Hashing with SL_2

Hash Functions as Monoid Homomorphisms

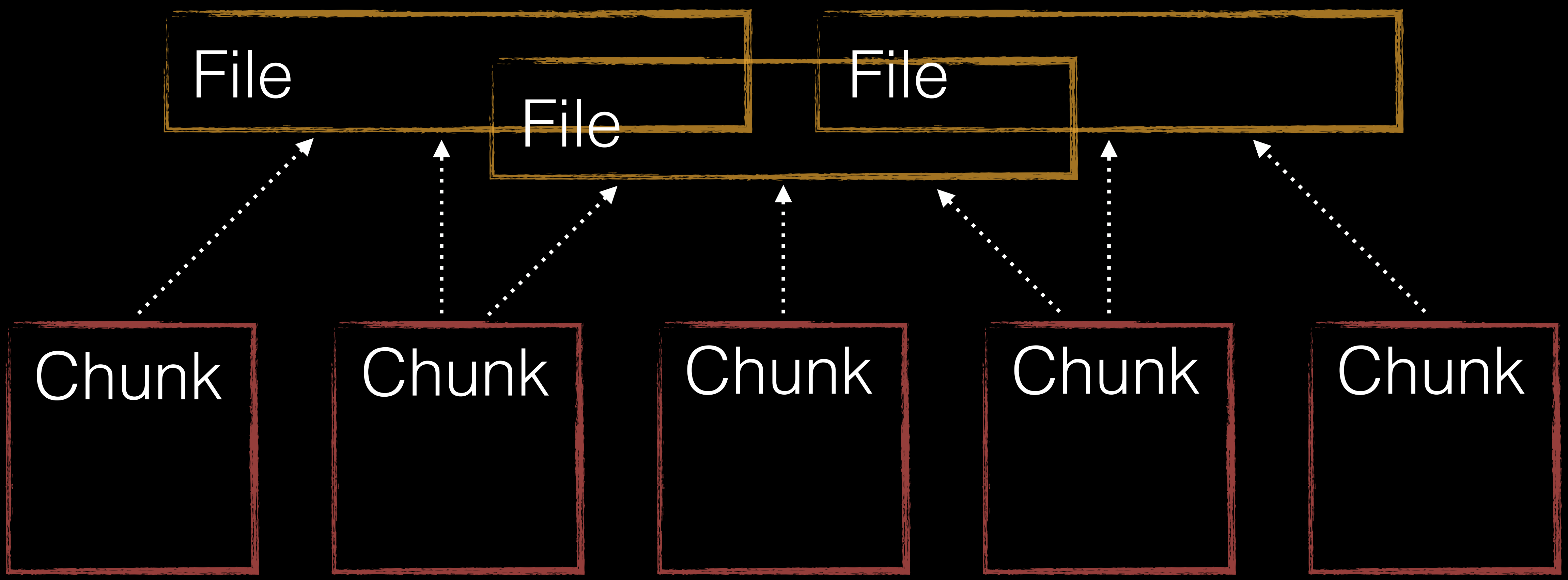
Based on the eponymous paper by Jean-Pierre Tillich and Gilles Zemor

Agenda

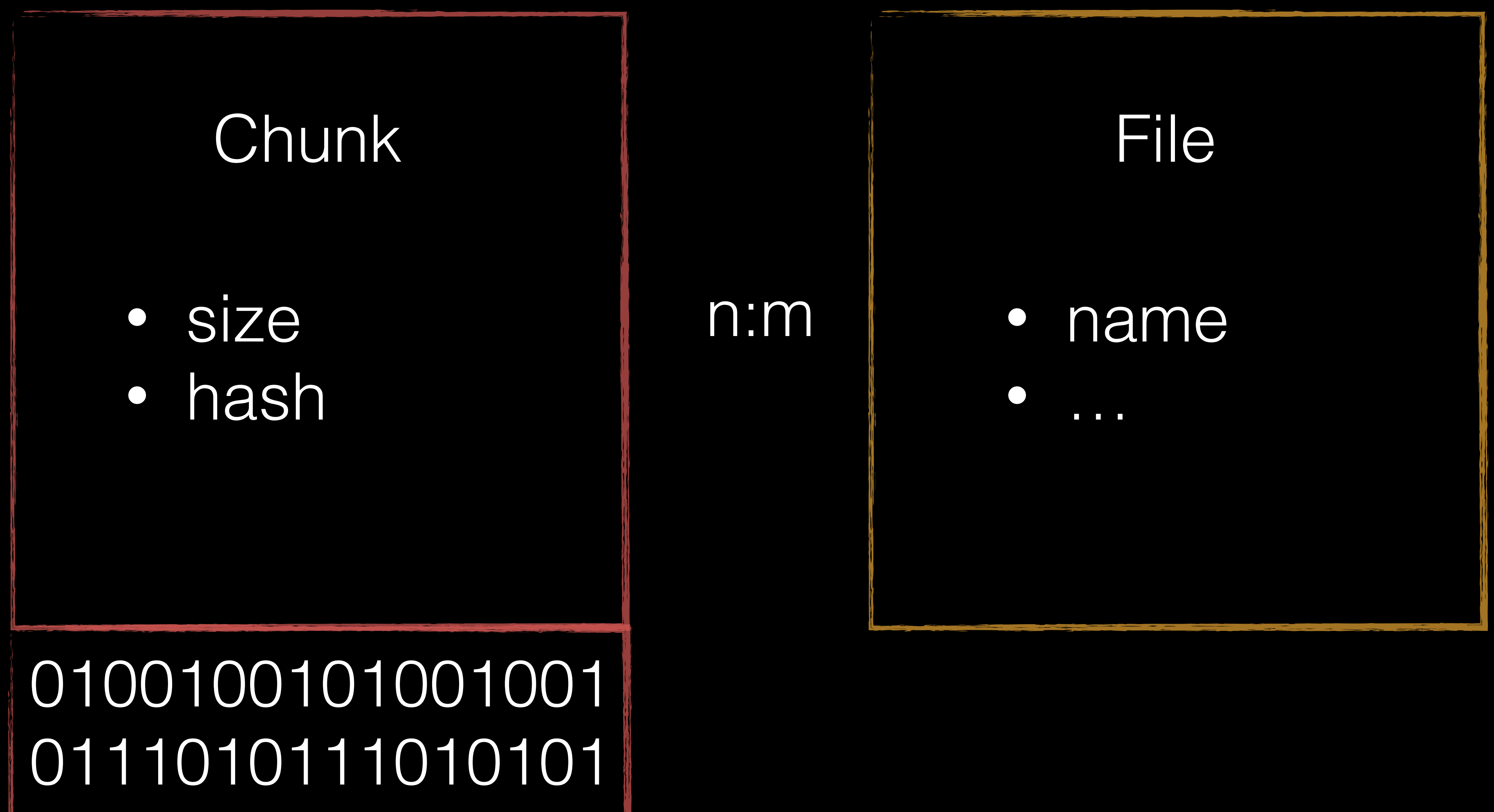
- Motivation
- Map-Reduce with Monoids
- Monoid Homomorphisms
- Hash Functions
- Cayley Graphs
- Practical Application and Results

