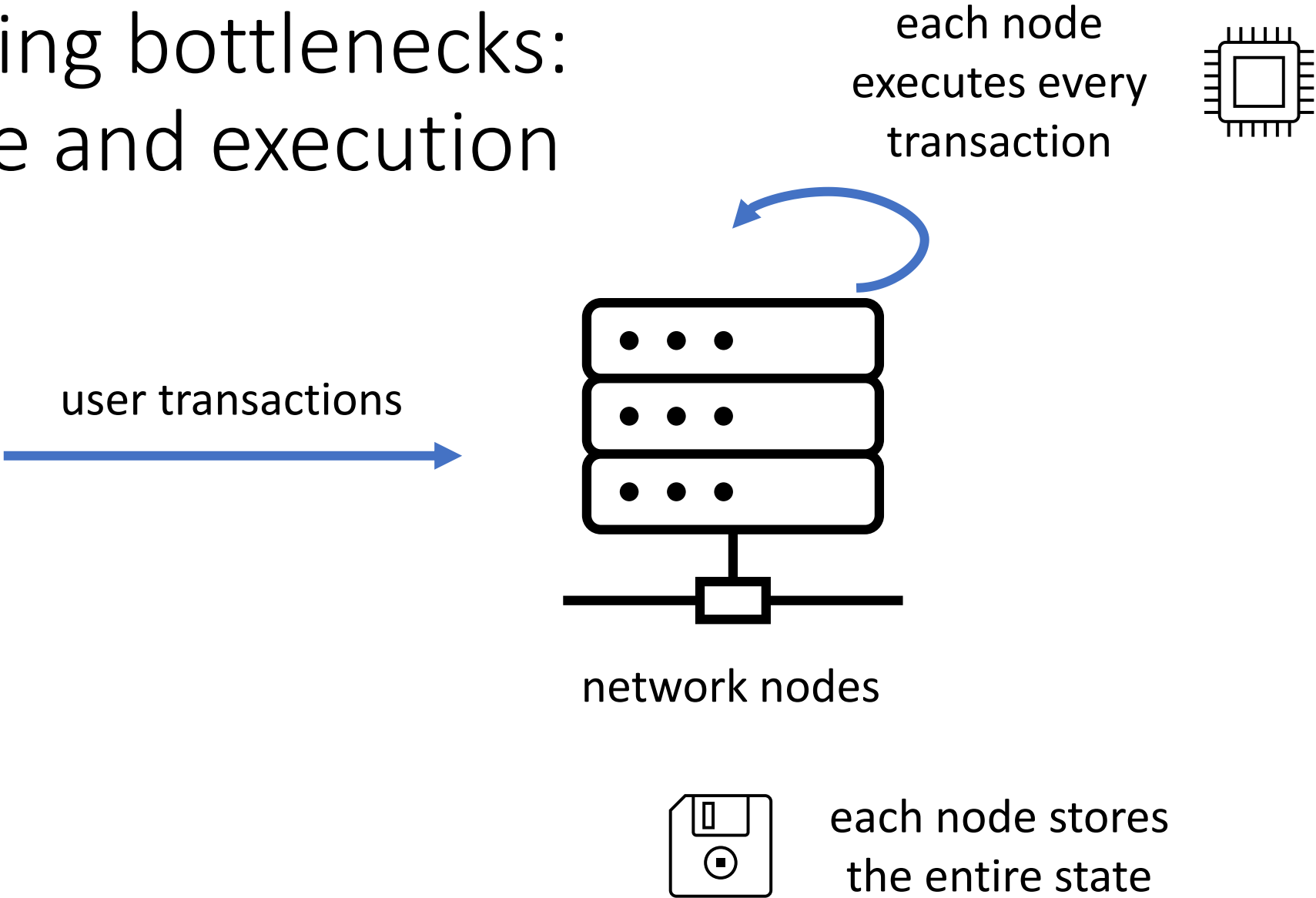
Scaling bottlenecks: state and execution



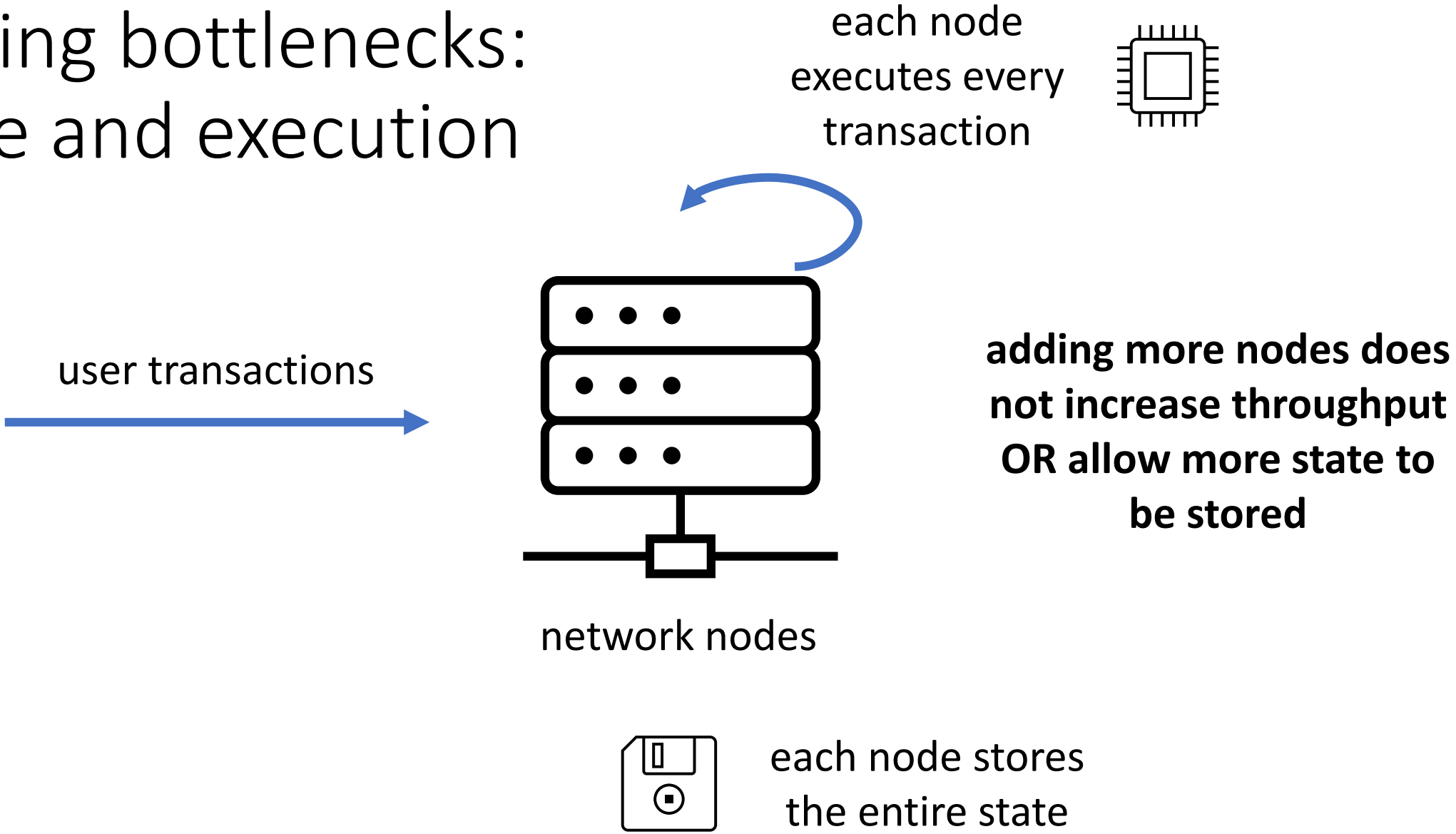
Scaling bottlenecks: state and execution



Scaling bottlenecks: state and execution



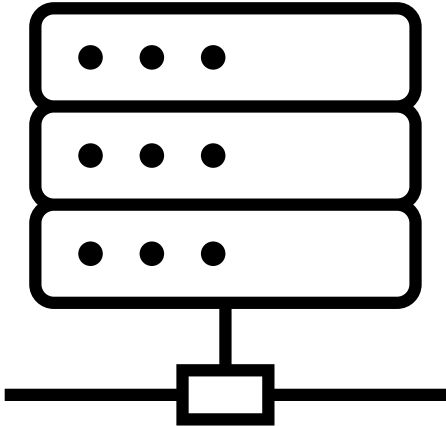
Scaling bottlenecks: state and execution



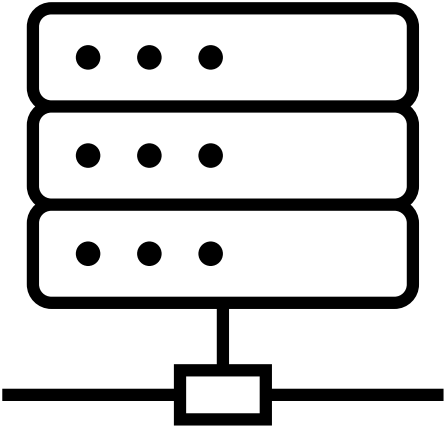
Scaling bottlenecks: state and execution

- 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

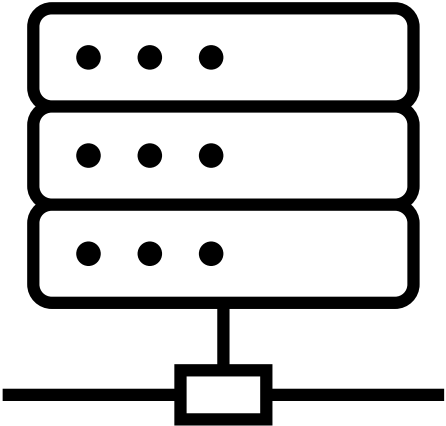
Sharding



shard 1 nodes



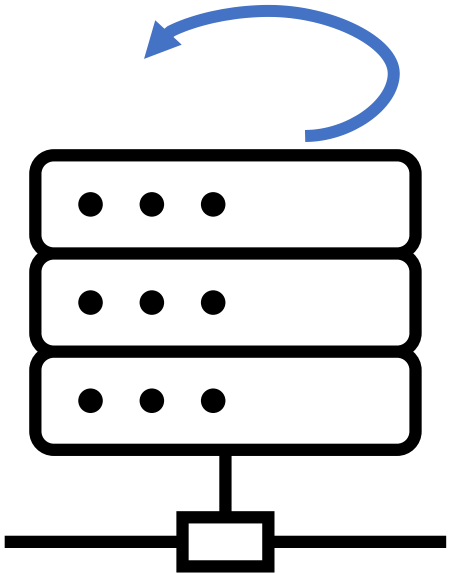
shard 2 nodes



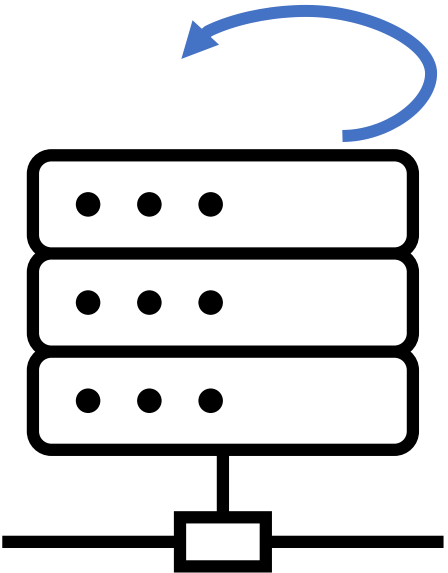
shard 3 nodes

Sharding

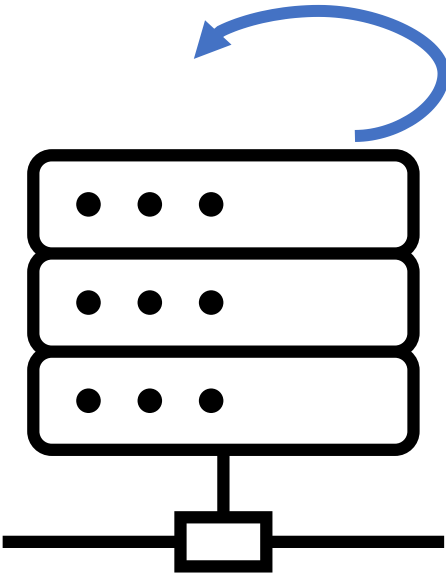
each shard
executes a subset
of transactions



shard 1 nodes



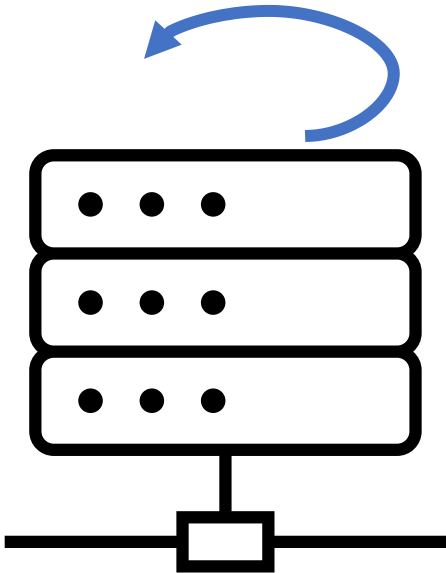
shard 2 nodes



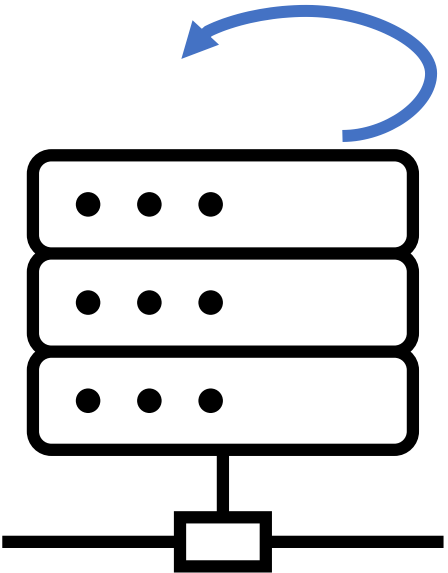
shard 3 nodes

Sharding

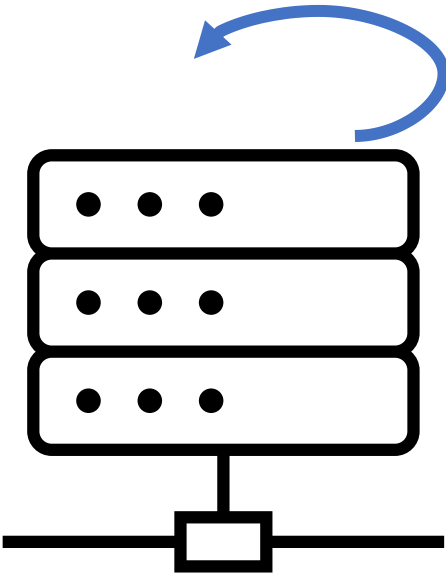
each shard
executes a subset
of transactions



