A Piggybacking Design Framework for Read-and-Download-efficient Distributed Storage Codes

K. V. Rashmi, Nihar B. Shah, Kannan Ramchandran

Berkeley
UNIVERSITY OF CALIFORNIA

Outline

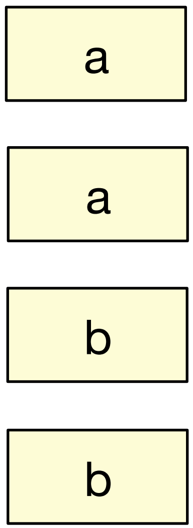
- Introduction & Motivation
  – Measurements from Facebook's Warehouse cluster
- The Piggybacking framework
- Via the Piggybacking framework
  – Best known codes for several settings
  – Comparison with other codes
  – Preliminary practical experiments
- Summary & future work

Outline

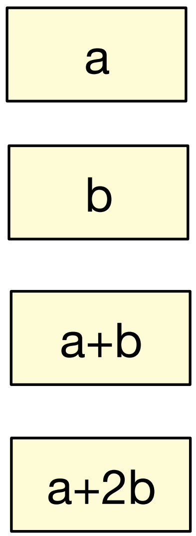
- Introduction & Motivation
  - Measurements from Facebook's Warehouse cluster
- The Piggybacking framework
- Via the Piggybacking framework
  - Best known codes for several settings
  - Comparison with other codes
  - Preliminary practical experiments
- Summary & future work

Redundancy: replication, erasure codes

- Redundancy for reliability & availability
- Replication: expensive for large scale data
- Erasure codes: storage-efficient



Replication

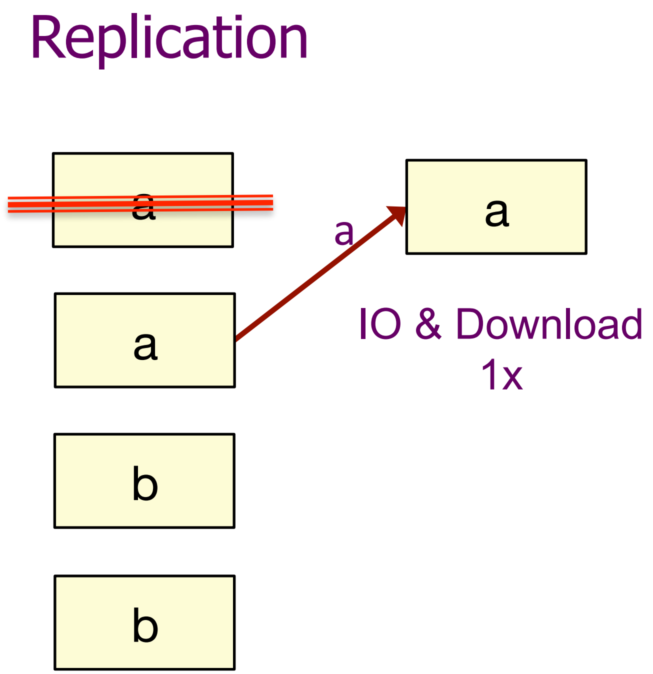


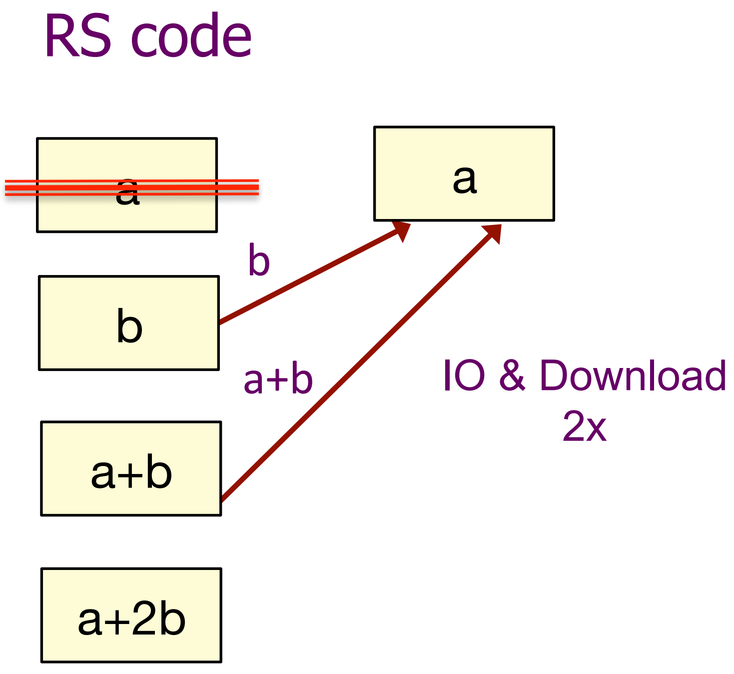
(4, 2) Reed-Solomon code

However…

RS codes increase disk IO and download during repair

Repair: increased disk IO & download





Typical RS repair:

IO & download = size of message

Motivation: Facebook's Warehouse Cluster Measurements

- Multiple tens of PBs and growing
- Multiple thousands of nodes

[Rashmi et al., USENIX HotStorage 2013]
"A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster"

Motivation: Facebook's Warehouse Cluster Measurements

- Multiple tens of PBs and growing
- Multiple thousands of nodes

Reducing storage requirements is of high importance

- Uses (14, 10) RS code for storage efficiency
  - on less-frequently accessed data
- Multiple PBs of RS coded data

[Rashmi et al., USENIX HotStorage 2013]
"A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster"