Motivation

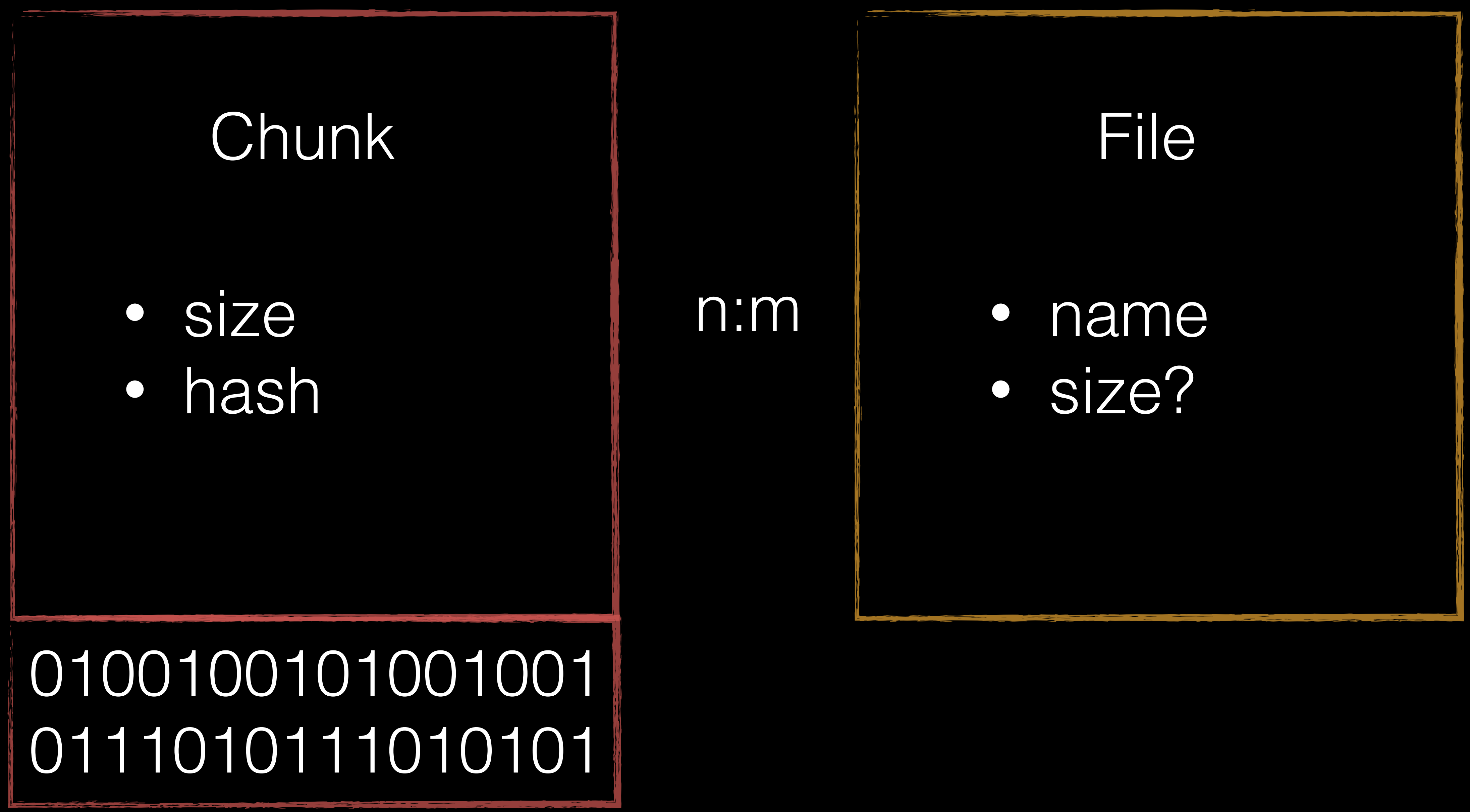
Distributed File Storage



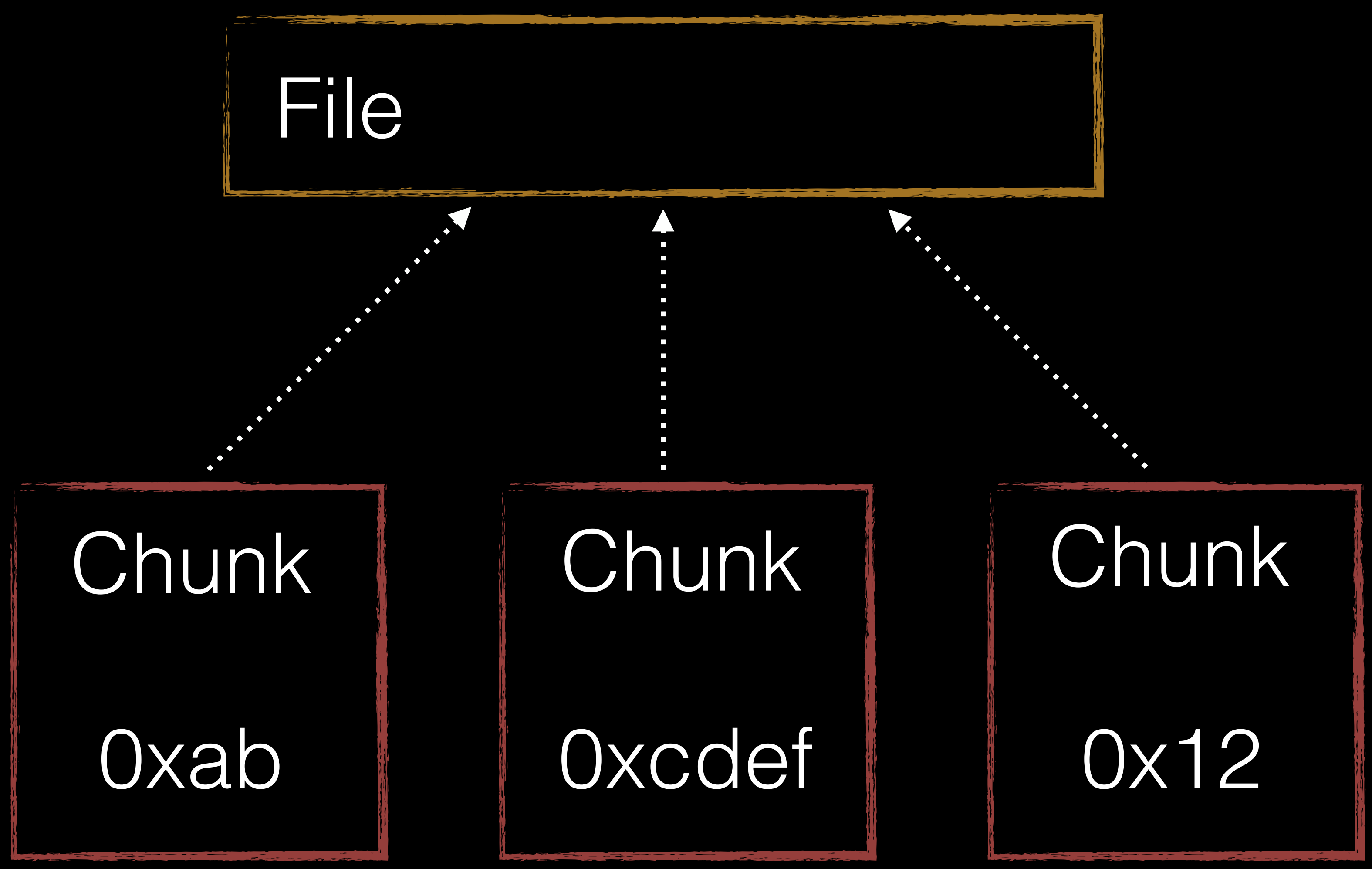
Distributed File Storage



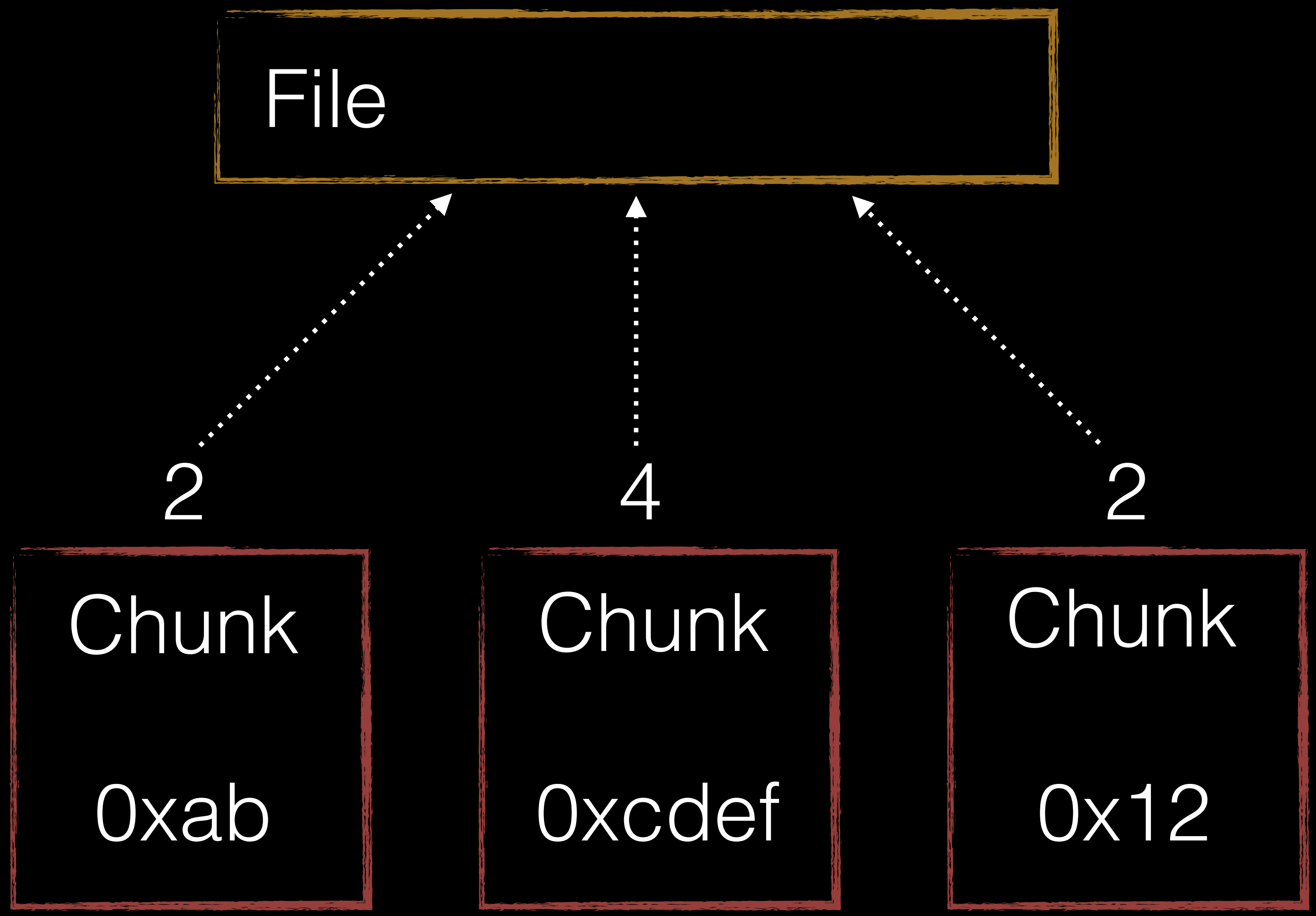
Distributed File Storage



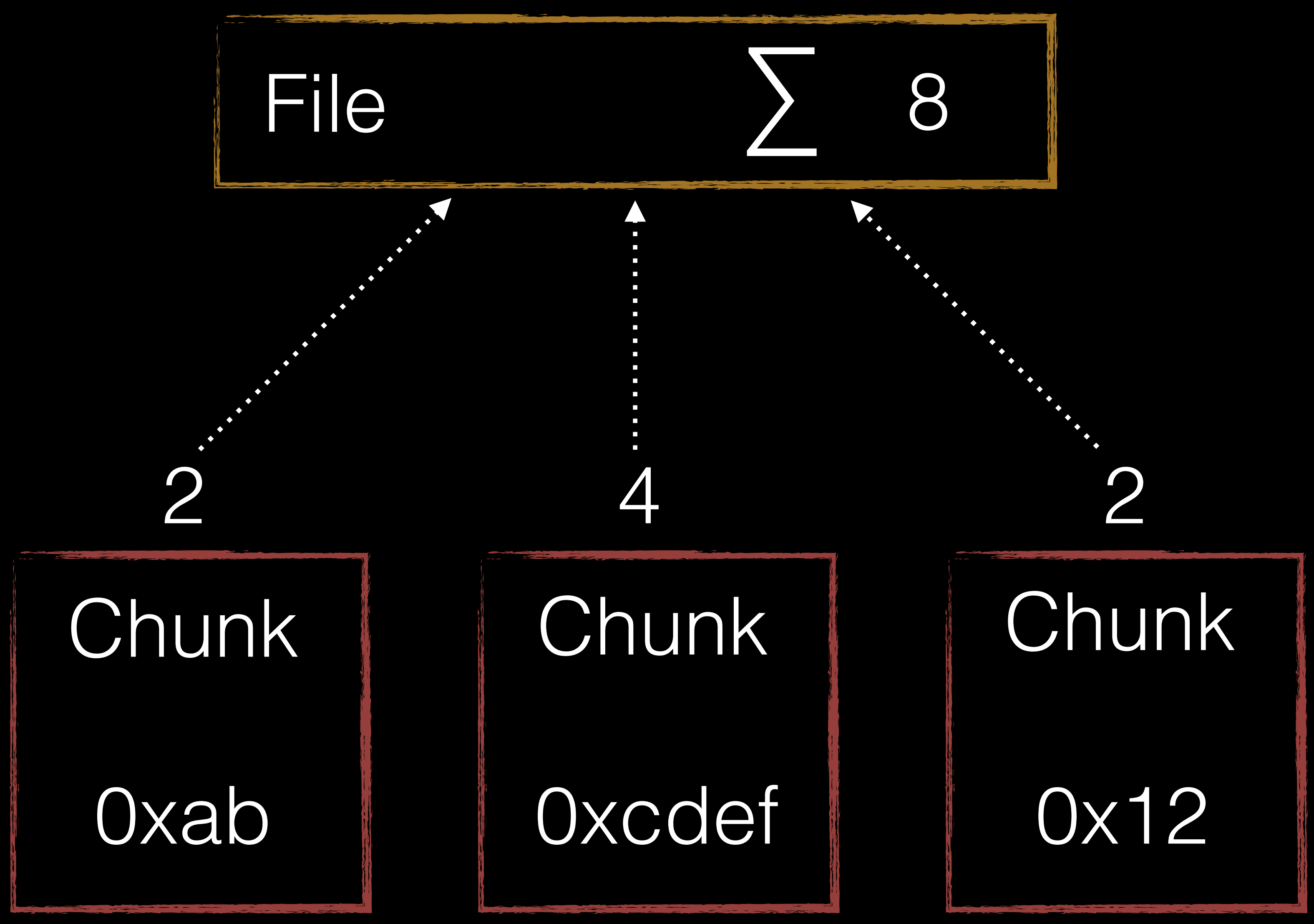
Distributed File Storage



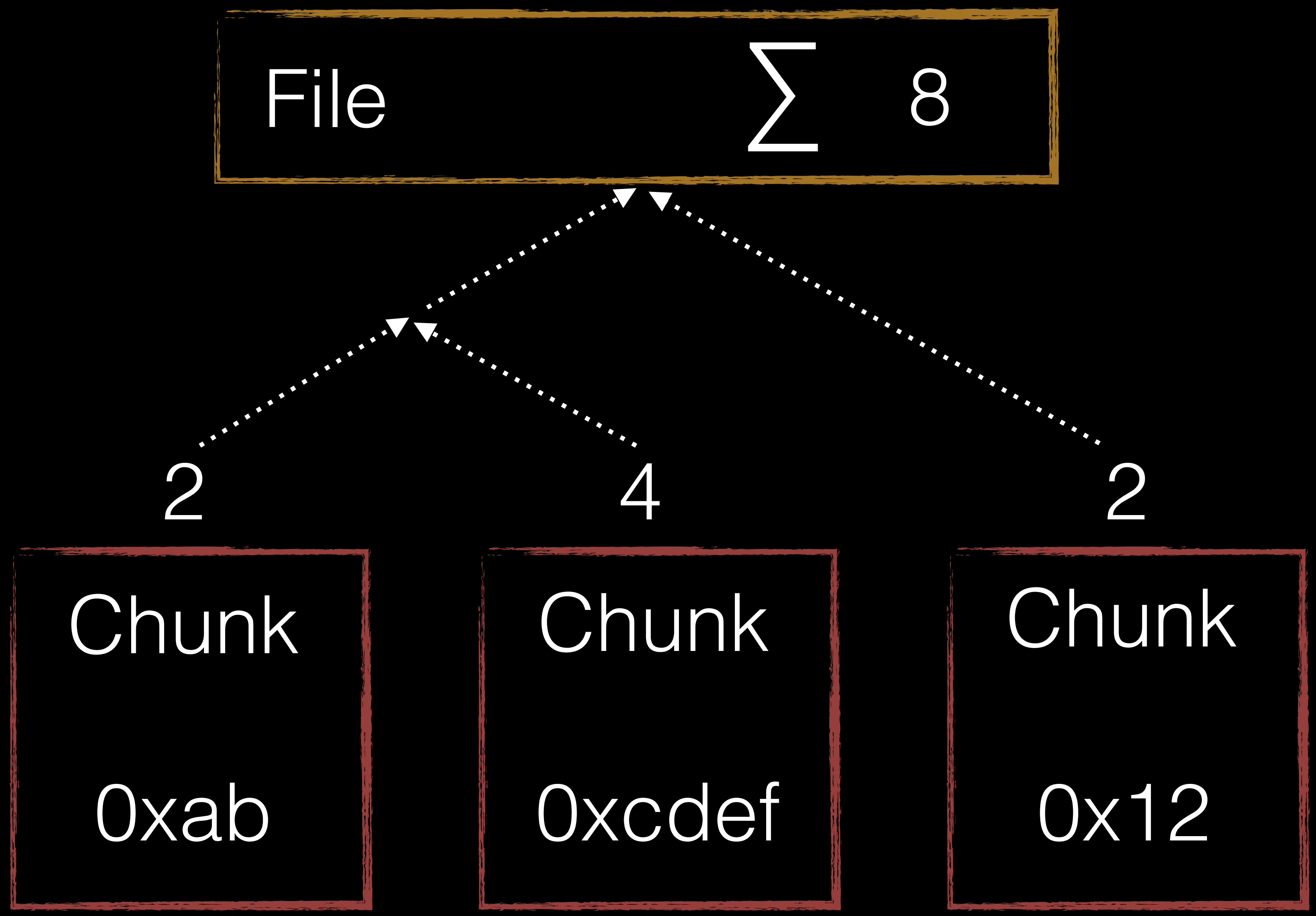
Distributed File Storage



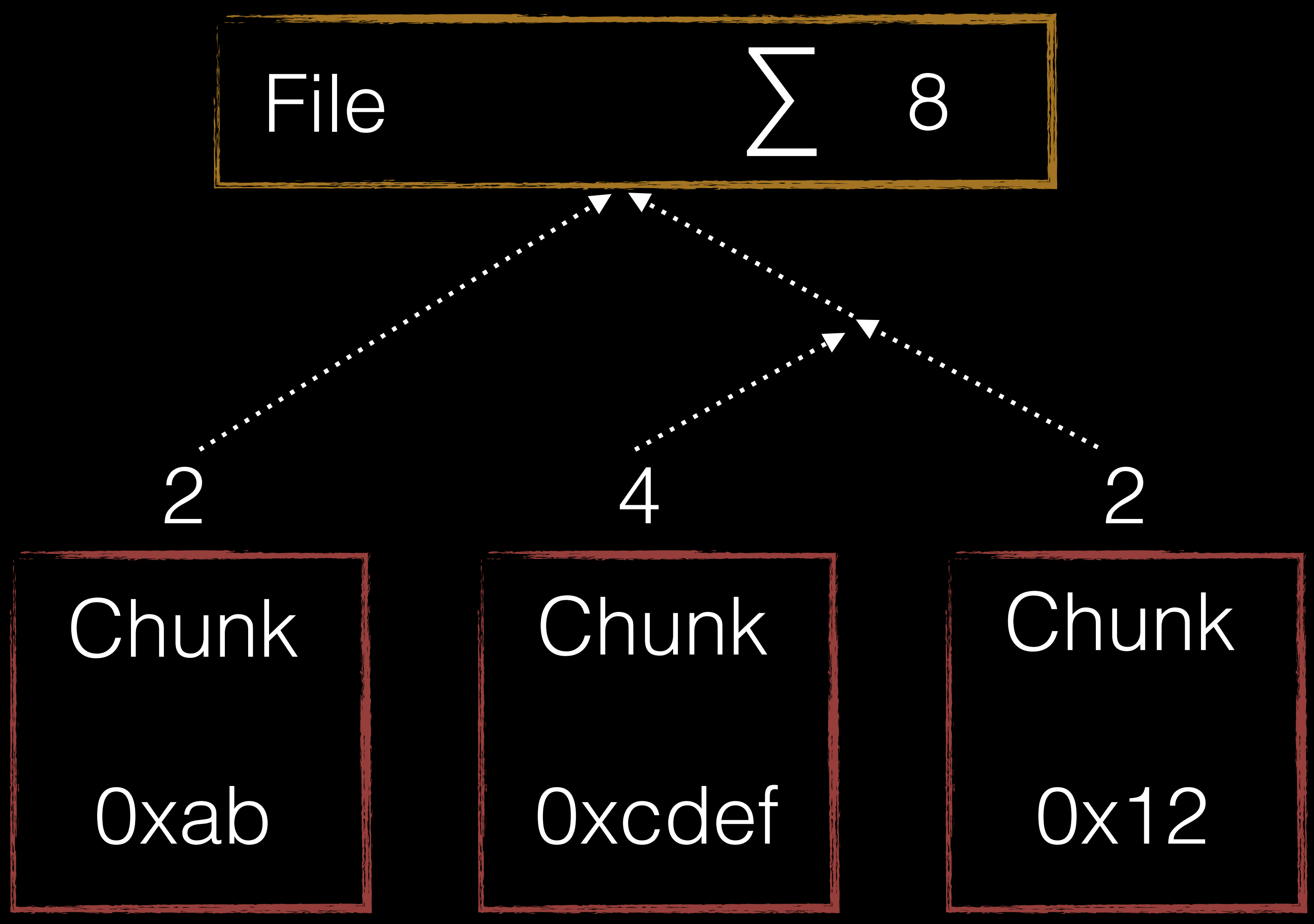
Distributed File Storage



Distributed File Storage