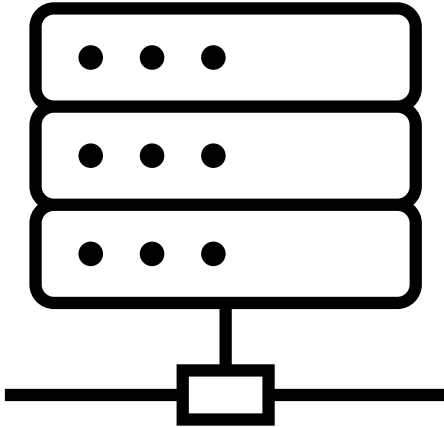
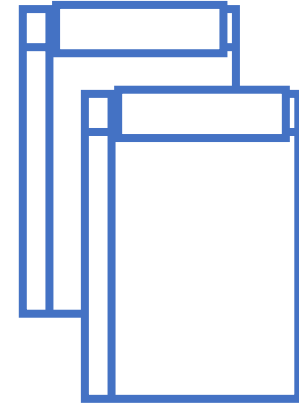
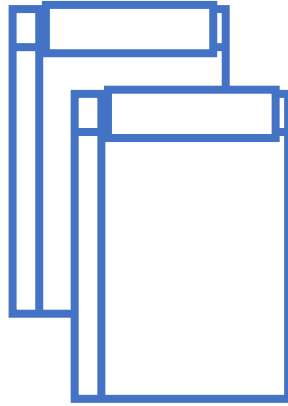
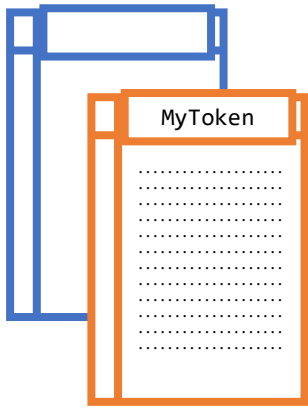
shard 1 nodes



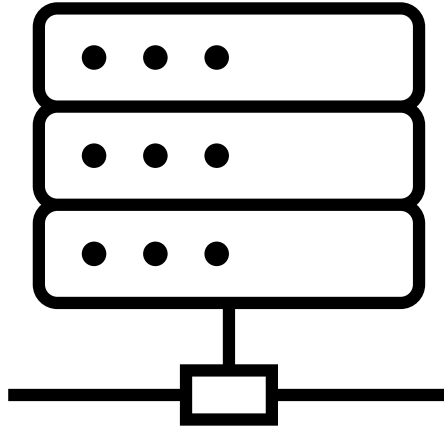
shard 2 nodes



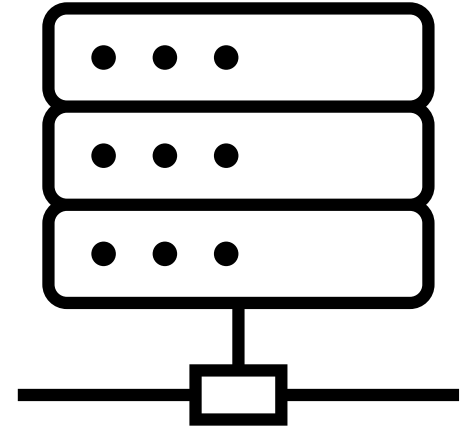
shard 3 nodes



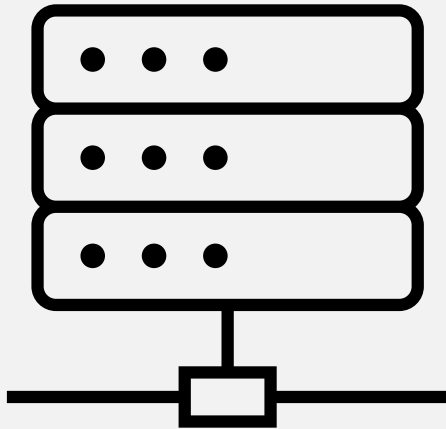
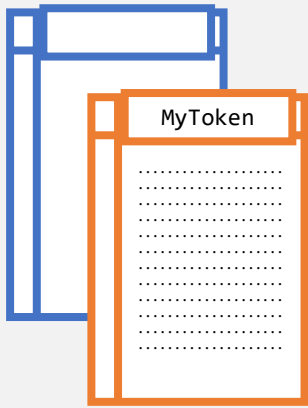
shard 1 nodes



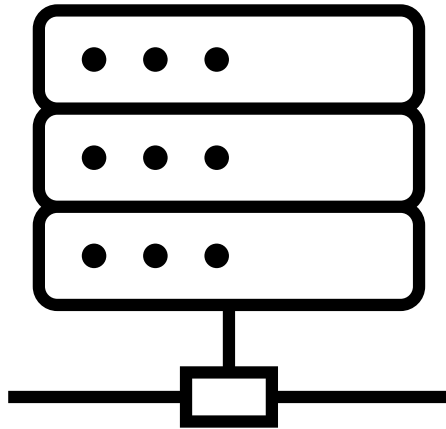
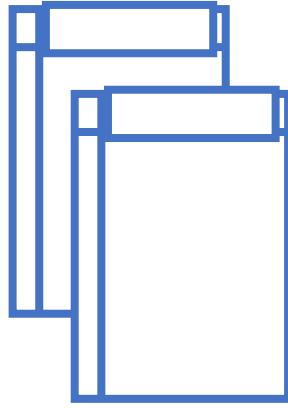
shard 2 nodes



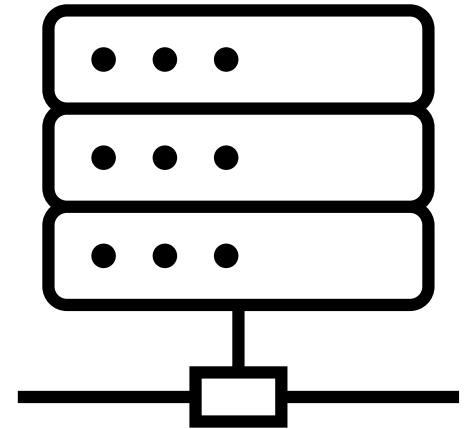
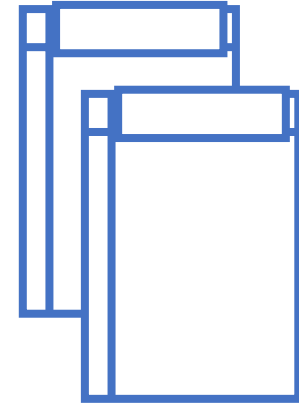
shard 3 nodes



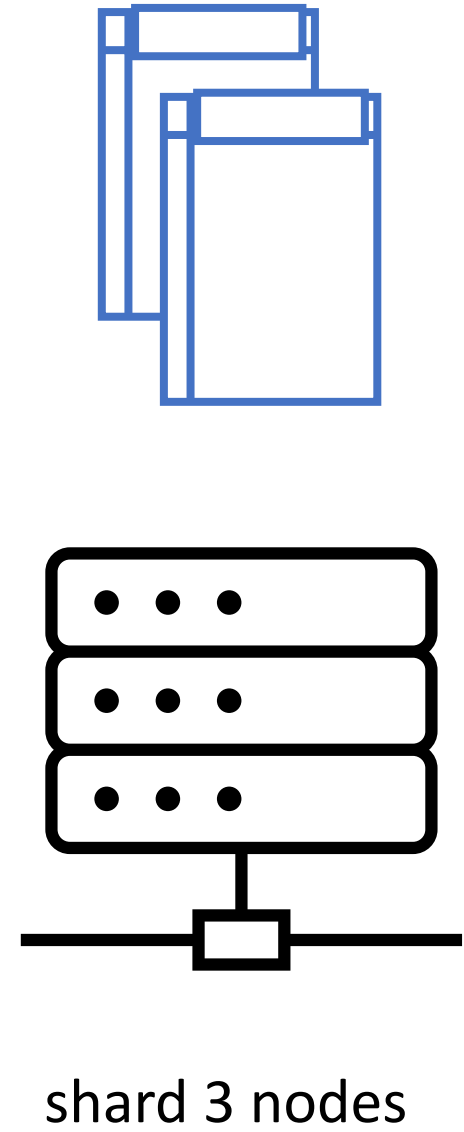
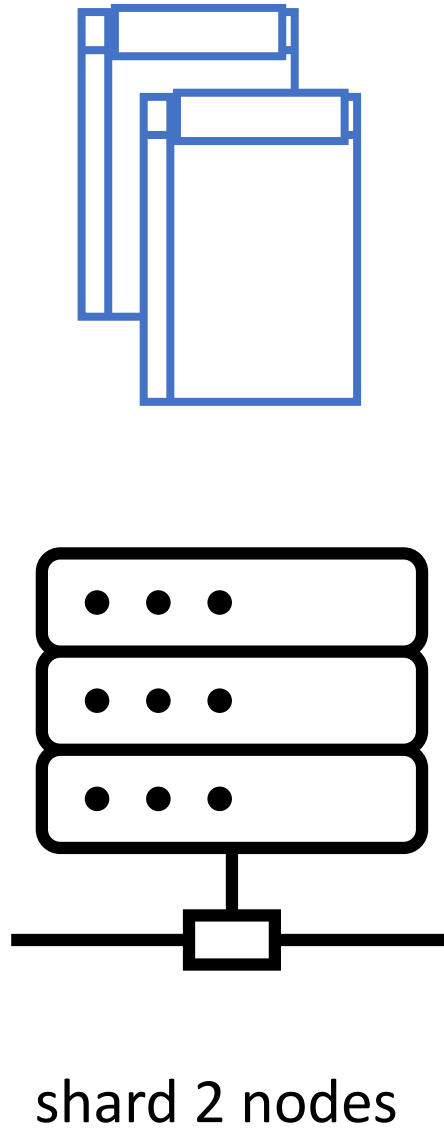
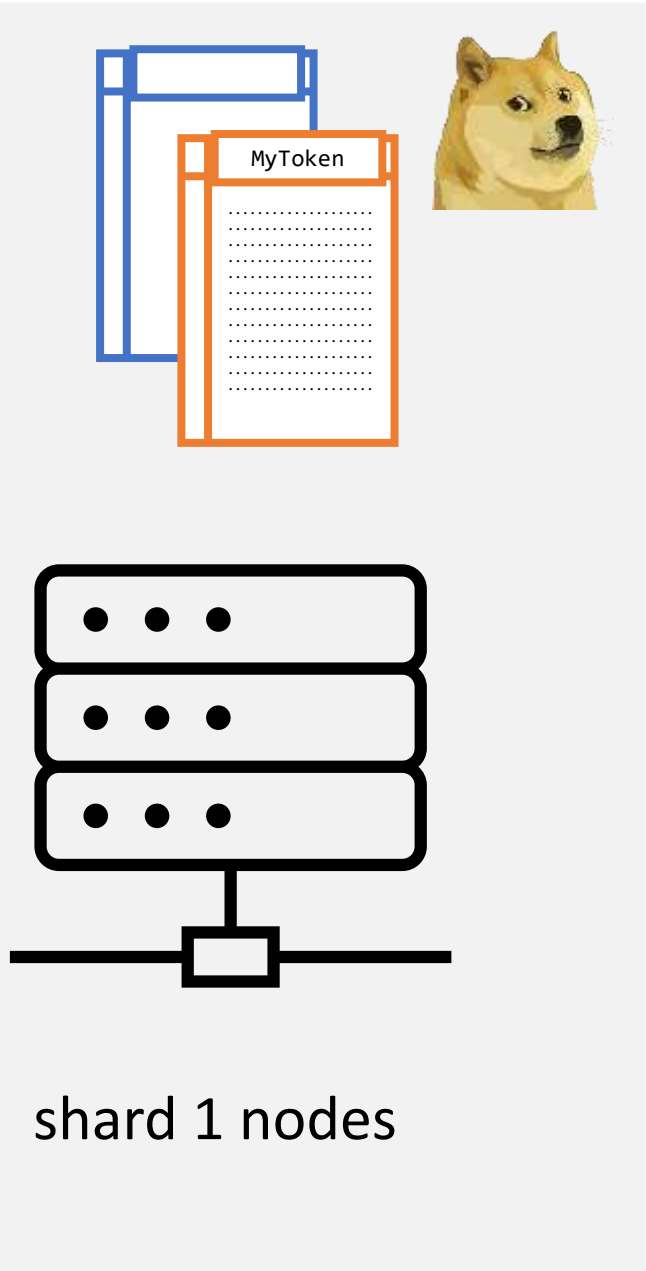
shard 1 nodes