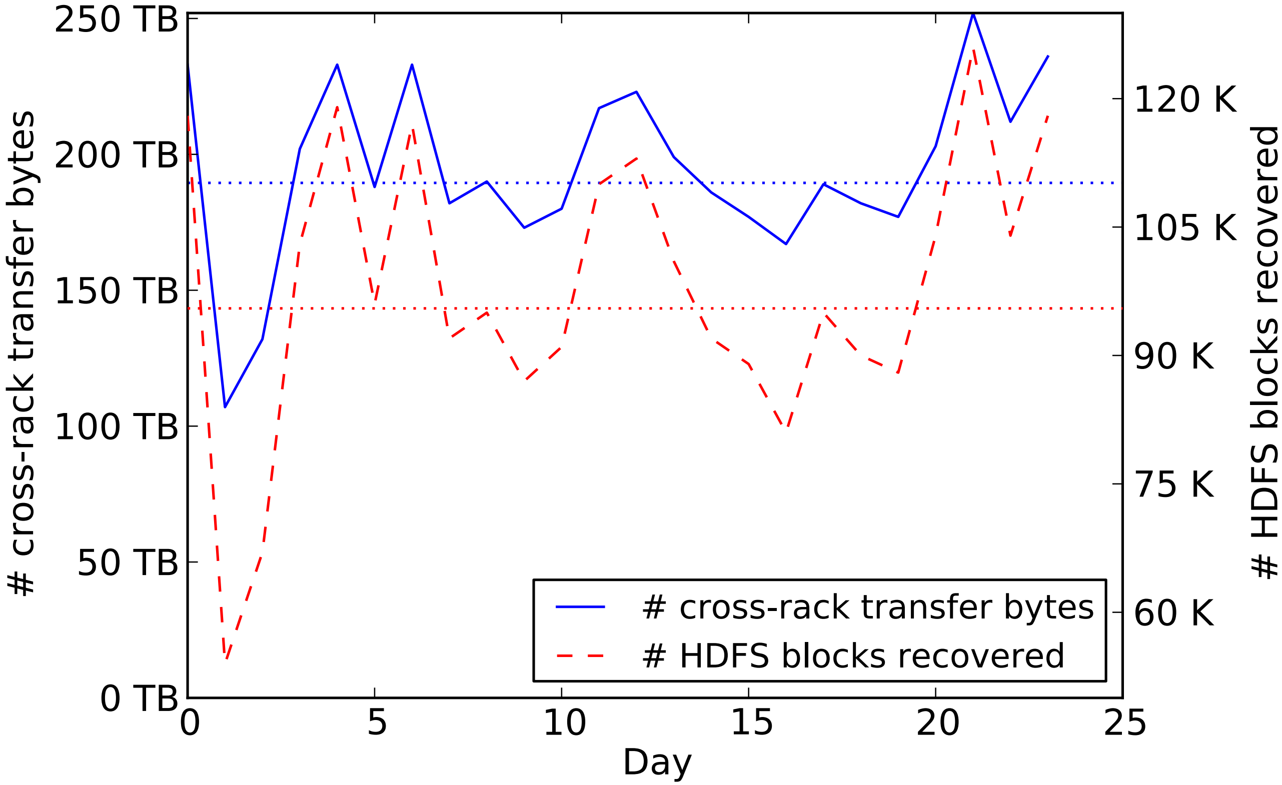
Breakdown of repairs

# repairs	% of repairs
1	98.08
2	1.87
3	0.036
4	9×10^{-6}
≥ 5	9×10^{-9}

Dominant scenario: Single repairs

Rashmi et al., USENIX HotStorage 2013

Amount of transfer



- Median of 180 TB transferred across racks per day for repair
- Around 5 times that under 3x replication

Rashmi et al., USENIX HotStorage 2013

Outline

- Introduction & Motivation
  - Measurements from Facebook's Warehouse cluster
- The Piggybacking framework
- Via the Piggybacking framework
  - Best known codes for several settings
  - Comparison with other codes
  - Preliminary practical experiments
- Summary & future work

Piggybacking RS codes: Toy Example

Step 1: Take 2 stripes of (4, 2) Reed-Solomon code

systematic 1	a_1	b_1
systematic 2	a_2	b_2
parity 1	a_1+a_2	b_1+b_2
parity 2	a_1+2a_2	b_1+2b_2

Piggybacking RS codes: Toy Example

Step 2: Add 'piggybacks'

a_1	b_1
a_2	b_2
a_1+a_2	b_1+b_2
a_1+2a_2	$b_1+2b_2+a_1$

No additional storage!

Fault-Tolerance

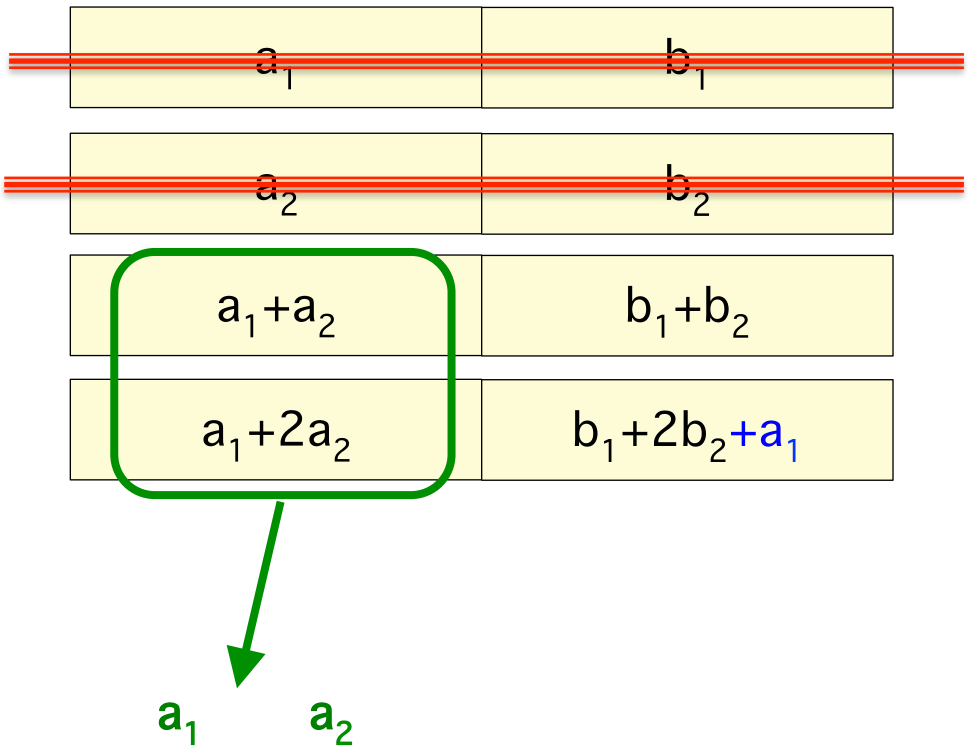
Same fault tolerance as RS code:

can tolerate any 2 failures

~~a_1~~	~~b_1~~
~~a_2~~	~~b_2~~
a_1+a_2	b_1+b_2
a_1+2a_2	$b_1+2b_2+a_1$

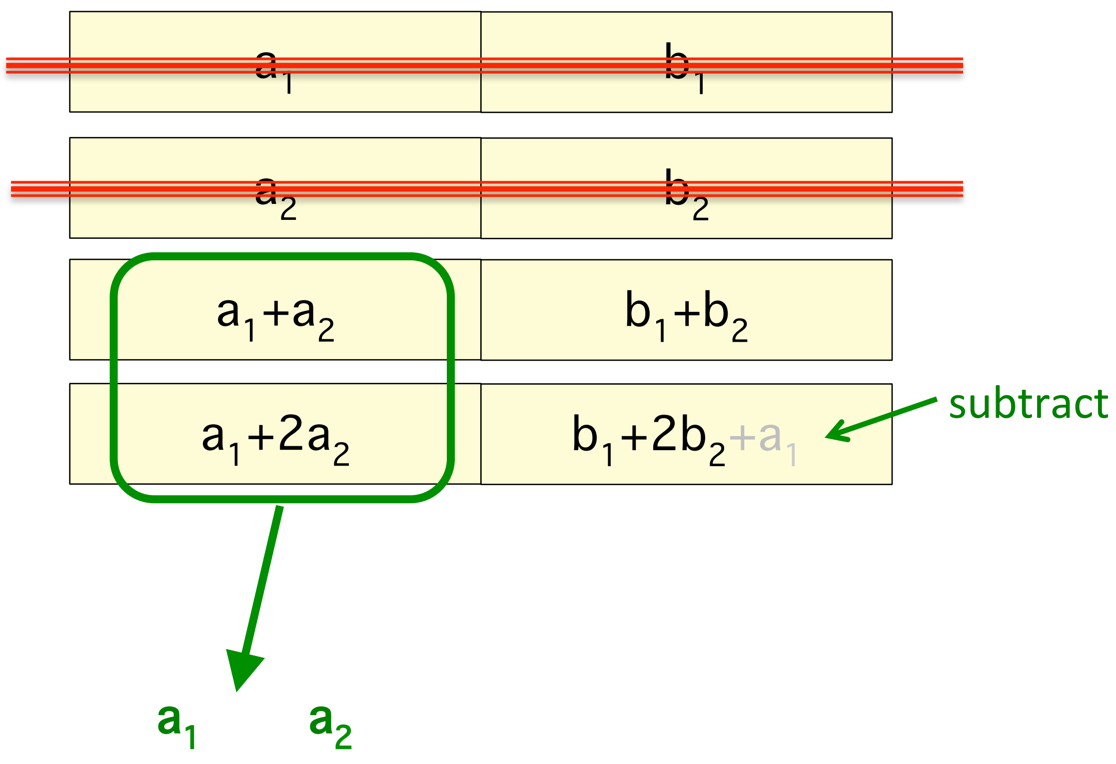
Fault-Tolerance

Same fault tolerance as RS code:
can tolerate failure of any 2 nodes



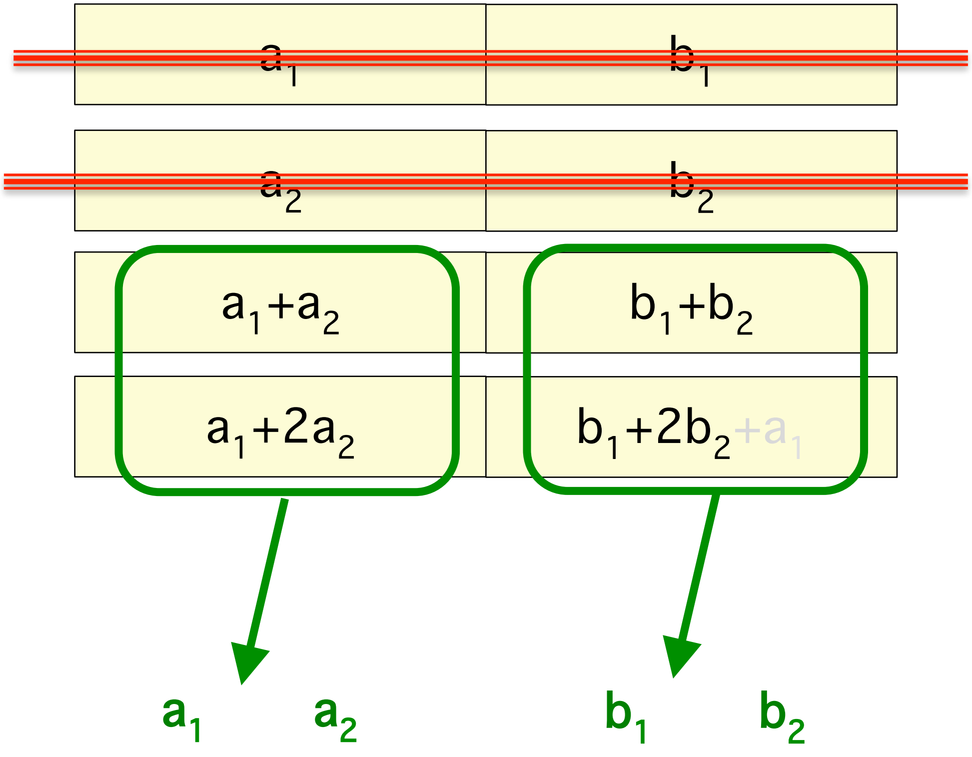
Fault-Tolerance

Same fault tolerance as RS code:
can tolerate failure of any 2 nodes



Fault-Tolerance

Same fault tolerance as RS code:
can tolerate failure of any 2 nodes

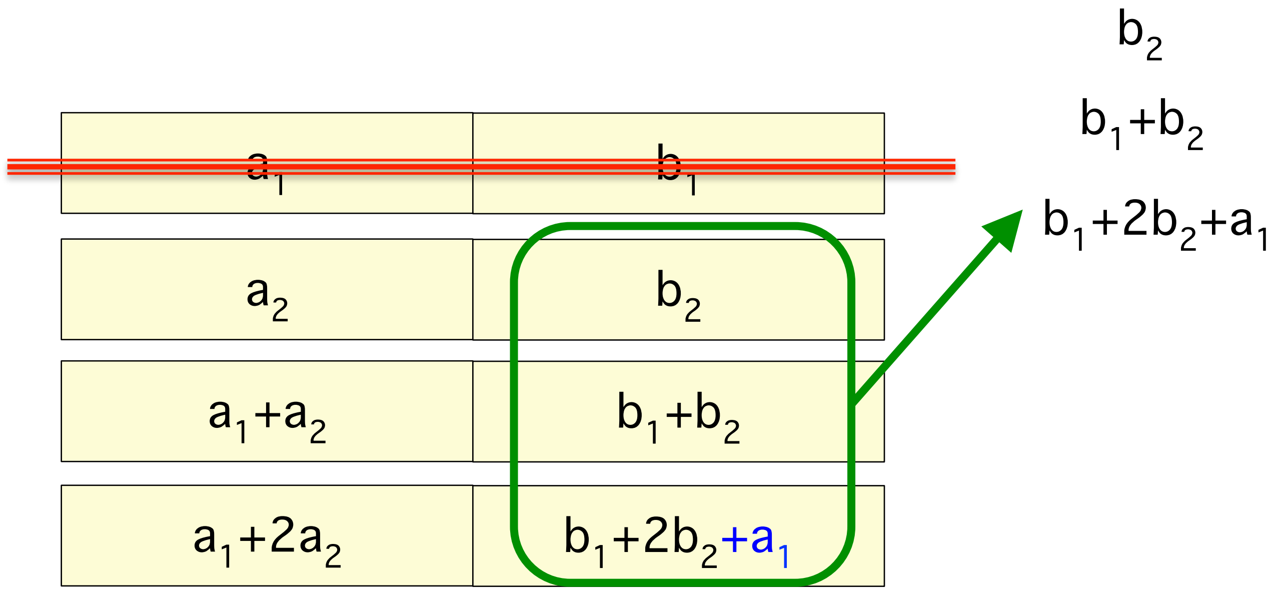


Repair

~~a_1~~	~~b_1~~
a_2	b_2
a_1+a_2	b_1+b_2
a_1+2a_2	$b_1+2b_2+a_1$

Repair

IO & Download = 3
(instead of 4 as in RS)



Repair

IO & Download = 3
(instead of 4 as in RS)