Distributed File Storage



Distributed File Storage

File	$\sum$	0

Map-Reduce with Monoids

Monoids

Type `M`

Identity element

```
zero: M
```

Binary operation

```
|+|: (M, M) => M
```

Monoid Laws

Left identity

```
forAll { (x: M) => zero |+| x == x }
```

Right identity

```
forAll { (x: M) => x |+| zero == x }
```

Associativity

```
forAll { (x: M, y: M, z: M) =>
x |+| (y |+| z) == (x |+| y) |+| z }
```

Concatenation

Type `Chunk = Array[Byte]`

Identity element

```
def zero: Chunk = Array()
```

Binary operation

```
def |+|: (Chunk, Chunk) => Chunk = _++_
```

Concatenation

Left identity

```
forAll { (x: Chunk) => zero |+| x == x }
```

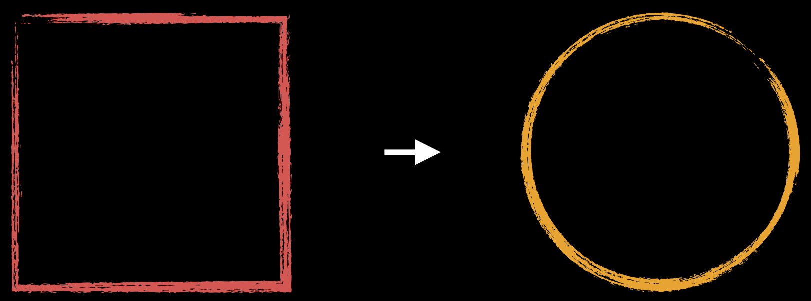
Right identity

```
forAll { (x: Chunk) => x |+| zero == x }
```

Associativity

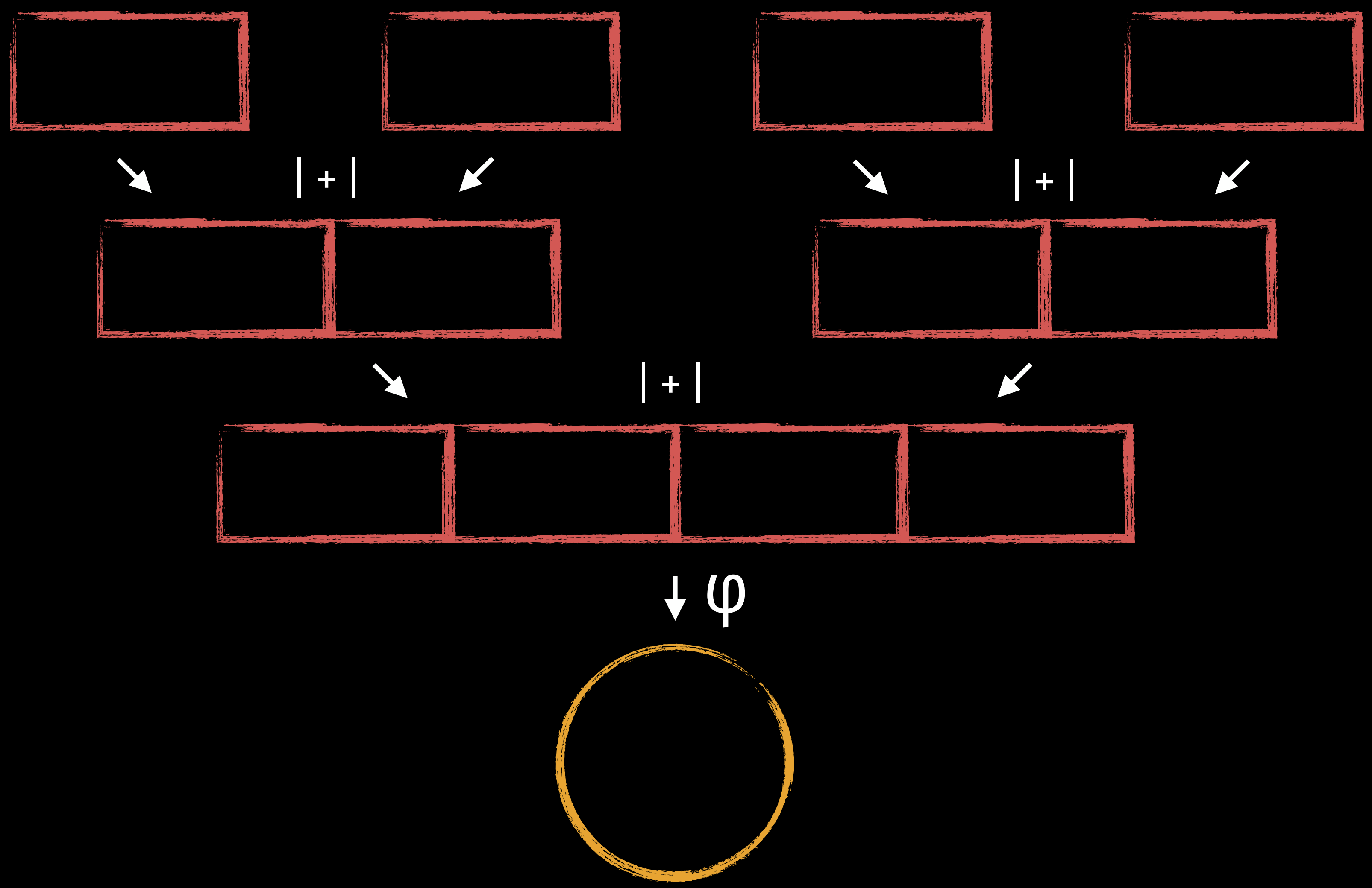
```
forAll { (x: Chunk, y: Chunk, z: Chunk) =>
  x |+| (y |+| z) == (x |+| y) |+| z }
```

Calculating size

φ: 