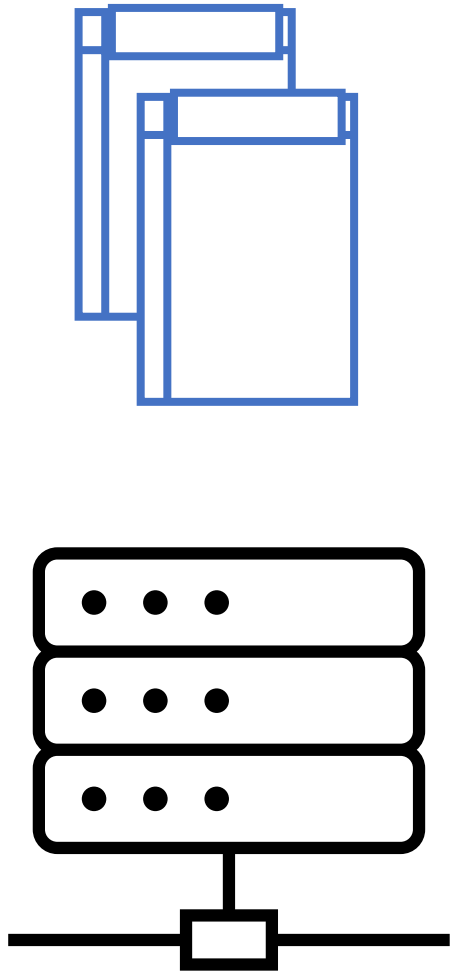
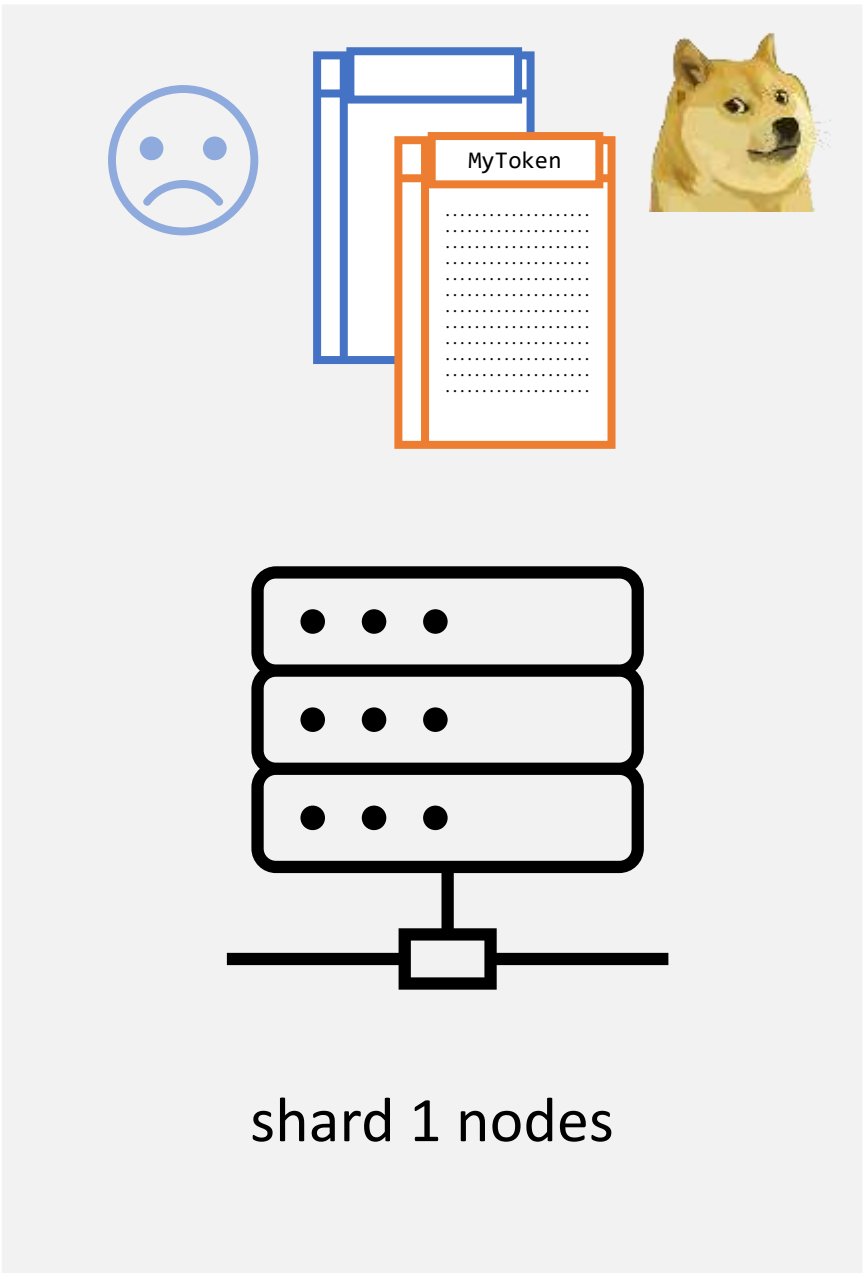


shard 2 nodes

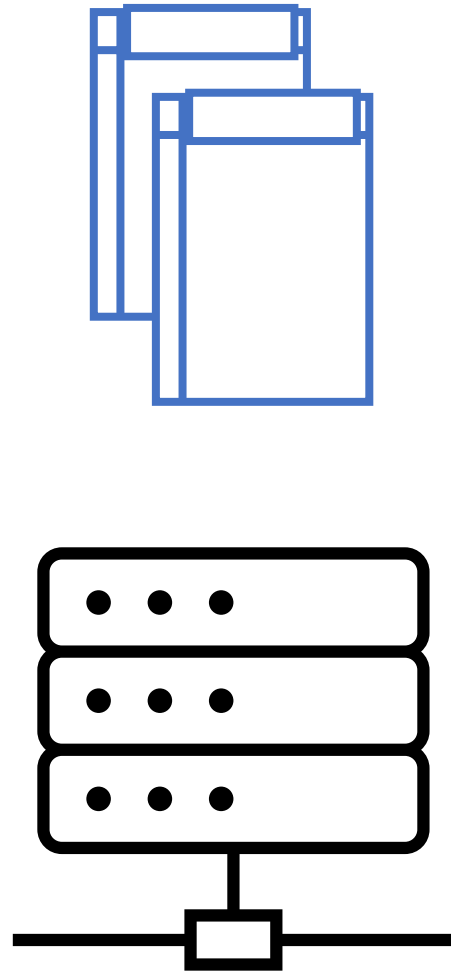


shard 3 nodes

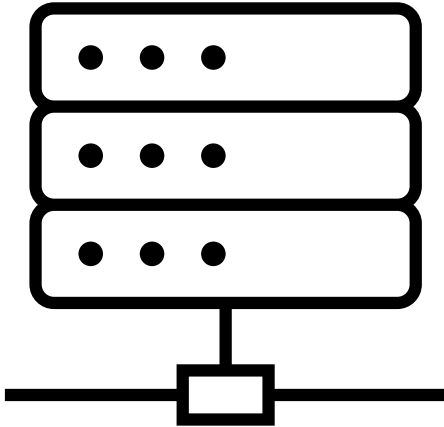
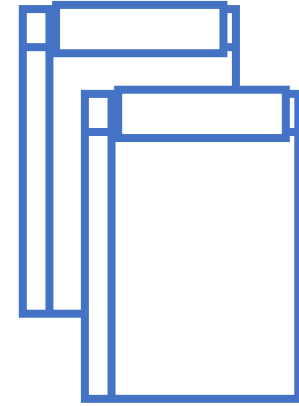
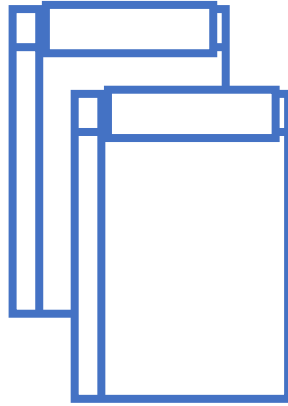
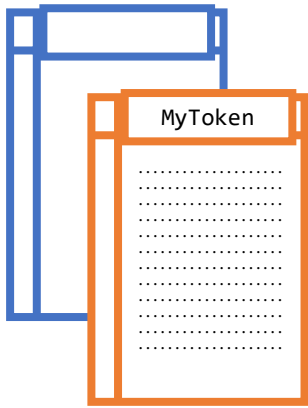




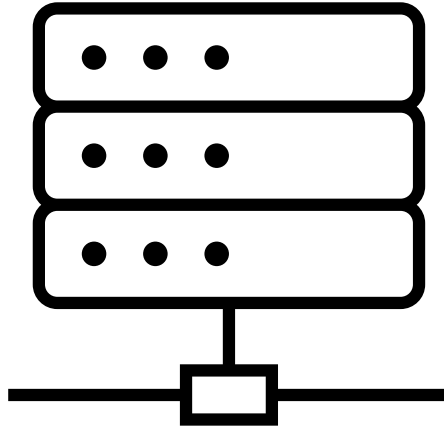
shard 2 nodes



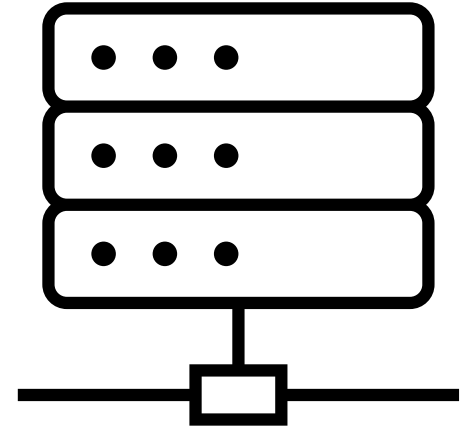
shard 3 nodes



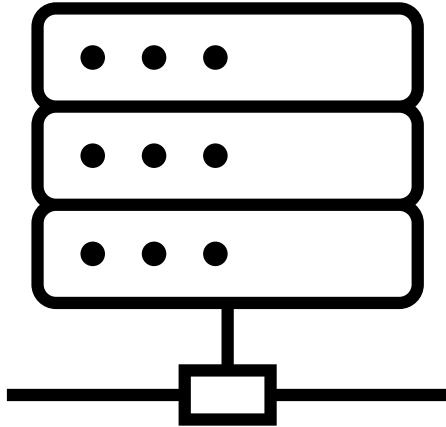
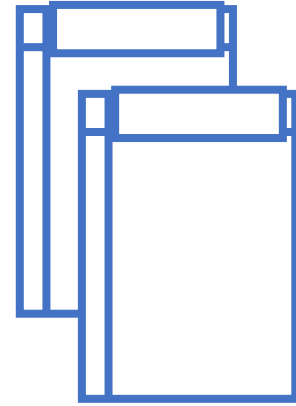
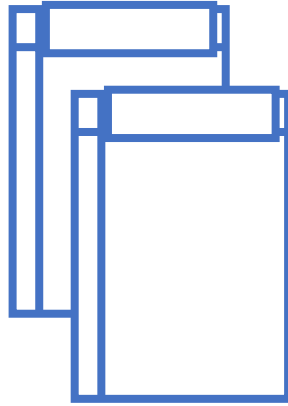
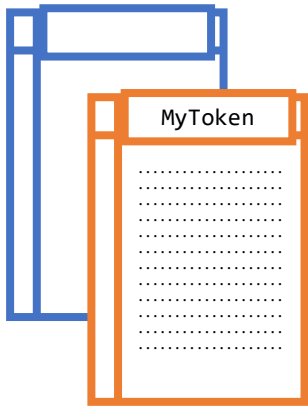
shard 1 nodes