~~a_1~~	~~b_1~~
a_2	b_2
a_1+a_2	b_1+b_2
a_1+2a_2	$b_1+2b_2+a_1$

subtract

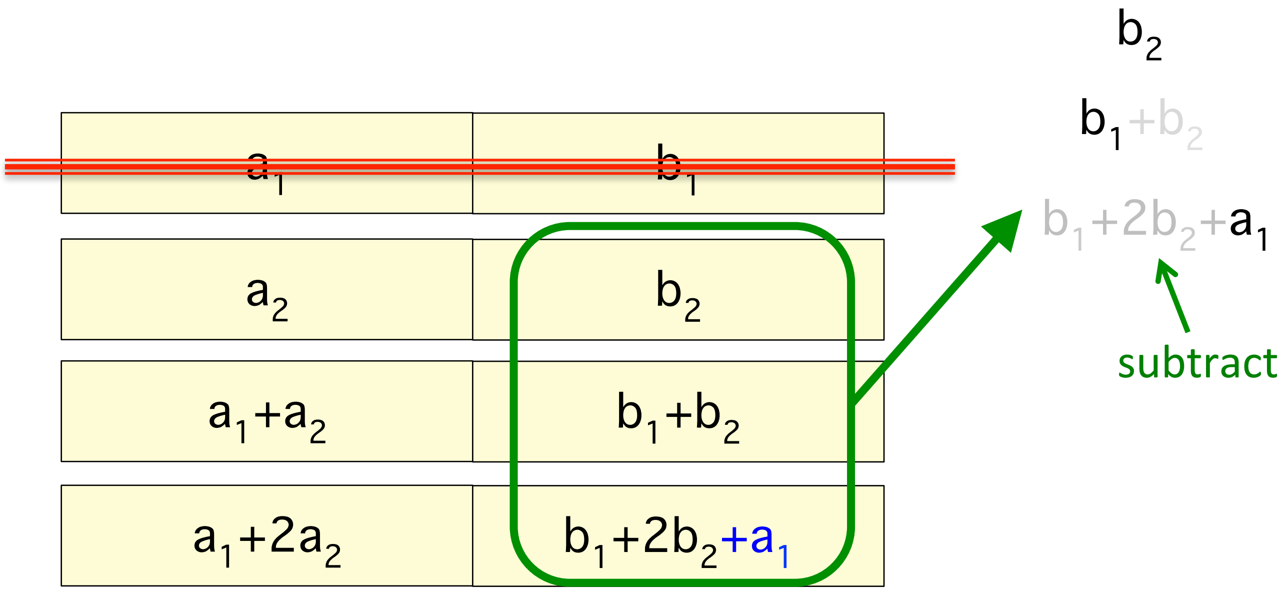
b_2

b_1+b_2

$b_1+2b_2+a_1$

Repair

IO & Download = 3
(instead of 4 as in RS)



General Piggybacking Framework

Step 1: Take 2 (or more) stripes of (n, k) code C

a_1	b_1
⋮	⋮
a_k	b_k
$f_1(a_1,\ldots,a_k)$	$f_1(b_1,\ldots,b_k)$
⋮	⋮
$f_{n-k}(a_1,\ldots,a_k)$	$f_{n-k}(b_1,\ldots,b_k)$

General Piggybacking Framework

Step 2: Add `Piggybacks’

a_1	b_1
$\vdots$	$\vdots$
a_k	b_k
$f_1(a_1,\ldots,a_k)$	$f_1(b_1,\ldots,b_k) + p_1(a_1,\ldots,a_k)$
$\vdots$	$\vdots$
$f_{n-k}(a_1,\ldots,a_k)$	$f_{n-k}(b_1,\ldots,b_k) + p_{n-k}(a_1,\ldots,a_k)$

General Piggybacking Framework

Decoding: use decoder of C

a_1	b_1
⋮	⋮
a_k	b_k
$f_1(a_1,\ldots,a_k)$	$f_1(b_1,\ldots,b_k) + p_1(a_1,\ldots,a_k)$
⋮	⋮
$f_{n-k}(a_1,\ldots,a_k)$	$f_{n-k}(b_1,\ldots,b_k) + p_{n-k}(a_1,\ldots,a_k)$

recover $a_1,\ldots,a_k$ as in C

subtract piggybacks;
recover $b_1,\ldots,b_k$ as in C

General Piggybacking Framework

Piggybacking does not reduce minimum distance

∴ Can choose arbitrary functions for piggybacking

Theorem 1: Let $U_1,\ldots,U_\alpha$ be random variables corresponding to the messages associated to the α stripes of the base code. For $i \in \{1,\ldots,n\}$, let X_i denote the (encoded) data stored in node i under the base code. Let Y_i denote the (encoded) data stored in node i upon piggybacking of that base code. Then for any subset of nodes $S \subseteq \{1,\ldots,n\}$,

$$I\left(\{Y_i\}_{i \in S}; U_1,\ldots,U_\alpha\right) \geq I\left(\{X_i\}_{i \in S}; U_1,\ldots,U_\alpha\right) .$$

General Piggybacking Framework

Piggybacking functions should be designed such that they can be used for repair

e.g.,

~~a_1~~	~~b_1~~
a_2	b_2
a_1+a_2	b_1+b_2
a_1+2a_2	$b_1+2b_2+a_1$

- We propose 3 designs of piggyback functions
  - details in the paper

Outline

- Introduction & Motivation
  - Measurements from Facebook's Warehouse cluster
- The Piggybacking framework
- Via the Piggybacking framework
  - Best known codes for several settings
  - Comparison with other codes
  - Preliminary practical experiments
- Summary & future work

Via the Piggybacking framework…

1. "Practical" High-rate MDS codes:

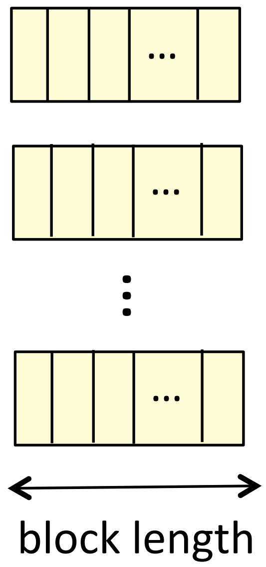
   Lowest known IO & download during repair

- Storage constrained systems: MDS & high-rate

- Then, why not high-rate Minimum Storage Regenerating (MSR) codes ?
  - Require block length exponential in k *(Tamo et al. 2011)*

Block length:

- number of sub-divisions of data units
- need high granularity of data
- low read efficiency



Comparison with High-rate MSR

n	k	Block length			IO & Download (% of message size)		
		RS	Piggy-RS	MSR	RS	Piggy-RS	MSR
16	14	1	4	128	100	77	54
25	22	1	4	3154	100	69	36
210	200	1	4	10^{20}	100	56	11