Projection `size: Chunk => Size`

Calculating size



Calculating size

Type `Size = Long`

Identity element

```
def zero: Size = 0
```

Binary operation

```
def |+|: (Size, Size) => Size = _+_
```

Calculating size

Left identity

```
forAll { (x: Size) => zero |+| x == x }
```

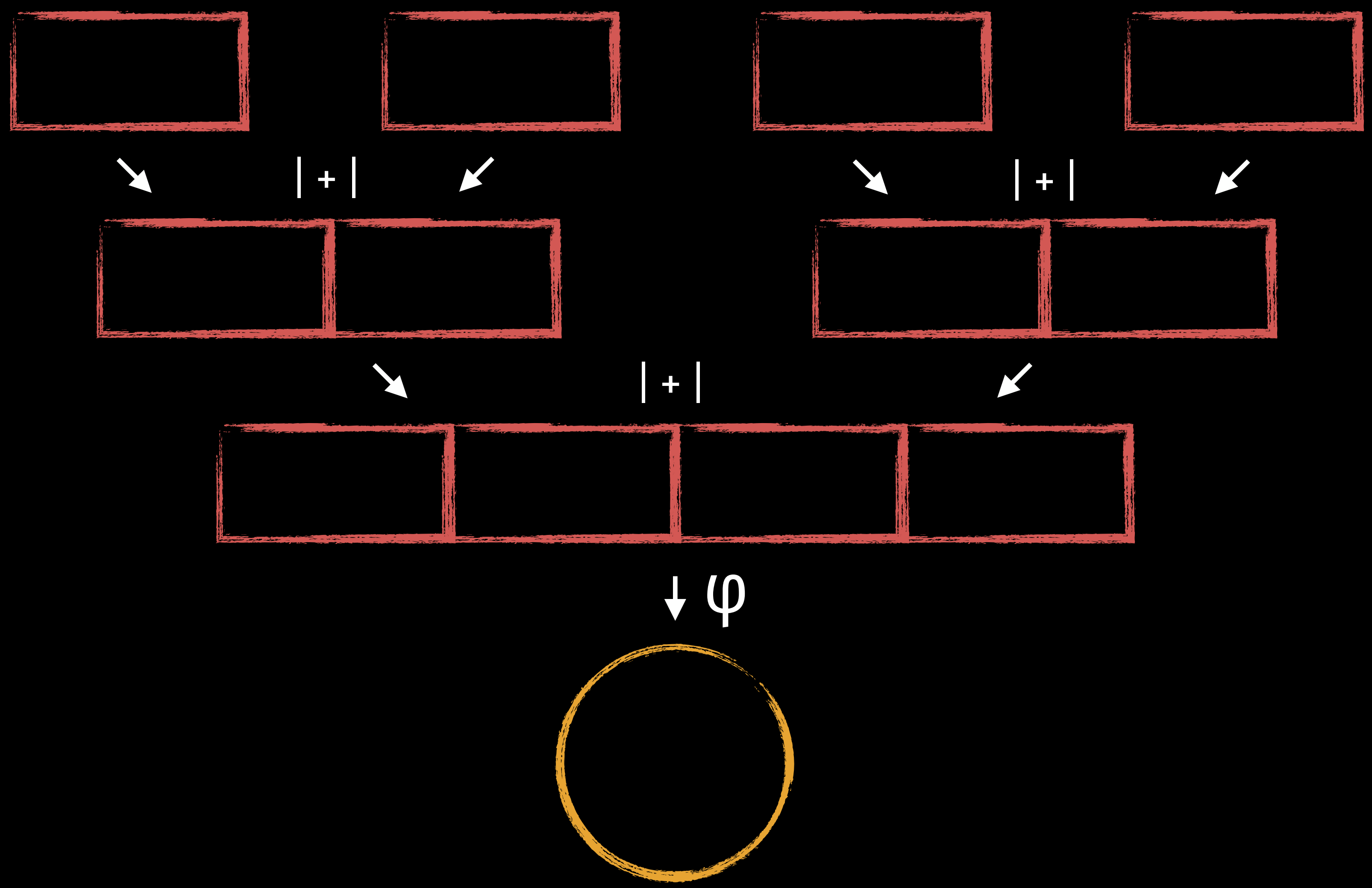
Right identity

```
forAll { (x: Size) => x |+| zero == x }
```

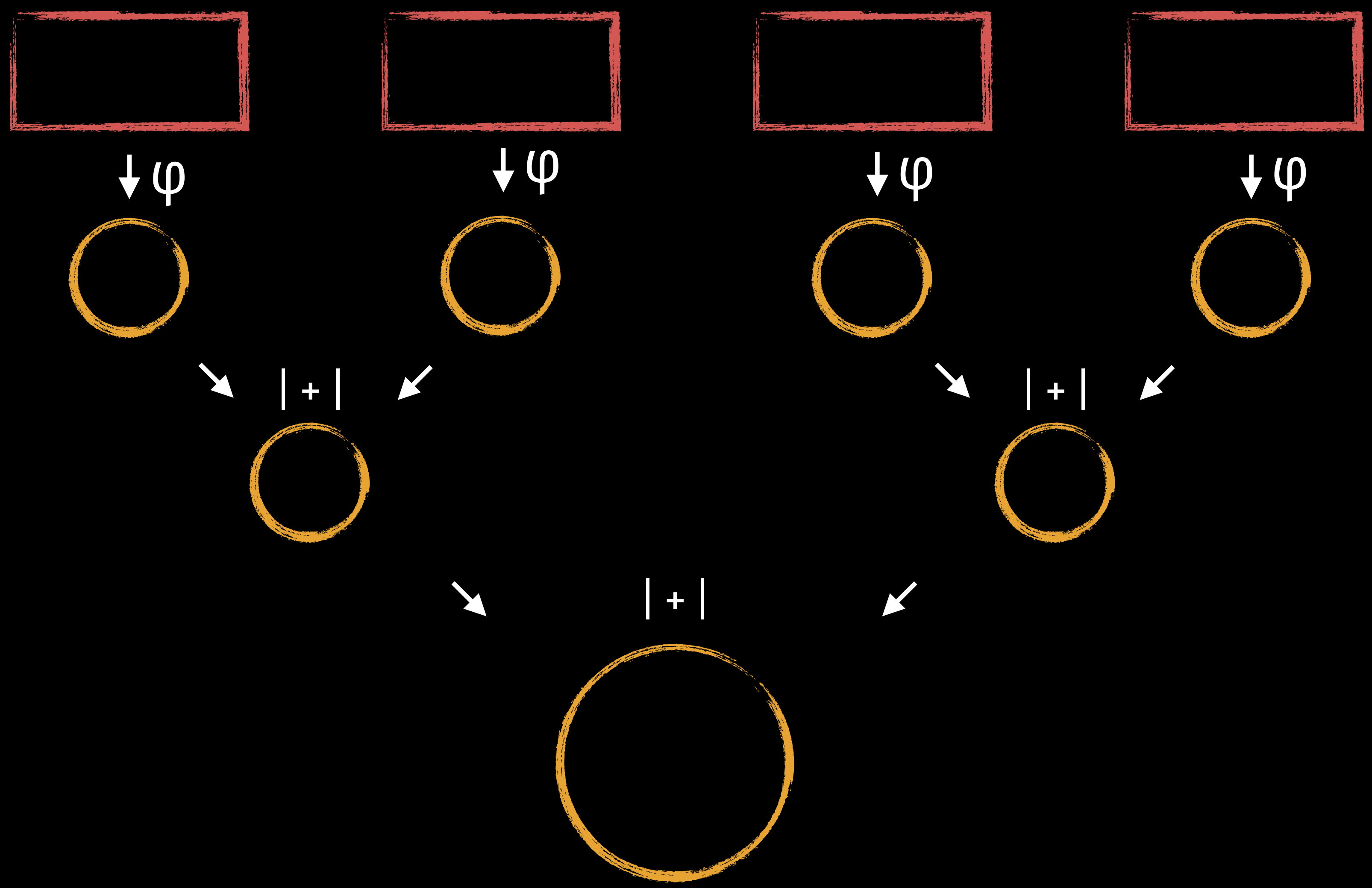
Associativity

```
forAll { (x: Size, y: Size, z: Size) =>
  x |+| (y |+| z) == (x |+| y) |+| z }
```

Calculating size



Calculating size



Monoid Homomorphisms

Monoid Homomorphisms

Monoid `(A, |+|`$_A$`, zero`$_A$`)`

Monoid `(B, |+|`$_B$`, zero`$_B$`)`

Morphism `φ: A => B`

Monoid Homomorphisms

Preserves identity

```
φ(zeroA) == zeroB
```

Preserves associativity

```
forAll { (x: A, y: A) =>
φ(x |+|A y) == φ(x) |+|B φ(y) }
```

Monoid Homomorphisms

Monoid `(Array[Byte], _++_, Array())`

Monoid `(Long, _+_, 0)`

Morphism `size: Array[Byte] => Long`

Monoid Homomorphisms

Preserves identity

```
size(Array()) == 0
```

Preserves associativity

```
forAll { (x: Chunk, y: Chunk) =>
size(x ++ y) == size(x) + size(y) }
```

Formulating the **reduction step** as a **lawful monoidal concatenation**, we can **ensure the correctness** of the map-reduce computation regardless of grouping and empty elements.

Formulating the **mapping step** as a **lawful monoid homomorphism**, we can **ensure** that while introducing optimisations we **maintain that correctness**.

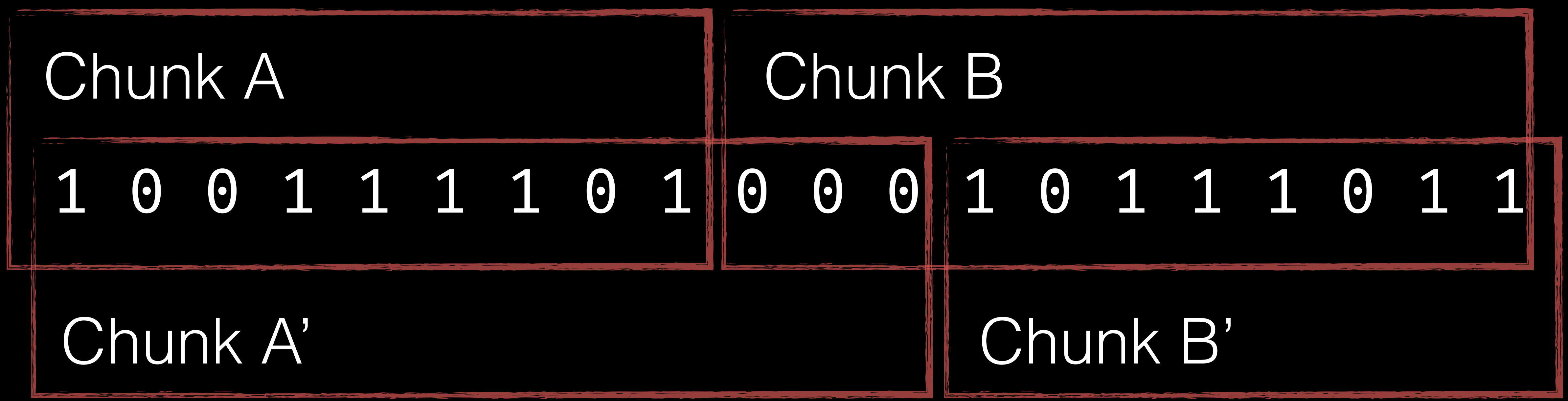
Chunk Sequence Equivalence

```
1 0 0 1 1 1 1 0 1 0 0 0 1 0 1 1 1 0 1 1
```