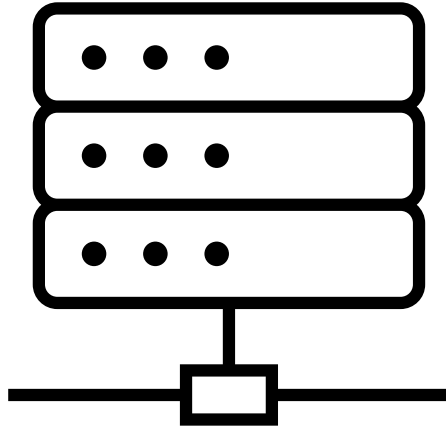
shard 2 nodes



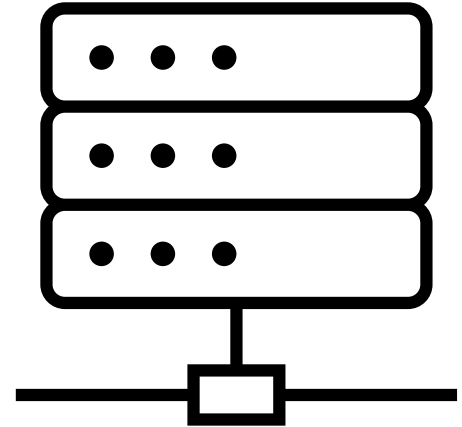
shard 3 nodes



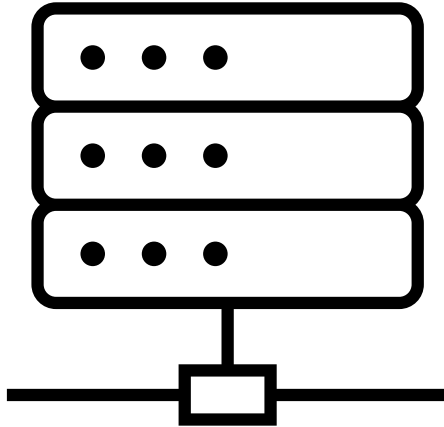
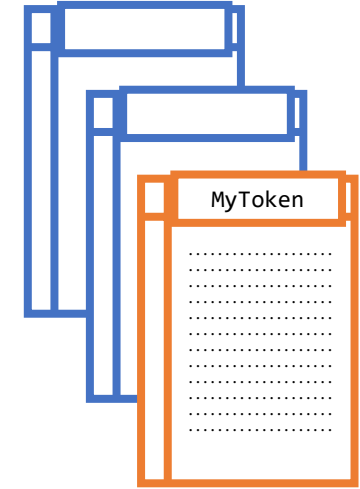
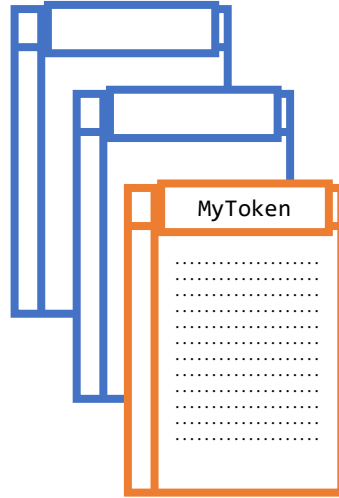
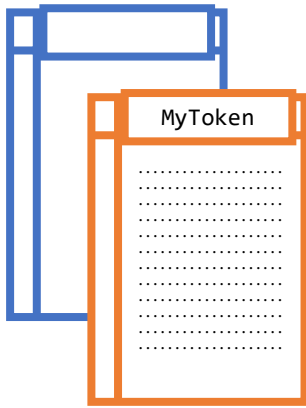
shard 1 nodes



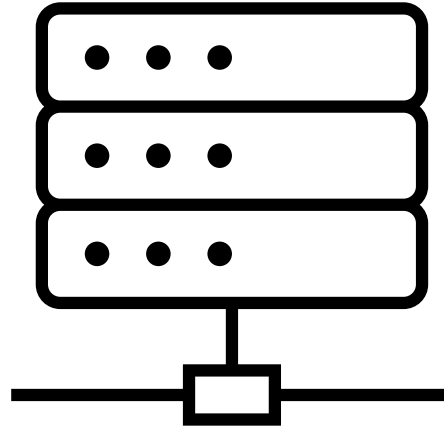
shard 2 nodes



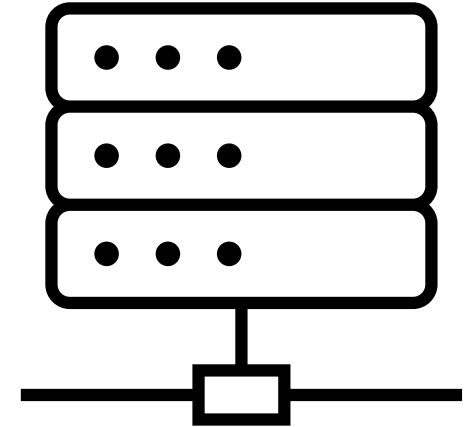
shard 3 nodes



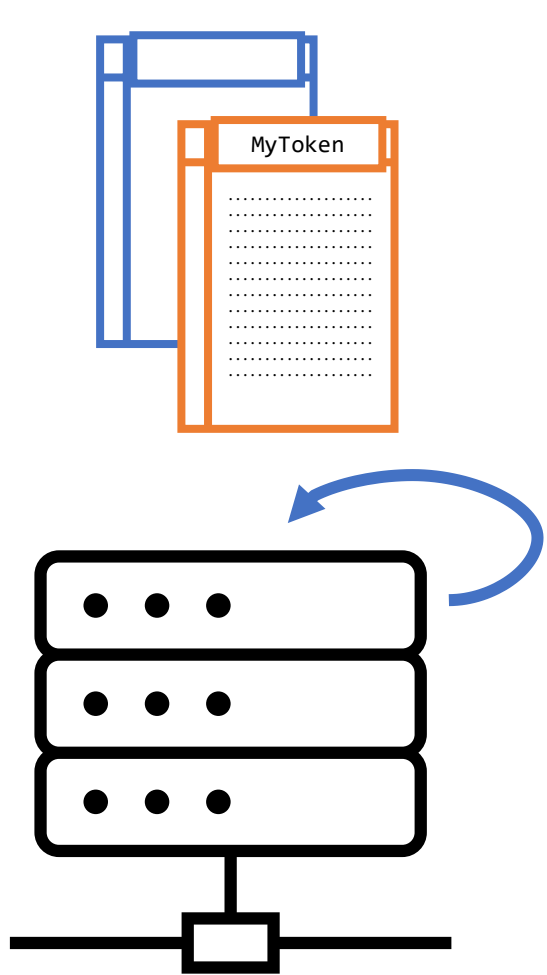
shard 1 nodes



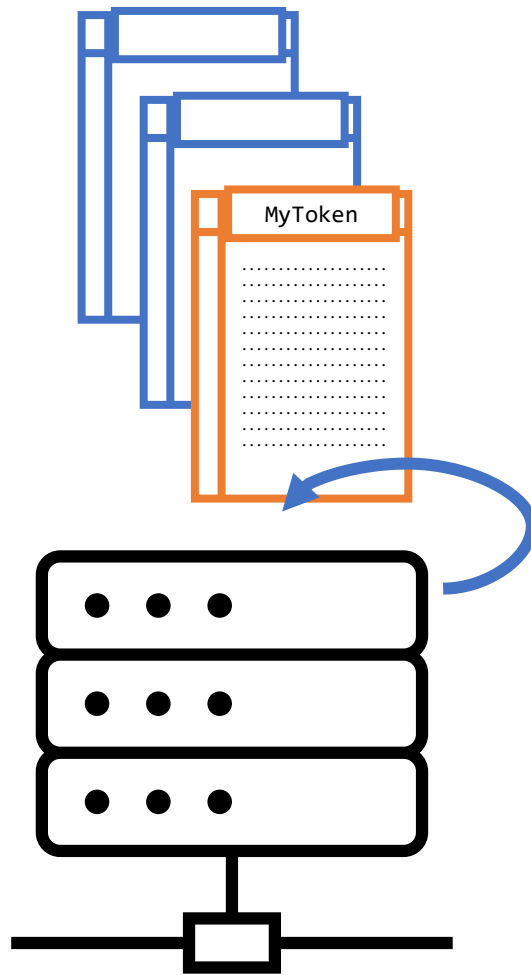
shard 2 nodes



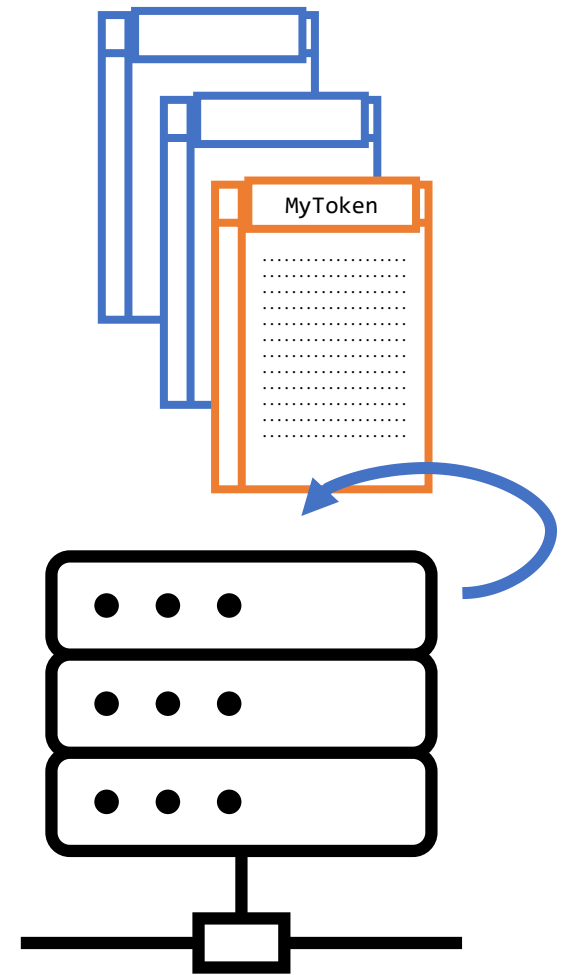
shard 3 nodes



shard 1 nodes



shard 2 nodes



shard 3 nodes

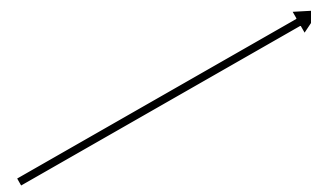
Look at the contract's code to learn how to shard it!



Smart Contract **Sharding**

with Static Program Analysis

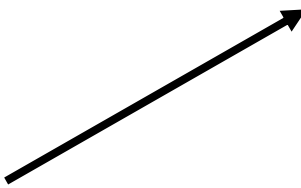
problem we are working on



Smart Contract **Sharding**

with Static Program Analysis

Smart Contract **Sharding**



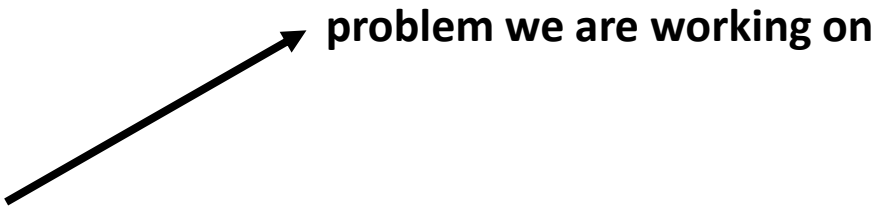
problem we are working on

with Static Program Analysis

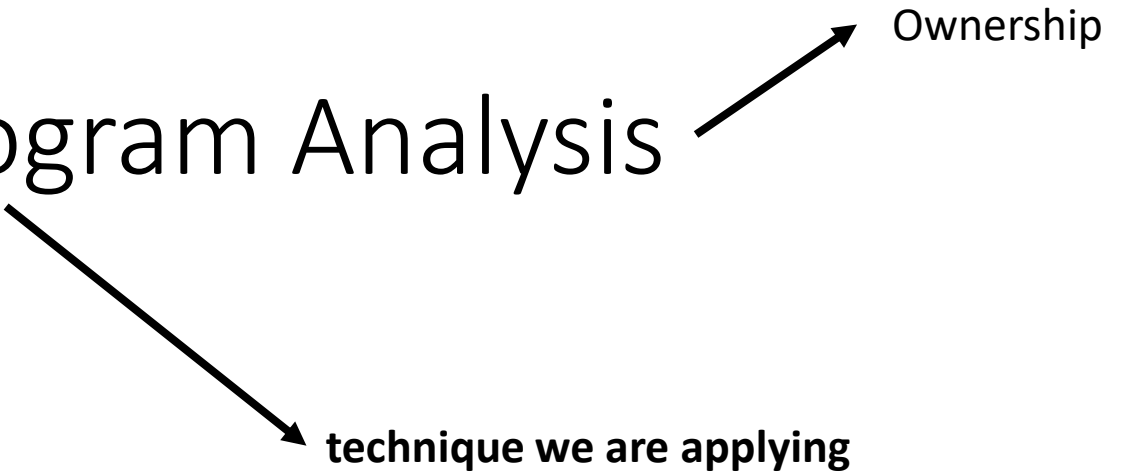


technique we are applying

Smart Contract **Sharding**



problem we are working on

with Static Program Analysis 

Ownership

technique we are applying

Smart Contract Sharding

problem we are working on

The diagram features the text 'Smart Contract Sharding' in a large font. The word 'Smart Contract' is in blue, and 'Sharding' is in red. An arrow points from the top of the word 'Sharding' to the text 'problem we are working on' located above and to the right of the main title.

with Static Program Analysis

Ownership




Commutativity

technique we are applying

The diagram features the text 'with Static Program Analysis' in a large font. The word 'with' is in italics. An arrow points from the bottom of the word 'Analysis' to the text 'technique we are applying' located below and to the right. Two other arrows point from the right side of the word 'Analysis' to the words 'Ownership' and 'Commutativity' located to the right of the main text.