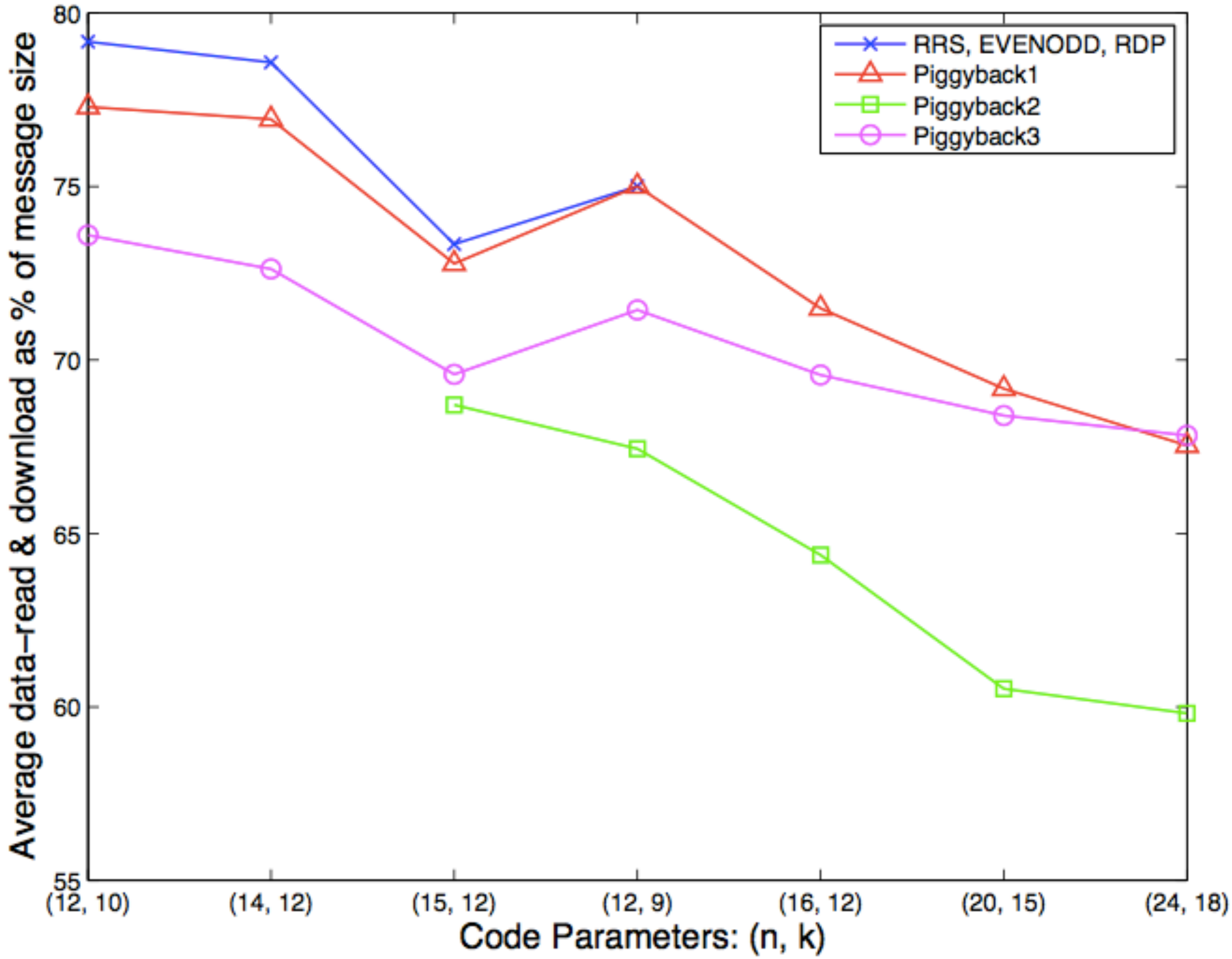
Comparison With Other Codes

Code	MDS	Parameters	Block length (in k)
High-rate MSR	Y	all	exponential
Product-matrix MSR etc.	Y	low rate	linear
Rotated-RS	Y	≤ 3 parities	constant
EVENODD/RDP	Y	≤ 2 parities	linear
MBR	N	all	linear
Local repair	N	all	constant
Piggyback	**Y**	**all**	**constant / linear**

Comparison With Other Codes

Code	MDS	Parameters	Block length (in k)
High-rate MSR	Y	all	exponential
Product-matrix MSR etc.	Y	low rate	linear
Rotated-RS	Y	≤ 3 parities	constant
EVENODD/RDP	Y	≤ 2 parities	linear
MBR	N	all	linear
Local repair	N	all	constant
Piggyback	**Y**	**all**	**constant / linear**

Comparison With Other Codes



Via the Piggybacking framework…

2 Binary MDS (vector) codes

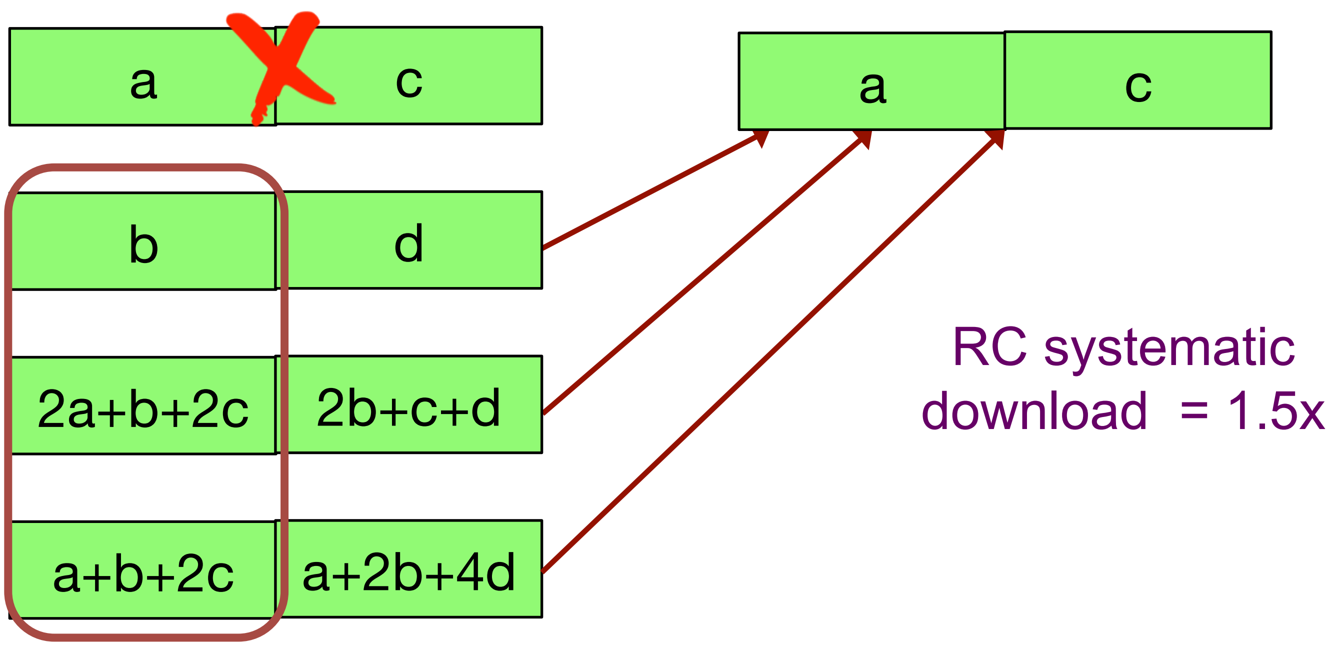
- lowest known IO & download during repair
- for all parameters where binary MDS (vector) codes exist
- (lowest when #parity ≥ 3;
  En Gad et al. ISIT 2013 for #parity=2)

Via the Piggybacking framework…

3. Enabling parity repair in regenerating codes designed for only systematic repair
   - efficiency in systematic repair retained
   - parity repair improved

Example…

- Regenerating code that repairs systematic nodes efficiently
- Parity node repair performed by downloading all data