Chunk Sequence Equivalence

Chunk A	Chunk B
1 0 0 1 1 1 1 0 1	0 0 0 1 0 1 1 1 0 1 1

Chunk Sequence Equivalence



Chunk Sequence Equivalence

Chunk A

Chunk B

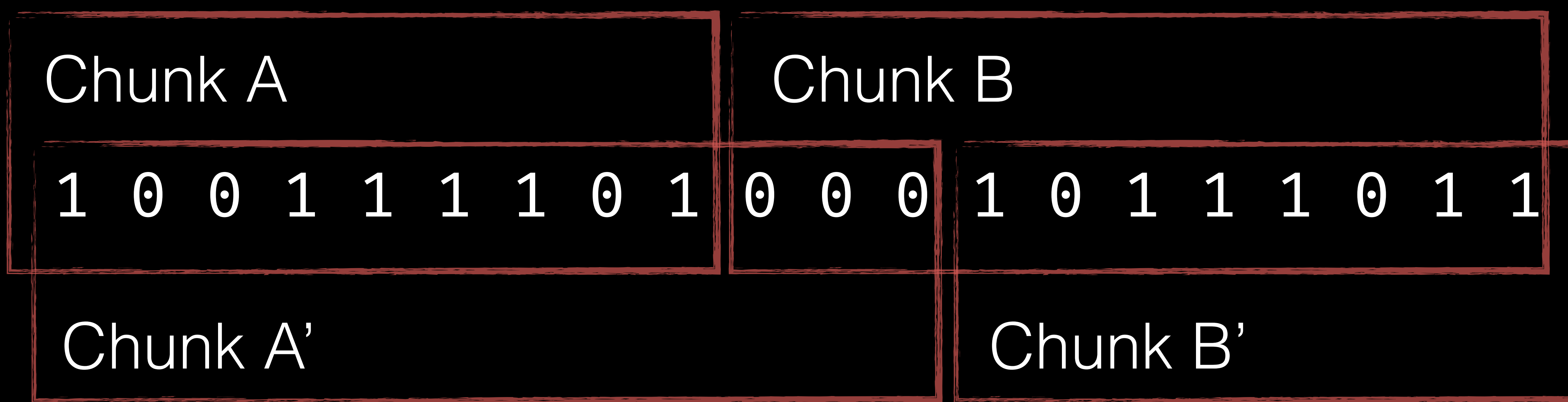
1 0 0 1 1 1 1 0 1 0 0 0 1 0 1 1 1 0 1 1

Chunk A’

Chunk B’

$$A \neq A'$$
$$B \neq B'$$
$$A + B = A' + B'$$

Chunk Sequence Equivalence



$$A \neq A'$$
$$B \neq B'$$
$$A + B = A' + B'$$

$$h(A) \neq h(A')$$
$$h(B) \neq h(B')$$
$$h(A + B) = h(A' + B')$$

Chunk Sequence Equivalence

Chunk A

Chunk B

`1 0 0 1 1 1 1 0 1 0 0 0 1 0 1 1 1 0 1 1`

Chunk A'

Chunk B'

```
A ≠ A'
B ≠ B'
A + B = A' + B'

h(A) ≠ h(A')
h(B) ≠ h(B')
h(A + B) = h(A' + B')
```

h(A) + h(B) = h(A') + h(B')

Hash Functions

SYSV $ sum

```
$ echo -n | sum -s

0 0

$ echo -n c | sum -s

99 1

$ echo -n ab | sum -s

195 1

$ echo -n abc | sum -s

294 1
```

Cryptographic Hash Functions

1. Collision resistance: it is hard to find inputs $x_1 \neq x_2$ such that $H(x_1) = H(x_2)$.

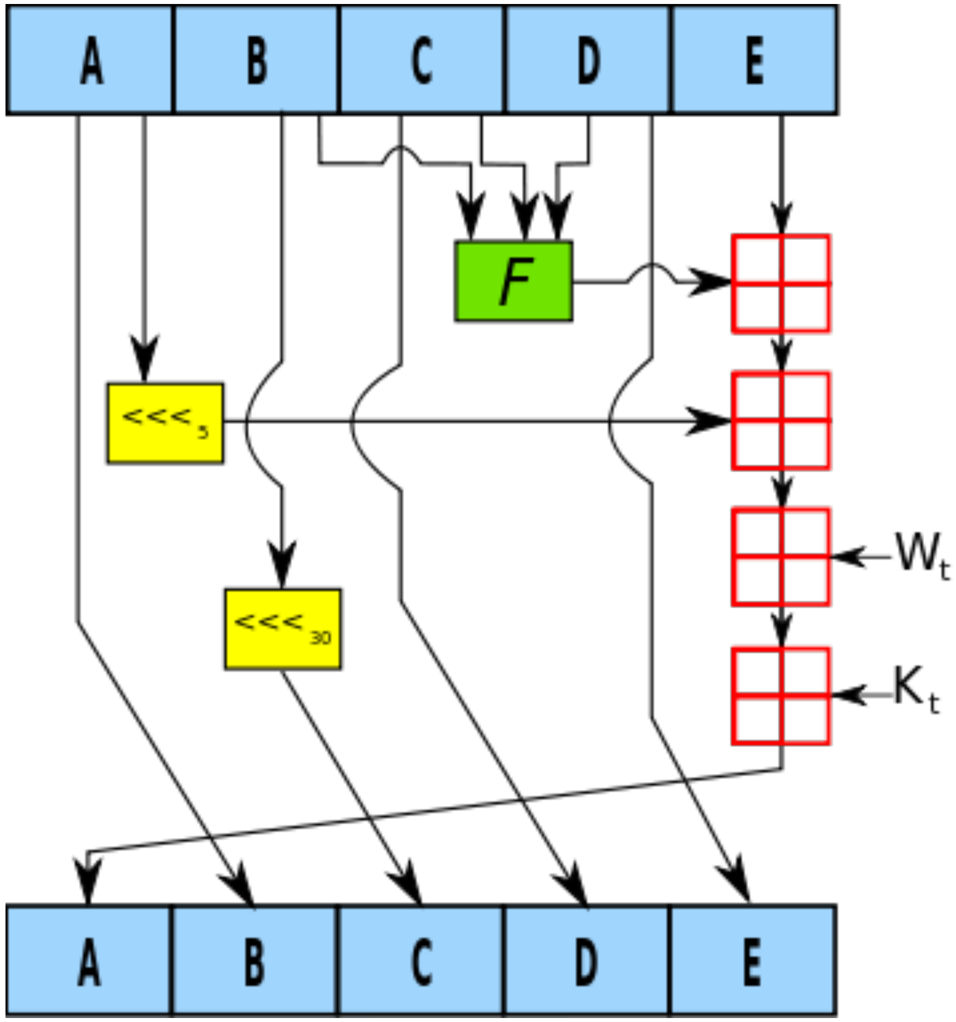
2. *S*econd preimage resistance: given input x_1, it is hard to find another input $x_2 \neq x_1$ such that $H(x_1) = H(x_2)$.

3. Preimage resistance: given output y, it is hard to find input x such that $H(x) = y$.

SHA1 $ shasum

```
$ echo -n | shasum

da39a3ee5e6b4b0d3255bfef95601890afd80709  -

$ echo -n c | shasum

84a516841ba77a5b4648de2cd0dfcb30ea46dbb4  -

$ echo -n ab | shasum

da23614e02469a0d7c7bd1bdab5c9c474b1904dc  -

$ echo -n abc | shasum

a9993e364706816aba3e25717850c26c9cd0d89d  -
```

SHA1 $ shasum



“The properties we
require are mainly a certain kind of weak
collision-freeness and some limited
“unpredictability.””

–Goldwasser, S. and Bellare, M., Lecture Notes on Cryptography

The **unlawfulness and complexity** of the compression function is the property at the **core of the security** of Merkle–Damgård constructions.

This construction is a **pinnacle of the imperative programming paradigm**. Introducing lots of state, and using low-level bit shuffling to solve a problem.

As **functional** programmers, we know the **value** of a **carefully constructed** solution that is based on **logically-founded laws**.