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)

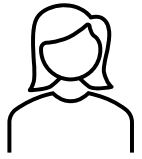
MyToken

Balances:

	Alice:	10
	Bob:	25
	Charlie:	12

Transitions:

BuyTokens(amount, buyer)
Transfer(amount, from, to)



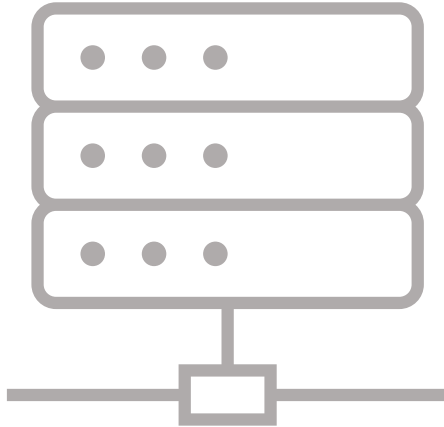
Alice: 1



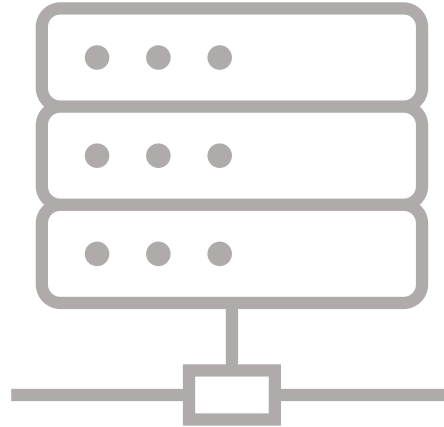
Bob: 25



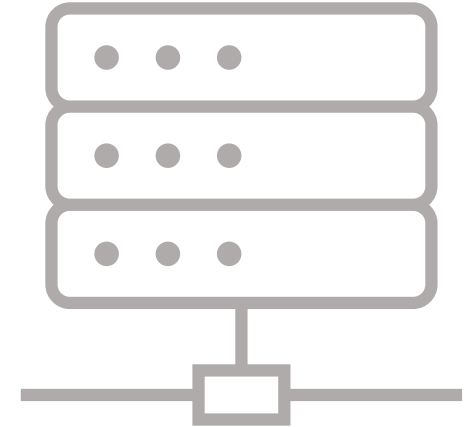
Charlie: 12



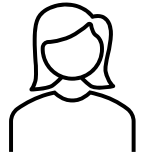
shard 1



shard 2



shard 3



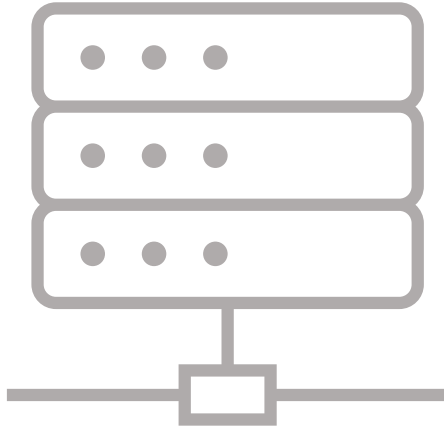
Alice: 11



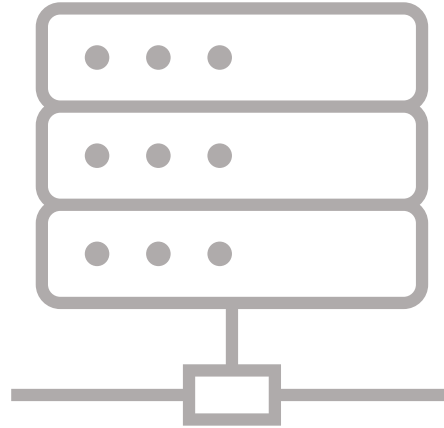
Bob: 25



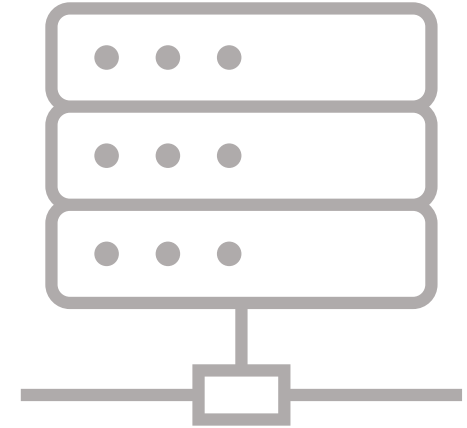
Charlie: 12



shard 1

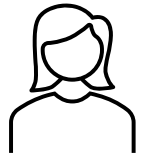


shard 2



shard 3

BuyTokens(amount, buyer)



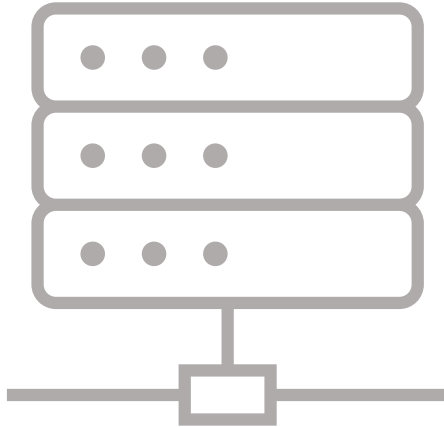
Alice: 11



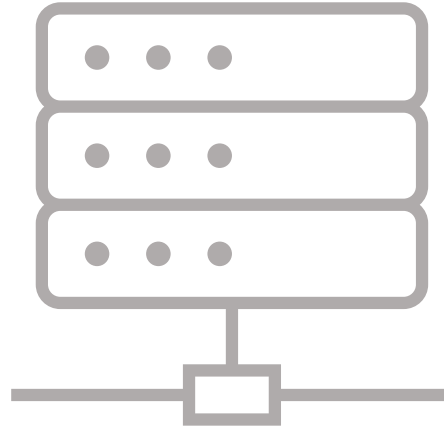
Bob: 25



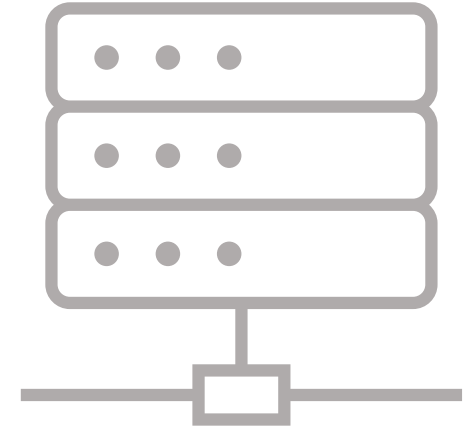
Charlie: 12



shard 1

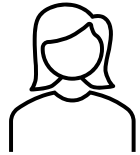


shard 2



shard 3

BuyTokens(5, Alice)



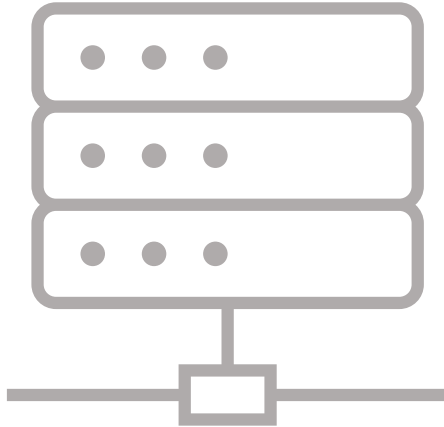
Alice: 11



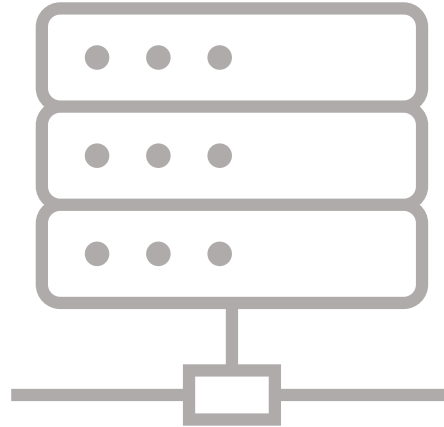
Bob: 25



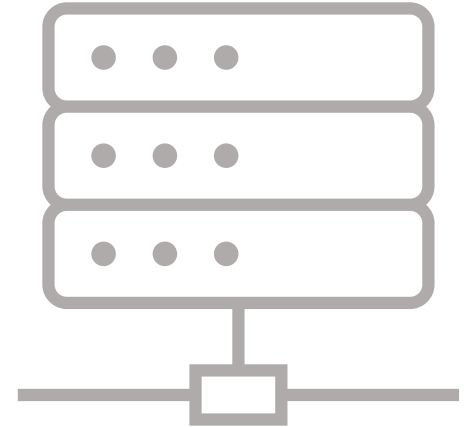
Charlie: 12



shard 1



shard 2

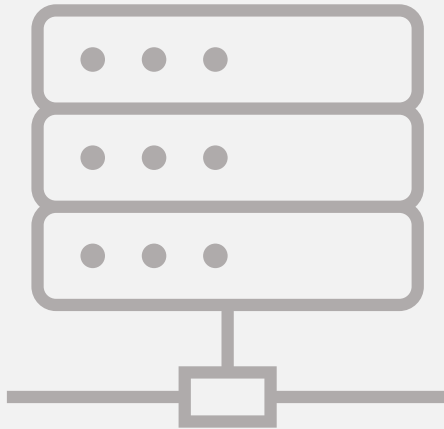


shard 3

BuyTokens(5, Alice)



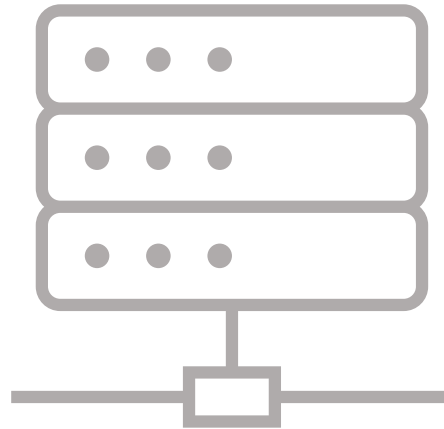
Alice: 11



shard 1



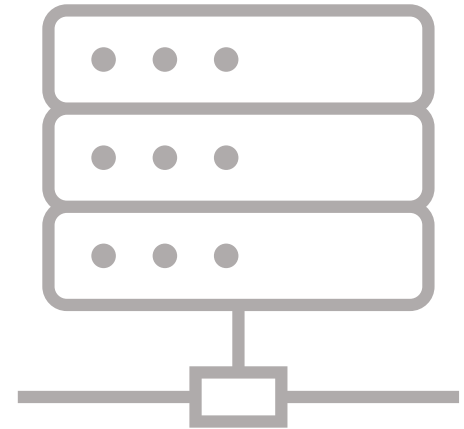
Bob: 25



shard 2



Charlie: 12



shard 3

BuyTokens(5, Alice)



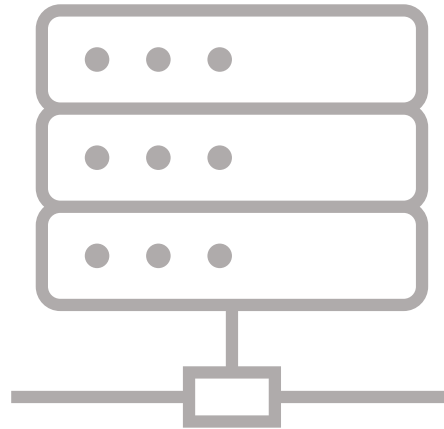
Alice: 11



shard 1



Bob: 25

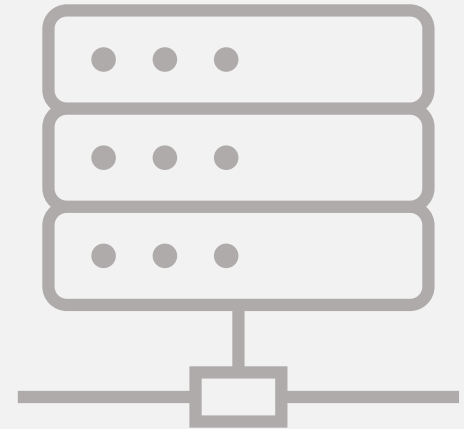


shard 2

BuyTokens(3, Charlie)