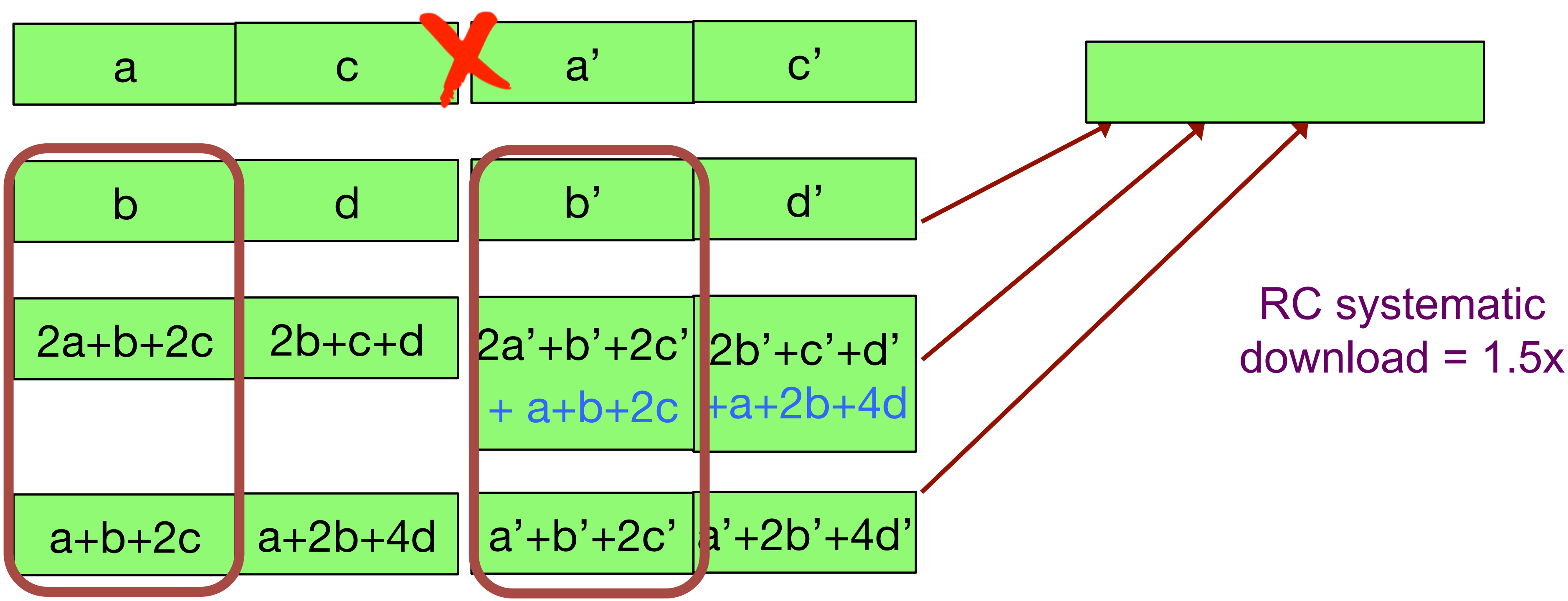
- Take two stripes of this code

a	c	a’	c’
b	d	b’	d’
2a+b+2c	2b+c+d	2a’+b’+2c’	2b’+c’+d’
a+b+2c	a+2b+4d	a’+b’+2c’	a’+2b’+4d’

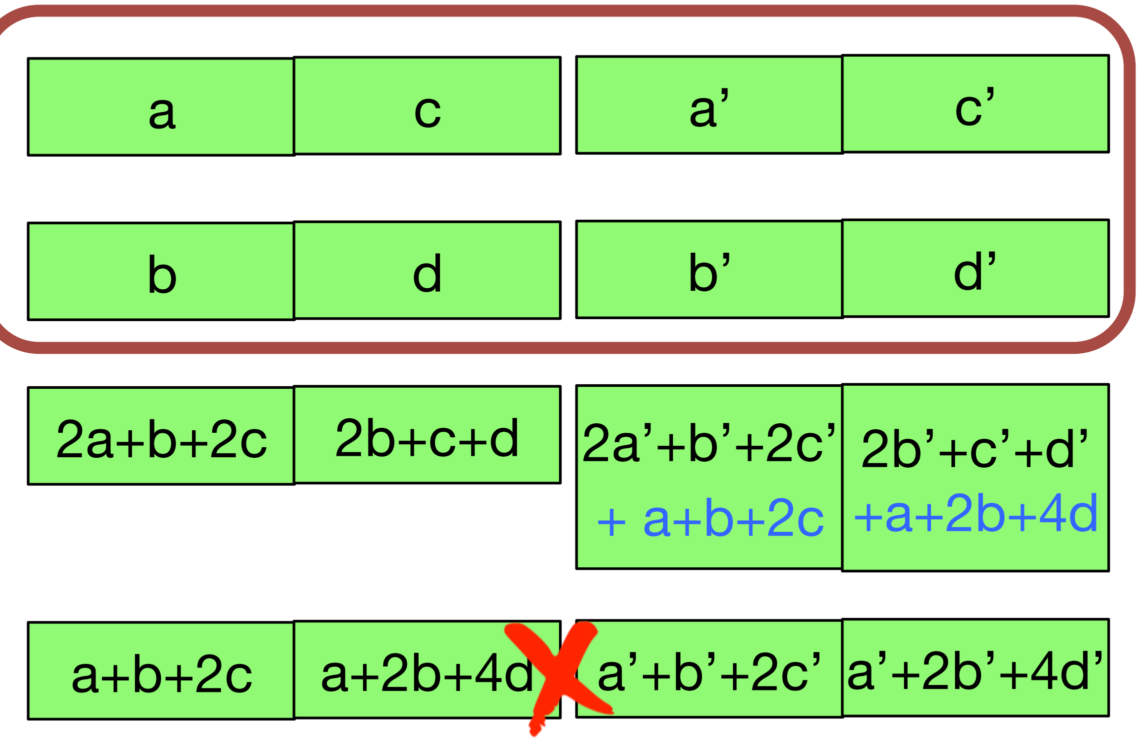
- Add Piggybacks of parities from first stripe onto second stripe

a	c	a’	c’
b	d	b’	d’
2a+b+2c	2b+c+d	2a’+b’+2c’ + a+b+2c	2b’+c’+d’ +a+2b+4d
a+b+2c	a+2b+4d	a’+b’+2c’	a’+2b’+4d’

- Systematic repair: same efficiency as original code (Piggyback can be subtracted off in the downloaded data)

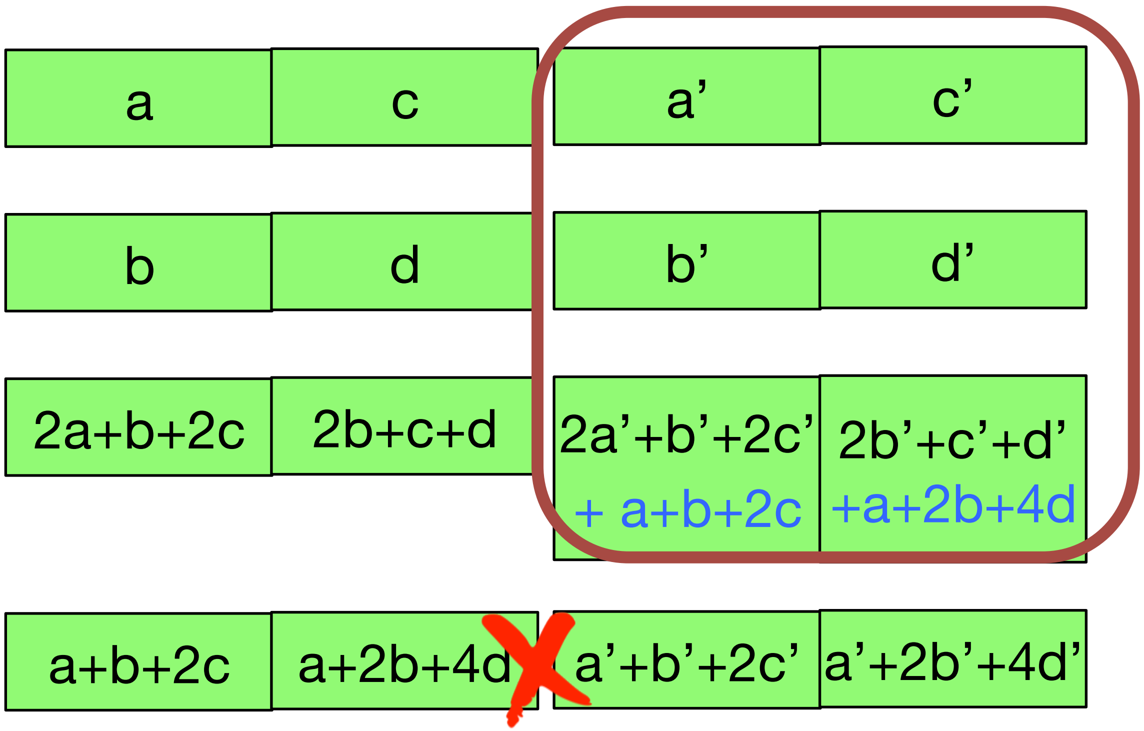


- Parity repair as in the original code requires 2x download



Original RC parity repair download = 2x

- Using the Piggybacks, need only 1.5x download



Second parity repair using Piggyback: download = 1.5x

Via the Piggybacking framework...