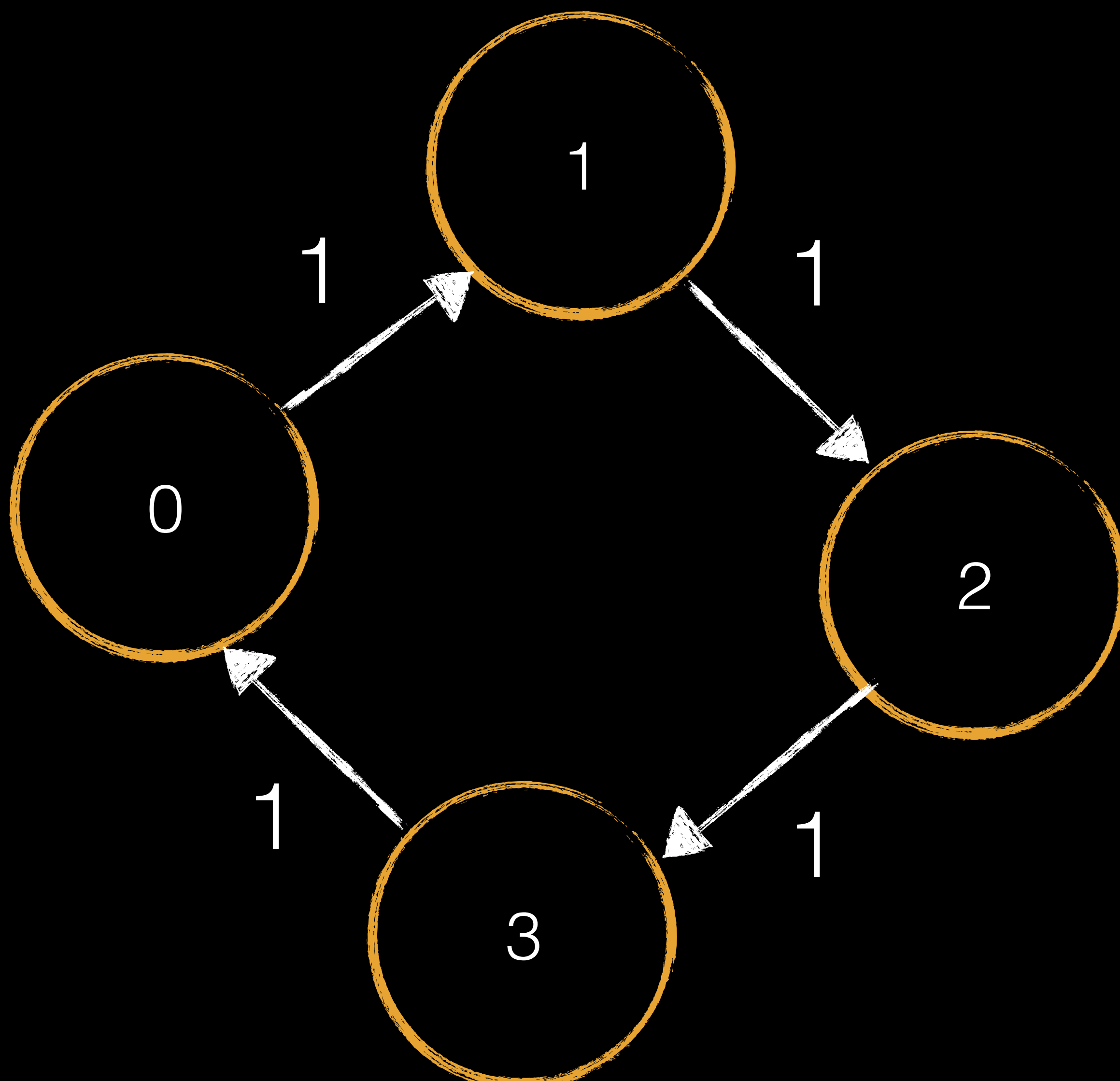
"… the SHA family […] [does] not really use any mathematical ideas to ensure the [desired] properties were met; the main idea was just to 'create a mess' by using complex iterations."

–Vladimir Shpilrain, 2015, Navigating in the Cayley graph of $SL_2(F_p)$ and applications to hashing

Cayley Graphs

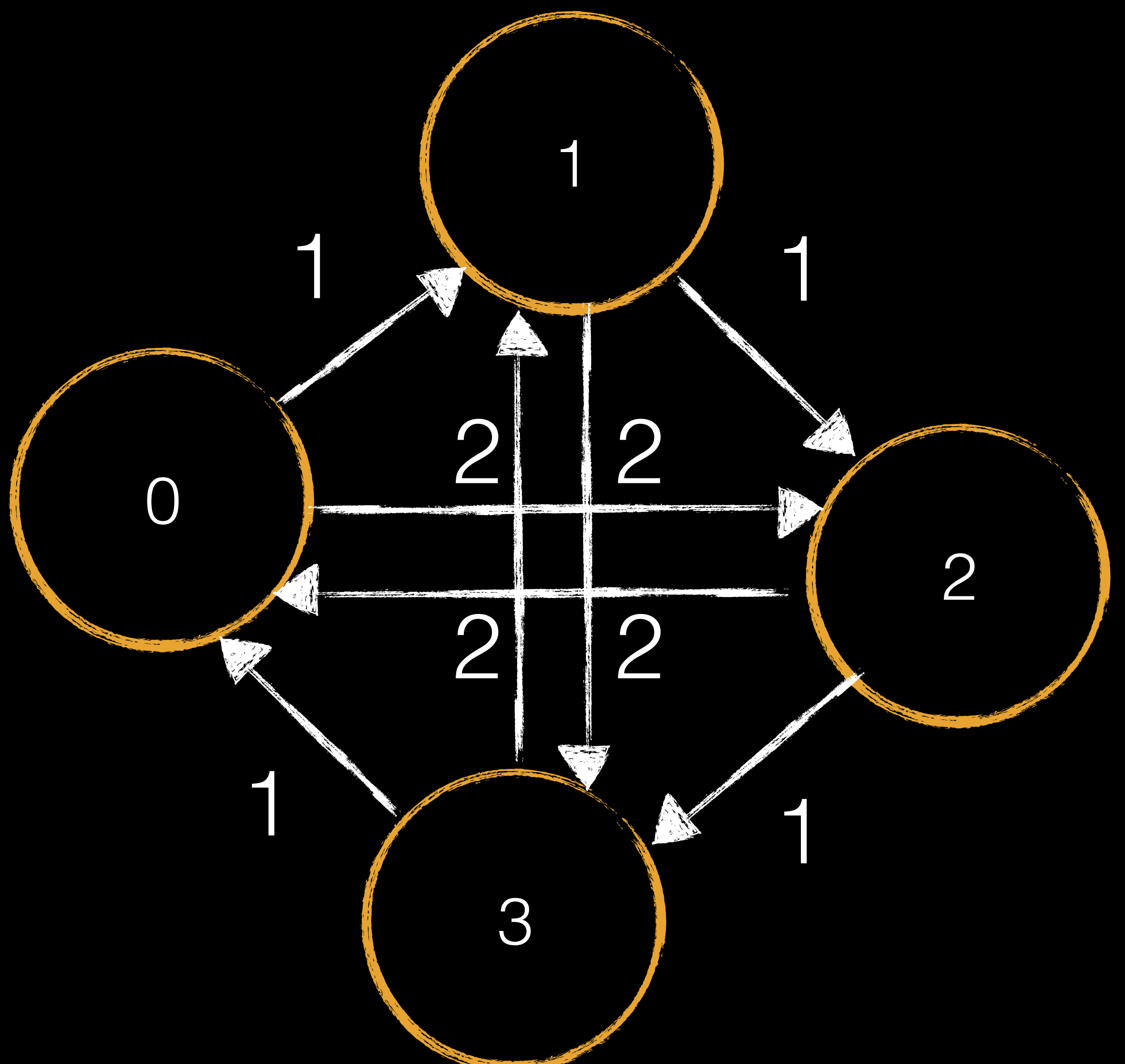
Cayley Graphs

$\Gamma(G = (\mathbb{N}/4, +), S = \{1\})$



Cayley Graphs

$\Gamma(G = (\mathbb{N}/4, +), S = \{1, 2\})$



Cayley Graphs

Associate:

$$\pi : \{0, 1\} \longrightarrow \{A, B\}$$

$$0 \longmapsto A$$

$$1 \longmapsto B$$

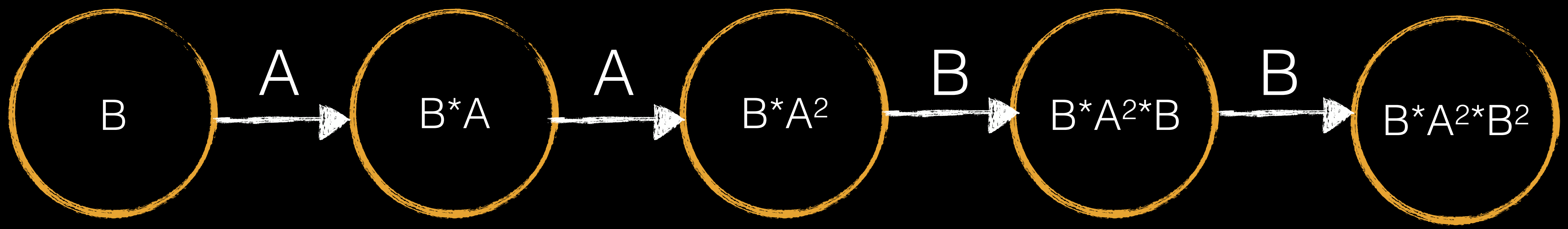
Then the hash code of a binary message $x_1x_2x\ldots x_n$ is the product:

$$\pi(x_1)\ \pi(x_2)\ \ldots\ \pi(x_n)$$

Cayley Graphs

Hashing a binary message represents a walk on the Cayley graph where the endpoint is the hash value.