Charlie: 12



shard 3

BuyTokens(5, Alice)



Alice: 11



Bob: 25

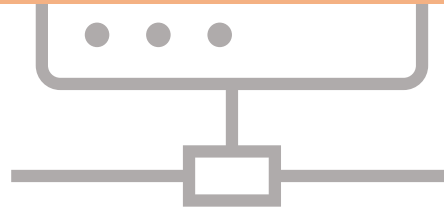


Charlie: 12

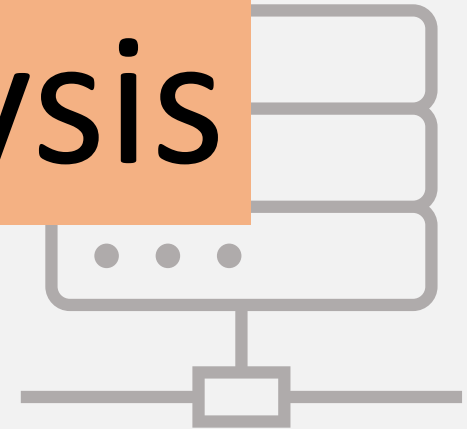
ownership analysis



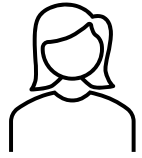
shard 1



shard 2



shard 3



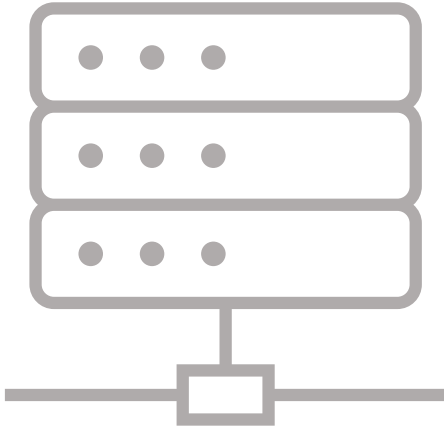
Alice: 11



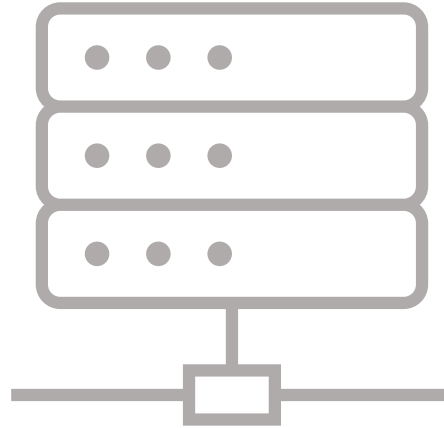
Bob: 25



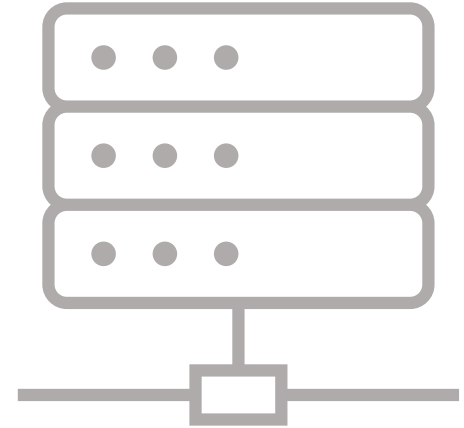
Charlie: 12



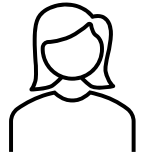
shard 1



shard 2



shard 3



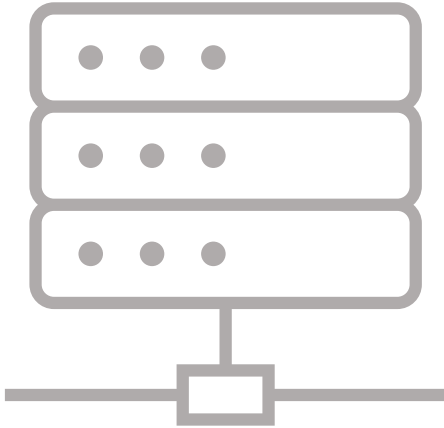
Alice: 11



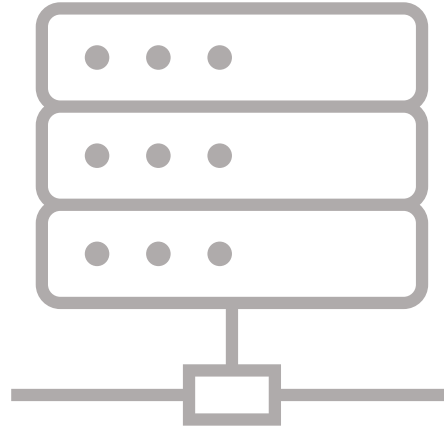
Bob: 25



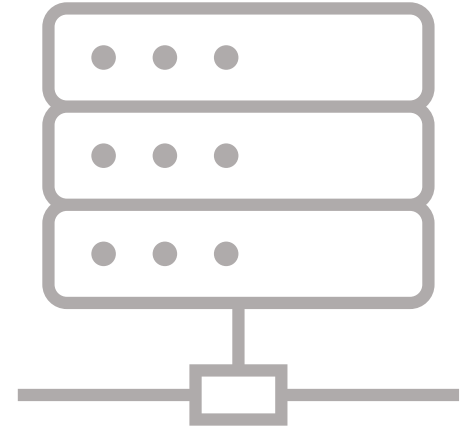
Charlie: 12



shard 1

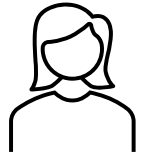


shard 2



shard 3

Transfer(5, Alice, Bob)



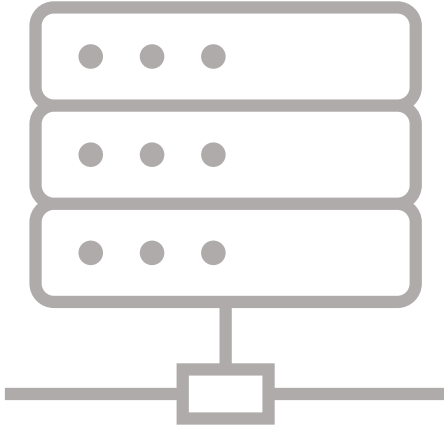
Alice: 11



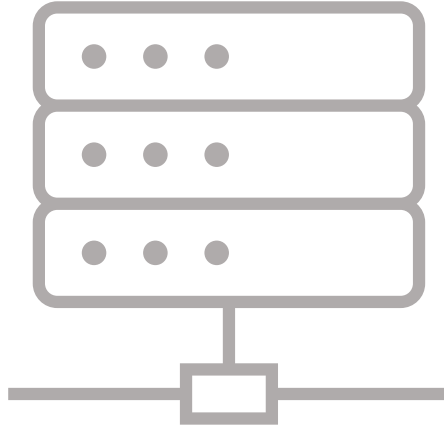
Bob: 25



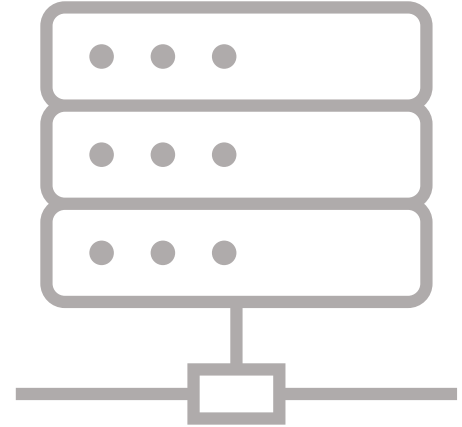
Charlie: 12



shard 1



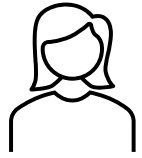
shard 2



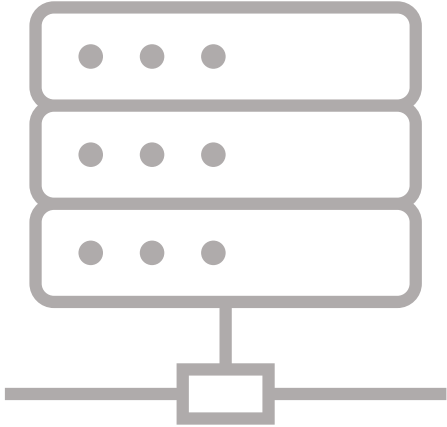
shard 3

Transfer(3, Charlie, Bob)

Transfer(5, Alice, Bob)



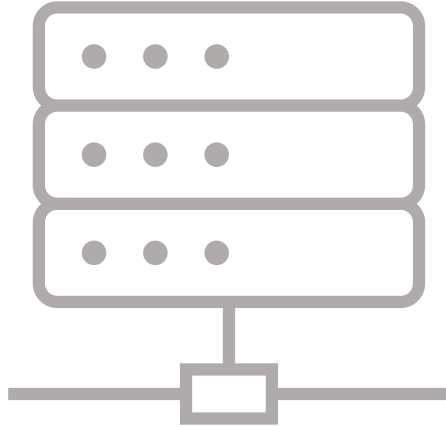
~~Alice: 11~~
Alice: 6



shard 1



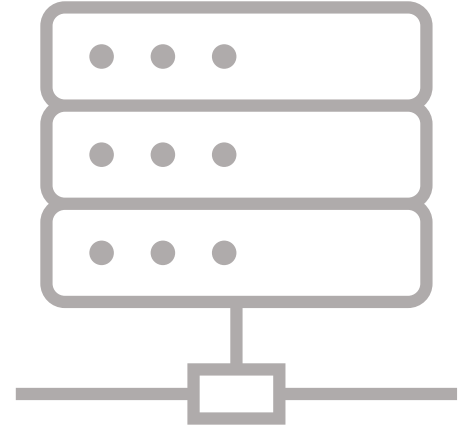
Bob: 25



shard 2



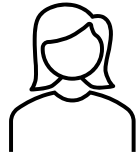
Charlie: 12



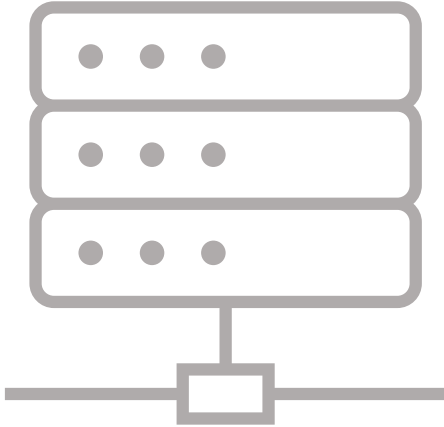
shard 3

Transfer(3, Charlie, Bob)

Transfer(5, Alice, Bob)



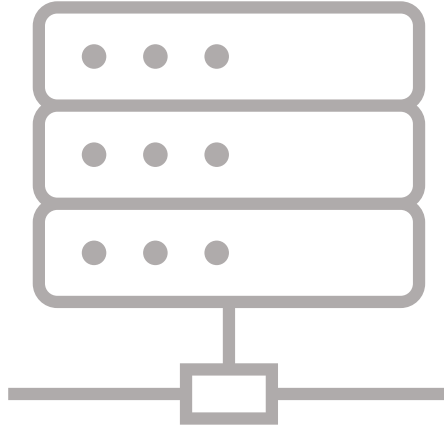
~~Alice: 11~~
Alice: 6



shard 1



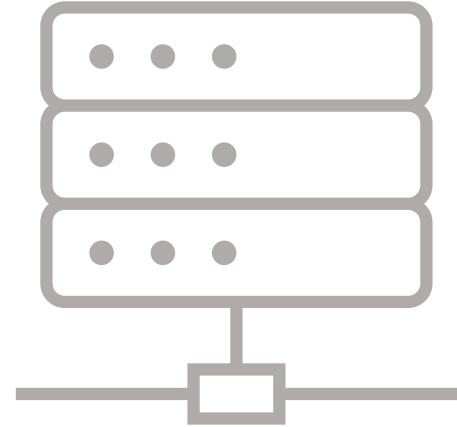
Bob: 25



shard 2



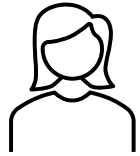
~~Charlie: 12~~
Charlie: 9



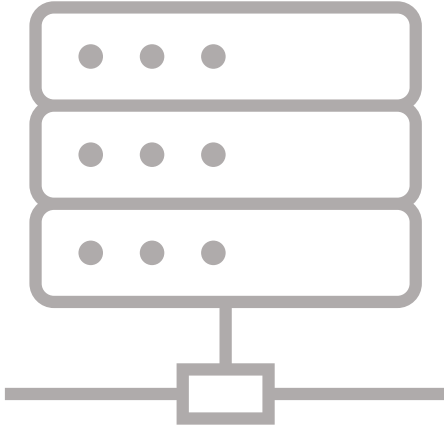
shard 3

Transfer(3, Charlie, Bob)

Transfer(5, Alice, Bob)