4 Currently implementing (14, 10) Piggyback-RS in HDFS

- 30% reduction in IO and download
- same storage & fault tolerance

(14,10) Piggybacked-RS code

Step 1: Take a (14, 10) Reed-Solomon code

a_1	b_1
⋮	⋮
a_{10}	b_{10}
$f_1(a_1,\ldots,a_{10})$	$f_1(b_1,\ldots,b_{10})$
$f_2(a_1,\ldots,a_{10})$	$f_2(b_1,\ldots,b_{10})$
$f_3(a_1,\ldots,a_{10})$	$f_3(b_1,\ldots,b_{10})$
$f_4(a_1,\ldots,a_{10})$	$f_4(b_1,\ldots,b_{10})$

(14, 10) Piggybacked-RS code

Step 2: Add `Piggybacks’

a_1	b_1
⋮	⋮
a_{10}	b_{10}
$f_1(a_1,\ldots,a_{10})$	$f_1(b_1,\ldots,b_{10})$
$f_2(a_1,\ldots,a_{10})$	$f_2(b_1,\ldots,b_{10}) + f_4(a_1,a_2,a_3,0,\ldots,0)$
$f_3(a_1,\ldots,a_{10})$	$f_3(b_1,\ldots,b_{10}) + f_4(0,\ldots,0,a_4,a_5,a_6,0,\ldots,0)$
$f_4(a_1,\ldots,a_{10})$	$f_4(b_1,\ldots,b_{10}) + f_4(0,\ldots,0,a_7,a_8,a_9,0)$

(14, 10) Piggybacked-RS code

Toleratesany 4 block failures

a_1	b_1
$\vdots$	$\vdots$
a_{10}	b_{10}
$f_1(a_1,\ldots,a_{10})$	$f_1(b_1,\ldots,b_{10})$
$f_2(a_1,\ldots,a_{10})$	$f_2(b_1,\ldots,b_{10}) + f_4(a_1,a_2,a_3,0,\ldots,0)$
$f_3(a_1,\ldots,a_{10})$	$f_3(b_1,\ldots,b_{10}) + f_4(0,\ldots,0,a_4,a_5,a_6,0,\ldots,0)$
$f_4(a_1,\ldots,a_{10})$	$f_4(b_1,\ldots,b_{10}) + f_4(0,\ldots,0,a_7,a_8,a_9,0)$

(14, 10) Piggybacked-RS code

Toleratess any 4 block failures

a_1	b_1
⋮	⋮
a_{10}	b_{10}
$f_1(a_1,\ldots,a_{10})$	$f_1(b_1,\ldots,b_{10})$
$f_2(a_1,\ldots,a_{10})$	$f_2(b_1,\ldots,b_{10}) + f_4(a_1,a_2,a_3,0,\ldots,0)$
$f_3(a_1,\ldots,a_{10})$	$f_3(b_1,\ldots,b_{10}) + f_4(0,\ldots,0,a_4,a_5,a_6,0,\ldots,0)$
$f_4(a_1,\ldots,a_{10})$	$f_4(b_1,\ldots,b_{10}) + f_4(0,\ldots,0,a_7,a_8,a_9,0)$