Message: 10011



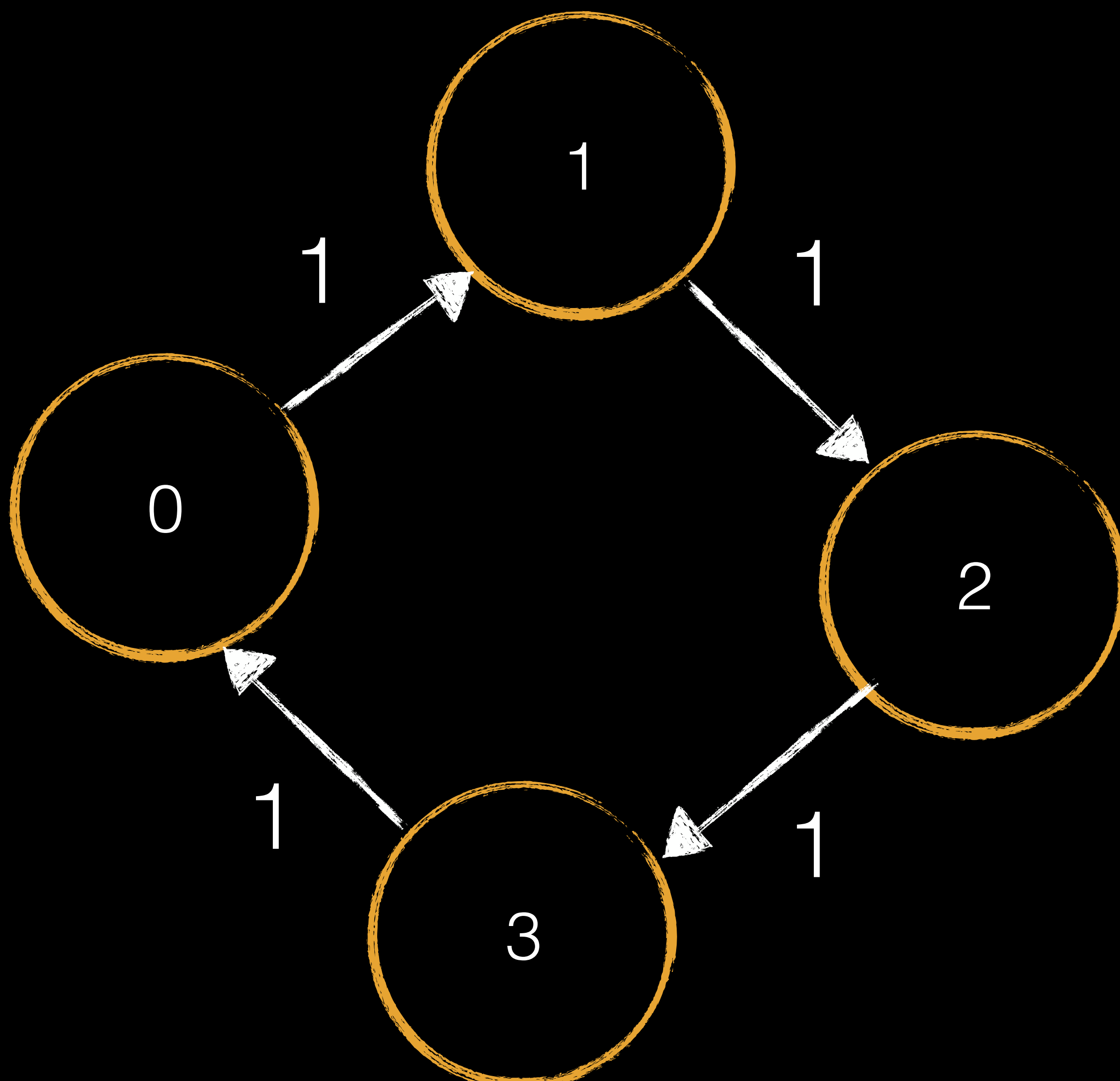
Cayley Graphs

Pick generators A and B of a semigroup S

…so that the Cayley graph generated by A and B has a large girth and therefore there will be no short relations (“collisions”).

Cayley Graphs

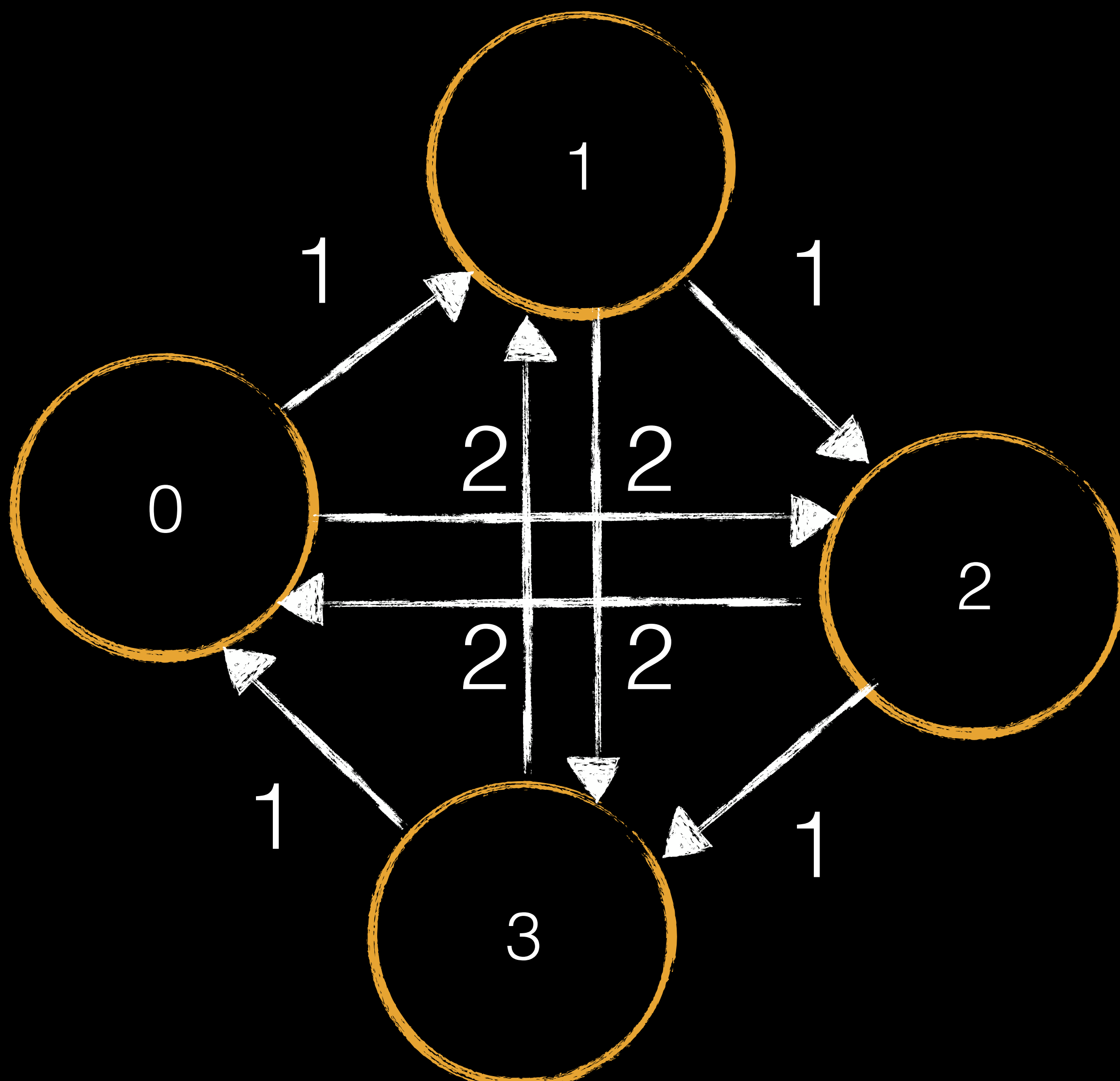
$\Gamma(G = (\mathbb{N}/4, +), S = \{1\})$



girth = 4

Cayley Graphs

$\Gamma(G = (\mathbb{N}/4, +), S = \{1, 2\})$



girth = 2

Cayley Graphs

What do we pick as our semigroup?

Needs to be **non-commutative** (otherwise collisions are trivial to produce).

Needs to be **hard to find inverses** (otherwise collisions are trivial to produce).

Hashing with SL_2

Let’s pick a finite binary field (modulo an irreducible polynomial):

$$F_{2^n} = F_2[X] / P_n(X)$$

Finding the inverse requires finding short factorisations in the finite group. This is know to be difficult for large cardinalities (Pspace-complete).

Finite field arithmetic is already used in real-world applications (CRC, ECC), and has hardware support on modern processors.

Hashing with SL_2

Let's pick the special linear group of matrices over that finite binary field:

$$A = \begin{pmatrix} X & 1 \\ 1 & 0 \end{pmatrix} \quad B = \begin{pmatrix} X & X+1 \\ 1 & 1 \end{pmatrix}$$

Matrix multiplication is non-commutative.

Embedding commutativity into non-commutativity has some cool properties to make cryptographic attacks harder.

Practical Application and Results

Demo

Results

- A hash function which is
  - strong
  - monoidal
    - composable
    - parallelisable

Performance

- Currently ~3x slower than OpenSSL SHA-256
- Producing 512 bit digests, so actually 1.5x faster per bit of output ;)
- And we can parallelise!
- Given a threaded runtime across >=4 cores, we can trivially chunk and hash a byte array in parallel, outperforming SHA-256

References

- **Paper**: https://www.rocq.inria.fr/secret/Jean-Pierre.Tillich/publications/HashingSL2.pdf
- **Scala** bindings: https://github.com/srijs/hwsl2-scala
- **Haskell** bindings: https://github.com/srijs/hwsl2-haskell
- Low-level **SIMD** code: https://github.com/srijs/hwsl2-core