~~Alice: 11~~
Alice: 6

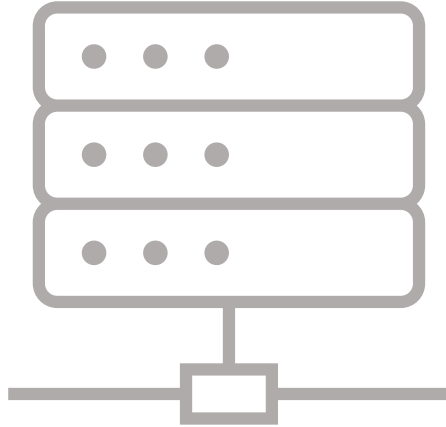


shard 1

Bob: +5



Bob: 25

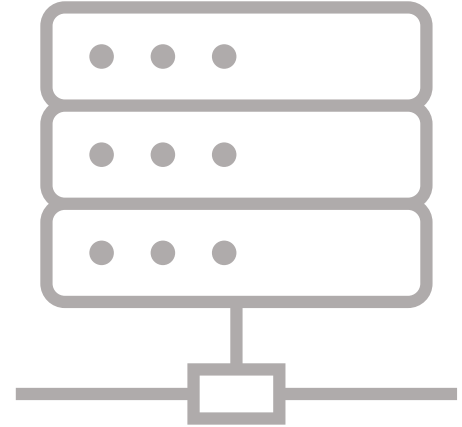


shard 2

Transfer(3, Charlie, Bob)

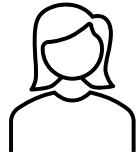


~~Charlie: 12~~
Charlie: 9

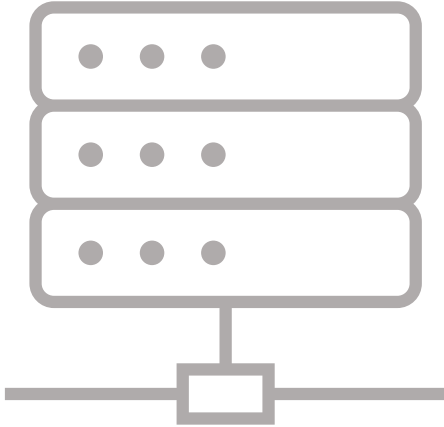


shard 3

Transfer(5, Alice, Bob)



~~Alice: 11~~
Alice: 6



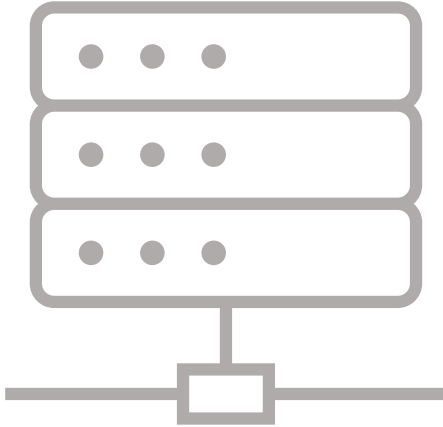
shard 1

Bob: +5

Bob: +3



Bob: 25

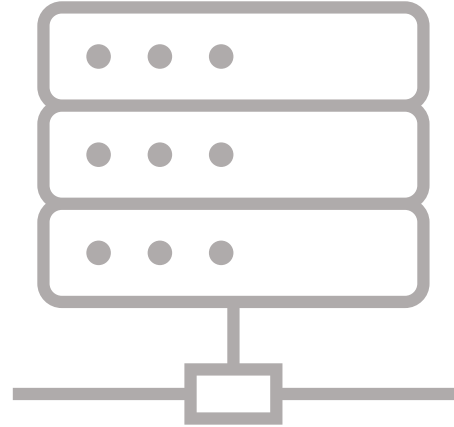


shard 2

Transfer(3, Charlie, Bob)

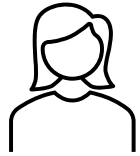


~~Charlie: 12~~
Charlie: 9

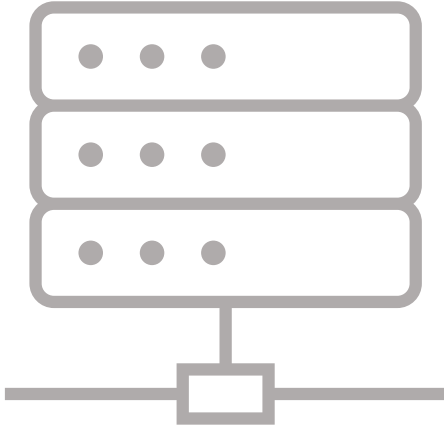


shard 3

Transfer(5, Alice, Bob)



~~Alice: 11~~
Alice: 6



shard 1

Bob: +5

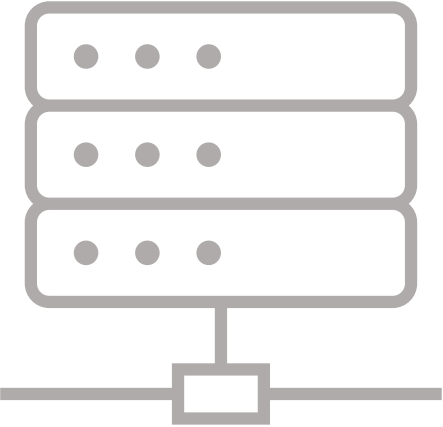
Bob: +3



= **Bob: +8**



Bob: 25

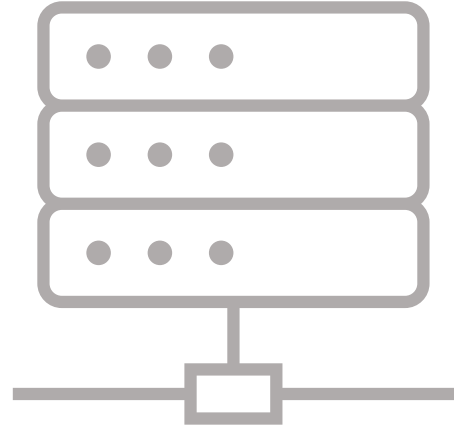


shard 2

Transfer(3, Charlie, Bob)

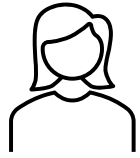


~~Charlie: 12~~
Charlie: 9

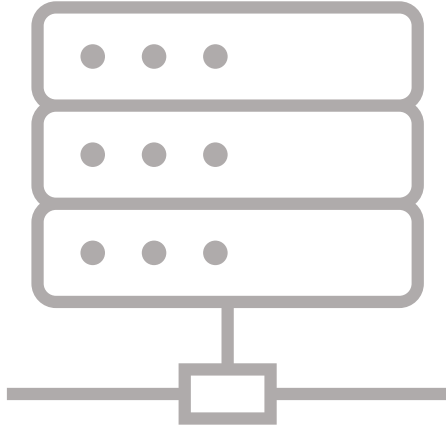


shard 3

Transfer(5, Alice, Bob)



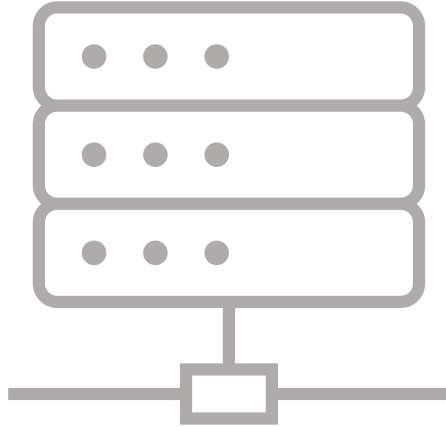
~~Alice: 11~~
Alice: 6



shard 1



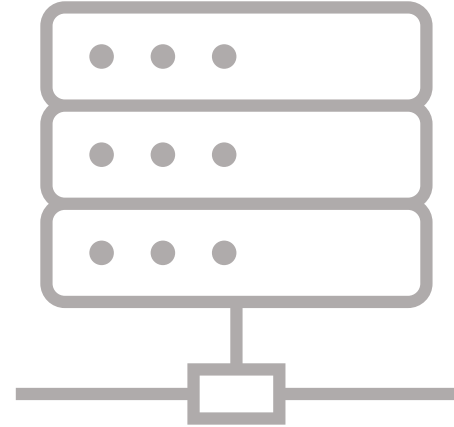
~~Bob: 25~~
Bob: 33



shard 2



~~Charlie: 12~~
Charlie: 9

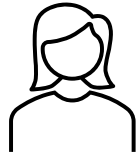


shard 3

Transfer(3, Charlie, Bob)

Transfer(5, Alice, Bob)

Transfer(3, Charlie, Bob)



~~Alice: 11~~
Alice: 6

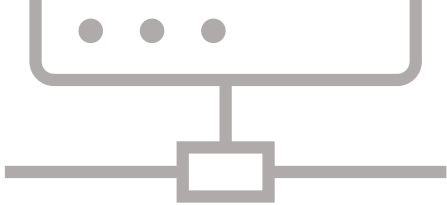


~~Bob: 25~~
Bob: 33

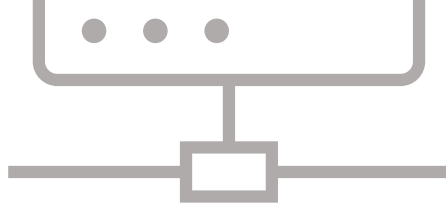


~~Charlie: 12~~
Charlie: 9

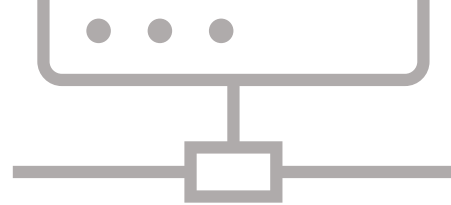
commutativity analysis



shard 1



shard 2



shard 3

CoSplit

Practical Smart Contract Sharding with
Static Program Analysis



+





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
```

```
1 transition Transfer(to: Address, amount: Uint)
2   from_bal <- balances[_sender];
3   match from_bal with
4   | Some bal =>
5     match amount ≤ bal with
6     | True =>
7       new_from_bal = builtin sub bal amount;
8       balances[_sender] := new_from_bal;
9       to_bal <- balances[to];
10      new_to_bal = match to_bal with
11      | Some bal => builtin add bal amount
12      | None => amount
13      end;
14      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
<hr/>		
Contrib. src.	$cs ::= x \mid f$	
Cardinality	$card ::= \text{None} \mid \text{Linear} \mid \text{NonLinear}$	
Operation	$op ::= + \mid - \mid * \mid \dots$	
Abstr. expr.	$e ::= \top \mid \overline{(cs, card, op)}$	
<hr/>		
Effect	$\varepsilon ::= \text{Read}(f) \mid \text{Write}(f, e) \mid \text{AcceptFunds} \mid$ $\text{Condition}(e) \mid \text{Event}(e) \mid \text{SendMsg}(e) \mid \top$	

```
1 transition Transfer(to: Address, amount: Uint)
2   from_bal <- balances[_sender];
3   match from_bal with
4   | Some bal =>
5     match amount ≤ bal with
6     | True =>
7       new_from_bal = builtin sub bal amount;
8       balances[_sender] := new_from_bal;
9       to_bal <- balances[to];
10      new_to_bal = match to_bal with
11      | Some bal => builtin add bal amount
12      | None => amount
13      end;
14      balances[to] := new_to_bal
```

```

1 transition Transfer(to: Address, amount: Uint)
2   from_bal <- balances[_sender];                               Read(balances[_sender])
3   match from_bal with
4     | Some bal =>
5       match amount ≤ bal with
6         | True =>
7           new_from_bal = builtin sub bal amount;
8           balances[_sender] := new_from_bal;
9           to_bal <- balances[to];
10          new_to_bal = match to_bal with
11            | Some bal => builtin add bal amount
12            | None => amount
13          end;
14          balances[to] := new_to_bal

```

```

1 transition Transfer(to: Address, amount: Uint)
2   from_bal <- balances[_sender];                                Read(balances[_sender])
3   match from_bal with                                         Condition(balances[_sender])
4   | Some bal =>
5     match amount ≤ bal with
6     | True =>
7       new_from_bal = builtin sub bal amount;
8       balances[_sender] := new_from_bal;
9       to_bal <- balances[to];
10      new_to_bal = match to_bal with
11      | Some bal => builtin add bal amount
12      | None => amount
13      end;
14      balances[to] := new_to_bal

```



```

1 transition Transfer(to: Address, amount: Uint)
2   from_bal <- balances[_sender];                               Read(balances[_sender])
3   match from_bal with                                         Condition(balances[_sender])
4   | Some bal =>                                               (balances[_sender], Linear, ∅)
5     match amount ≤ bal with
6     | True =>
7       new_from_bal = builtin sub bal amount;
8       balances[_sender] := new_from_bal;
9       to_bal <- balances[to];
10      new_to_bal = match to_bal with
11      | Some bal => builtin add bal amount
12      | None => amount
13      end;
14      balances[to] := new_to_bal

```

```

1 transition Transfer(to: Address, amount: Uint)
2   from_bal <- balances[_sender];                                Read(balances[_sender])
3   match from_bal with                                         Condition(balances[_sender])
4   | Some bal =>                                               (balances[_sender], Linear, ∅)
5   match amount ≤ bal with                                     Condition(balances[_sender], amount)
6   | True =>
7     new_from_bal = builtin sub bal amount;
8     balances[_sender] := new_from_bal;
9     to_bal <- balances[to];
10    new_to_bal = match to_bal with
11    | Some bal => builtin add bal amount
12    | None => amount
13  end;
14  balances[to] := new_to_bal

```

```

1 transition Transfer(to: Address, amount: Uint)
2   from_bal <- balances[_sender];                               Read(balances[_sender])
3   match from_bal with                                         Condition(balances[_sender])
4   | Some bal =>                                               (balances[_sender], Linear, ∅)
5   match amount ≤ bal with                                     Condition(balances[_sender], amount)
6   | True =>                                                    {(balances[_sender], Linear, sub),
7   new_from_bal = builtin sub bal amount;                       (amount, Linear, sub)}
8   balances[_sender] := new_from_bal;
9   to_bal <- balances[to];
10  new_to_bal = match to_bal with
11  | Some bal => builtin add bal amount
12  | None => amount
13  end;
14  balances[to] := new_to_bal

```

```

1 transition Transfer(to: Address, amount: Uint)
2   from_bal <- balances[_sender];           Read(balances[_sender])
3   match from_bal with                     Condition(balances[_sender])
4   | Some bal =>                            (balances[_sender], Linear, ∅)
5     match amount ≤ bal with              Condition(balances[_sender], amount)
6     | True =>                              {(balances[_sender], Linear, sub),
7       new_from_bal = builtin sub bal amount; (amount, Linear, sub)}
8     balances[_sender] := new_from_bal;    Write(balances[_sender],
9       to_bal <- balances[to];             {(balances[_sender], Linear, sub),
10      new_to_bal = match to_bal with      (amount, Linear, sub)})
11      | Some bal => builtin add bal amount
12      | None => amount
13    end;
14    balances[to] := new_to_bal

```

```

1 transition Transfer(to: Address, amount: Uint)
2   from_bal <- balances[_sender];           Read(balances[_sender])
3   match from_bal with                     Condition(balances[_sender])
4   | Some bal =>                             (balances[_sender], Linear, ∅)
5     match amount ≤ bal with               Condition(balances[_sender], amount)
6     | True =>                               {(balances[_sender], Linear, sub),
7       new_from_bal = builtin sub bal amount;   (amount, Linear, sub)}
8     balances[_sender] := new_from_bal;       Write(balances[_sender],
9     to_bal <- balances[to];                 {(balances[_sender], Linear, sub),
10    new_to_bal = match to_bal with           (amount, Linear, sub)})
11    | Some bal => builtin add bal amount     Read(balances[to])
12    | None => amount
13  end;
14  balances[to] := new_to_bal

```

```

1 transition Transfer(to: Address, amount: Uint)
2   from_bal <- balances[_sender];                               Read(balances[_sender])
3   match from_bal with                                         Condition(balances[_sender])
4   | Some bal =>                                                (balances[_sender], Linear, ∅)
5     match amount ≤ bal with                                    Condition(balances[_sender], amount)
6     | True =>                                                  {(balances[_sender], Linear, sub),
7       new_from_bal = builtin sub bal amount;                  (amount, Linear, sub)}
8     balances[_sender] := new_from_bal;                          Write(balances[_sender],
9     to_bal <- balances[to];                                     {(balances[_sender], Linear, sub),
10    new_to_bal = match to_bal with                               (amount, Linear, sub)})
11    | Some bal => builtin add bal amount                          Read(balances[to])
12    | None => amount
13  end;
14  balances[to] := new_to_bal                                   Write(balances[to],
                                                                {(balances[to], Linear, add),
                                                                (amount, Linear, add)})

```

Constraint

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

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

Join $\uplus_f ::= \text{OwnOverwrite} \mid \text{IntMerge}$

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

Join $\uplus_f ::= \text{OwnOverwrite} \mid \text{IntMerge}$

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

Join $\uplus_f ::= \text{OwnOverwrite} \mid \text{IntMerge}$

$\text{Owns}(\text{balances}[_\text{sender}])$

$\text{NoAliases}(\langle _ \text{sender}, \text{to} \rangle)$

$\text{Read}(\text{balances}[_\text{sender}])$

$\text{Condition}(\text{balances}[_\text{sender}])$

$\text{Condition}(\text{balances}[_\text{sender}], \text{amount})$

$\text{Write}(\text{balances}[_\text{sender}],$
 $\{(\text{balances}[_\text{sender}], \text{Linear}, \text{sub}),$
 $(\text{amount}, \text{Linear}, \text{sub})\})$

$\text{Read}(\text{balances}[\text{to}])$

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

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

Join $\uplus_f ::= \text{OwnOverwrite} \mid \text{IntMerge}$

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

Join $\uplus_f ::= \text{OwnOverwrite} \mid \text{IntMerge}$

Weak reads

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)})

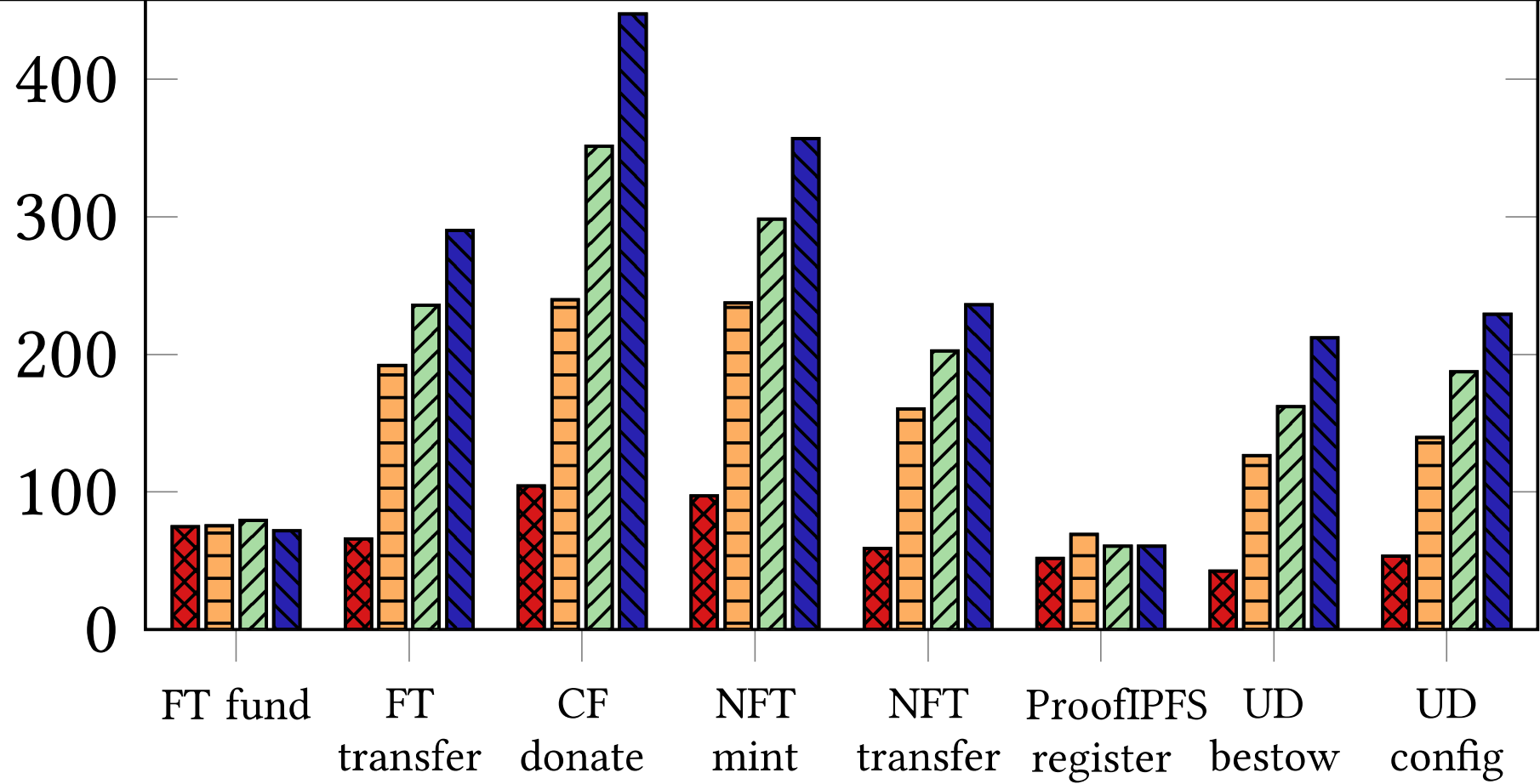
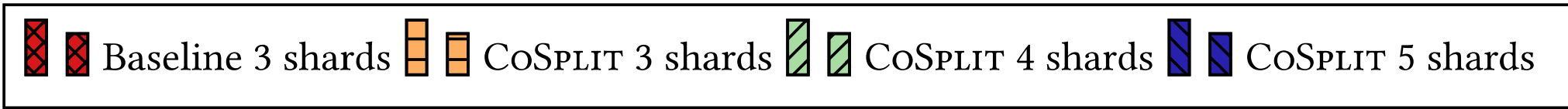
Read(balances[to])

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

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



Practical Smart Contract Sharding with Ownership and Commutativity Analysis

George Piliou*
National University of Singapore
Singapore
gpiliou@comp.nus.edu.sg

Anrith Kumar
Zilliq Research
United Kingdom
anrith@zilliq.com

Ilya Sergey
Yale-NUS College
National University of Singapore
Singapore
ilya.sergey@yale-nus.edu.sg

Abstract

Sharding is a popular way to achieve scalability in blockchain protocols, increasing their throughput by partitioning the set of transaction validators into a number of smaller committees, splitting the workload. Existing approaches for blockchain sharding, however, do not scale well when concurrent transactions alter the same replicated state component—a common scenario in Ethereum-style smart contracts.

We propose a novel approach for efficiently sharding such transactions. It is based on a folklore idea: state-transferring atomic operations that *commute* can be processed in parallel, with their cumulative result defined deterministically, while executing non-commuting operations requires one to own the state they alter. We present *Cosplit*—a static program analysis tool that *automatically* infers ownership and commutativity *invariants* for smart contracts and translates these summaries to sharding signatures that are used by the blockchain protocol to maximize parallelism. Our evaluation shows that using *Cosplit* introduces negligible overhead to the transaction validation cost, while the inferred signatures allow the system to achieve a significant increase in transaction processing throughput for real-world smart contracts.

CCS Concepts • Computing methodologies → Distributed programming languages.

Keywords Smart Contracts, Static Analysis, Parallelism

ACM Reference Format:

George Piliou, Anrith Kumar, and Ilya Sergey. 2021. Practical Smart Contract Sharding with Ownership and Commutativity Analysis. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 21)*, June 20–25, 2021, Virtual, Canada. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3453483.3454112>

*Work partially conducted while employed at Zilliq Research.



This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

PLDI 21, June 20–25, 2021, Virtual, Canada
© 2021 Copyright held by the author(s).
ACM ISBN 978-1-4503-8343-3/21/06.
<https://doi.org/10.1145/3453483.3454112>

1 Introduction

The idea of Nakamoto consensus (aka blockchain) has been instrumental for enabling decentralized digital currencies, such as Bitcoin [18]. The applications of blockchain have further expanded with the wide-spread adoption of smart contracts [6]—self-enforcing, self-executing protocols governing an interaction between several mutually distrusting parties. The Ethereum blockchain has provided a versatile framework for defining smart contracts as blockchain-represented stateful objects identified by their account numbers [5].

The open and decentralized nature of Nakamoto consensus comes at the price of throughput scalability. At a high level, in order for a sequence of transactions (so-called *block*) to be agreed upon system-wide, the system's participants (so-called *miners*) have to validate those transactions, with each *miner* executing them individually [4]. As a result, the throughput of blockchain systems such as Bitcoin and Ethereum does not improve, and even slightly deteriorates, as more participants join the system: Bitcoin currently processes up to 7 transactions per second, while Ethereum's throughput is around 10 transactions per second. Even worse, popular smart contracts may cause high congestion, forcing protocol participants to exclusively process transactions specific to those contracts. This phenomenon has been rampant in Ethereum in the past, multiple ICOs (Initial Coin Offerings, a form of a crowdfunding contract) and games, such as CryptoKitties, have rendered the system useless for any other purposes for noticeable periods of time [14].

Sharding in blockchains. One of the most promising approaches to increase blockchain throughput is to split the set of miners into a number of smaller committees, so they can process incoming transactions in parallel, subsequently achieving a global agreement via an additional consensus mechanism—an idea known as *sharding*. Sharding transaction execution, as well as sharding the replicated state, has been an active research topic recently, both in industry [23, 36, 39, 51, 64, 65, 67] and academia [1, 17, 36, 43, 66].

Many of these works focus exclusively on sharding the simplest kind of transactions—*one-to-one* transfers of digital funds—which are predominant in blockchain-based cryptocurrencies, while ignoring sharding of smart contracts [36, 43, 66, 67]. Existing proposals tackling smart contracts impose

I'm happy to
take questions.

tinyurl.com/cosplit-paper