recover $a_1,\ldots,a_{10}$
like in RS

(14, 10) Piggybacked-RS code

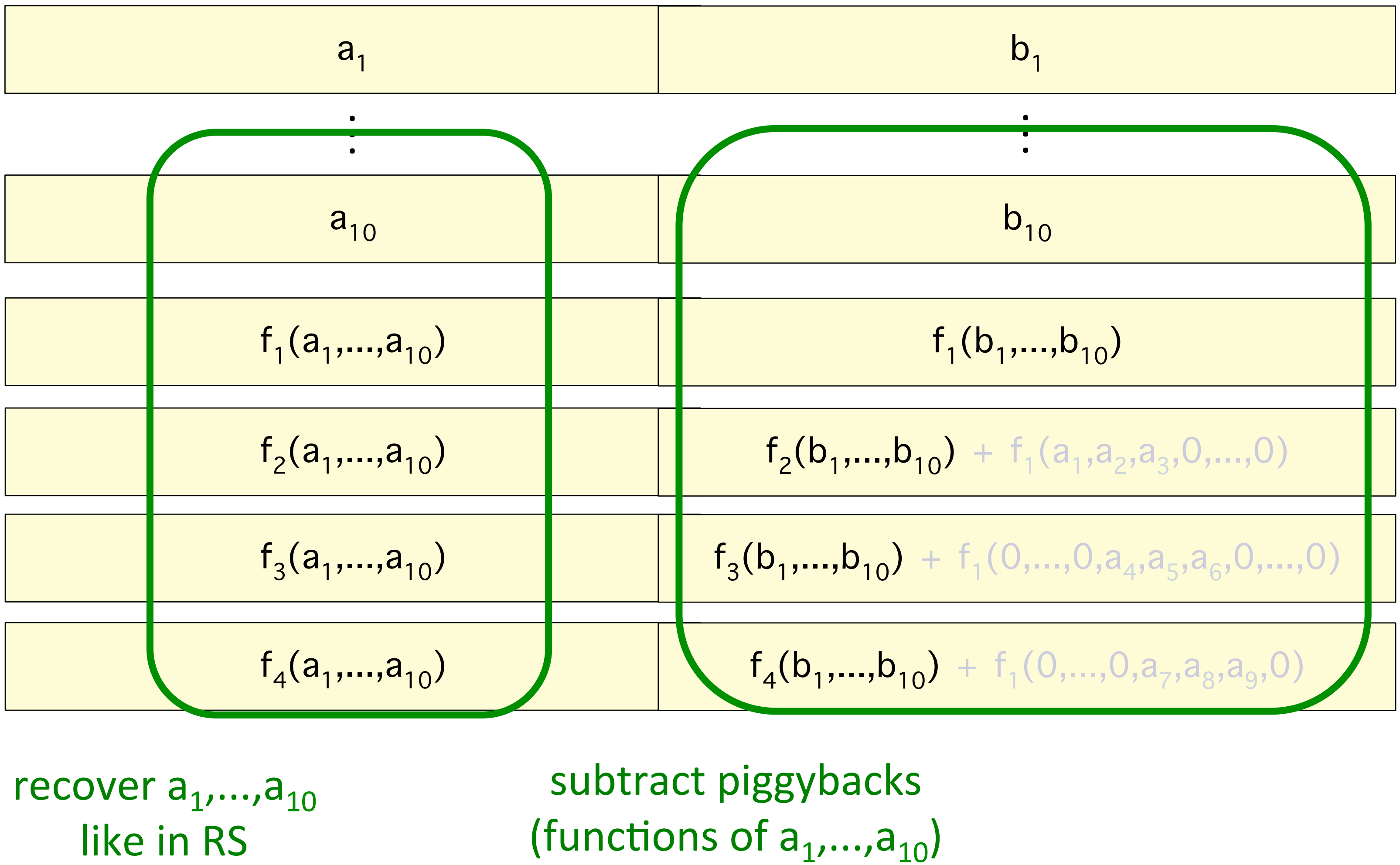
Tolerates any 4 block failures

a_1	b_1
⋮	⋮
a_{10}	b_{10}
$f_1(a_1,\ldots,a_{10})$	$f_1(b_1,\ldots,b_{10})$
$f_2(a_1,\ldots,a_{10})$	$f_2(b_1,\ldots,b_{10}) + f_4(a_1,a_2,a_3,0,\ldots,0)$
$f_3(a_1,\ldots,a_{10})$	$f_3(b_1,\ldots,b_{10}) + f_4(0,\ldots,0,a_4,a_5,a_6,0,\ldots,0)$
$f_4(a_1,\ldots,a_{10})$	$f_4(b_1,\ldots,b_{10}) + f_4(0,\ldots,0,a_7,a_8,a_9,0)$

recover $a_1,\ldots,a_{10}$
like in RS

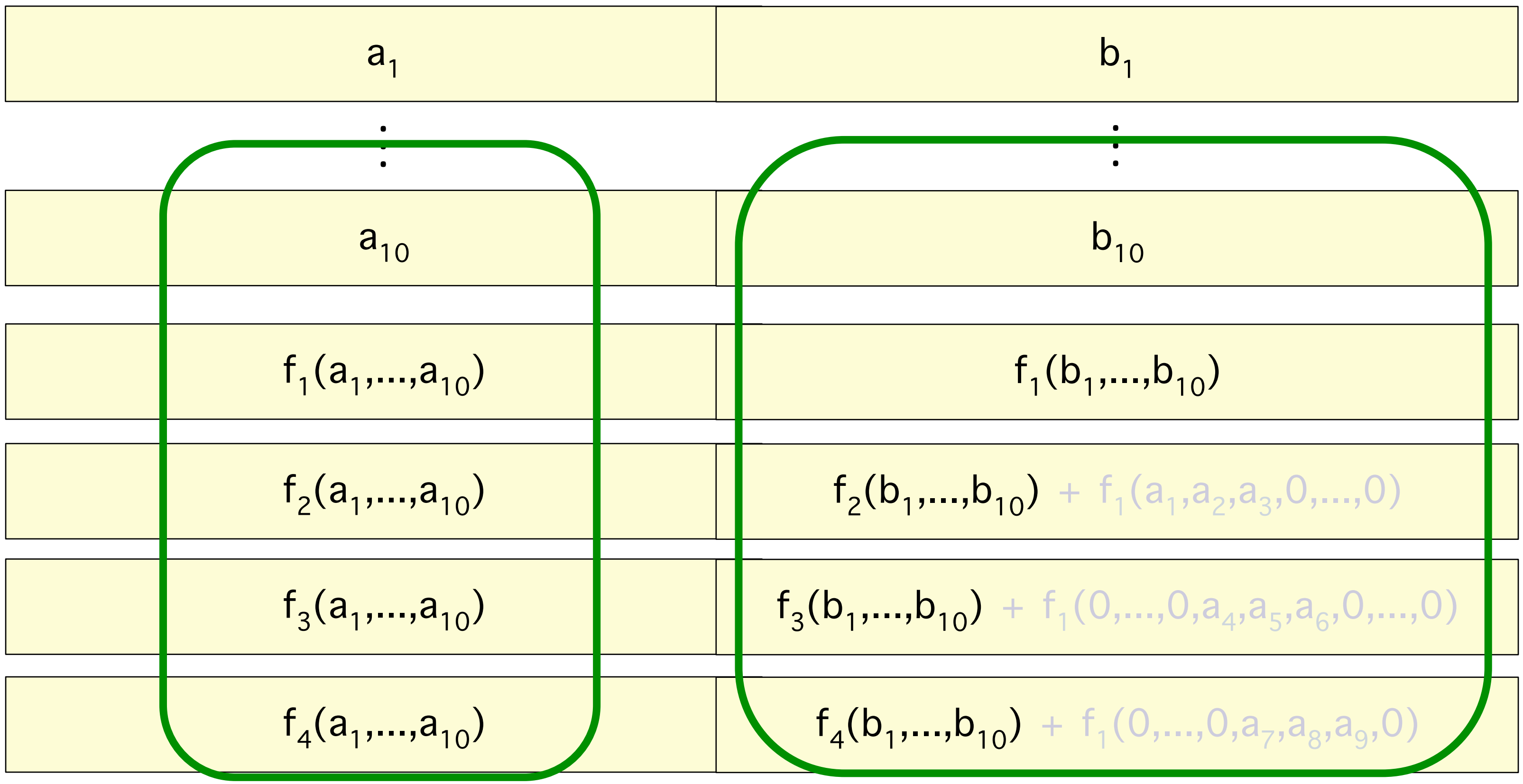
(14, 10) Piggybacked-RS code

Tolerates any 4 block failures



(14, 10) Piggybacked-RS code

Tolerates any 4 block failures



recover $a_1,\ldots,a_{10}$ like in RS

subtract piggybacks (functions of $a_1,\ldots,a_{10}$)

recover $b_1,\ldots,b_{10}$ like in RS

(14, 10) Piggybacked-RS code

Efficient data-recovery

a_1	b_1
a_2	b_2
a_3	b_3
⋮	⋮
a_{10}	b_{10}
$f_1(a_1,\ldots,a_{10})$	$f_1(b_1,\ldots,b_{10})$
$f_2(a_1,\ldots,a_{10})$	$f_2(b_1,\ldots,b_{10}) + f_4(a_1,a_2,a_3,0,\ldots,0)$
$f_3(a_1,\ldots,a_{10})$	$f_3(b_1,\ldots,b_{10}) + f_4(0,\ldots,0,a_4,a_5,a_6,0,\ldots,0)$
$f_4(a_1,\ldots,a_{10})$	$f_4(b_1,\ldots,b_{10}) + f_4(0,\ldots,0,a_7,a_8,a_9,0)$

(14, 10) Piggybacked-RS code

Efficient data-recovery

a_1	b_1
a_2	b_2
a_3	b_3
⋮	⋮
a_{10}	b_{10}
$f_1(a_1,\ldots,a_{10})$	$f_1(b_1,\ldots,b_{10})$
$f_2(a_1,\ldots,a_{10})$	$f_2(b_1,\ldots,b_{10}) + f_4(a_1,a_2,a_3,0,\ldots,0)$

(14, 10) Piggybacked-RS code

Efficient data-recovery

a_1	b_1
a_2	b_2
a_3	b_3
⋮	⋮
a_{10}	b_{10}
$f_1(a_1,\ldots,a_{10})$	$f_1(b_1,\ldots,b_{10})$
$f_2(a_1,\ldots,a_{10})$	$f_2(b_1,\ldots,b_{10})$ + $f_4(a_1,a_2,a_3,0,\ldots,0)$

recover $b_1,\ldots,b_{10}$
like in RS

(14, 10) Piggybacked-RS code

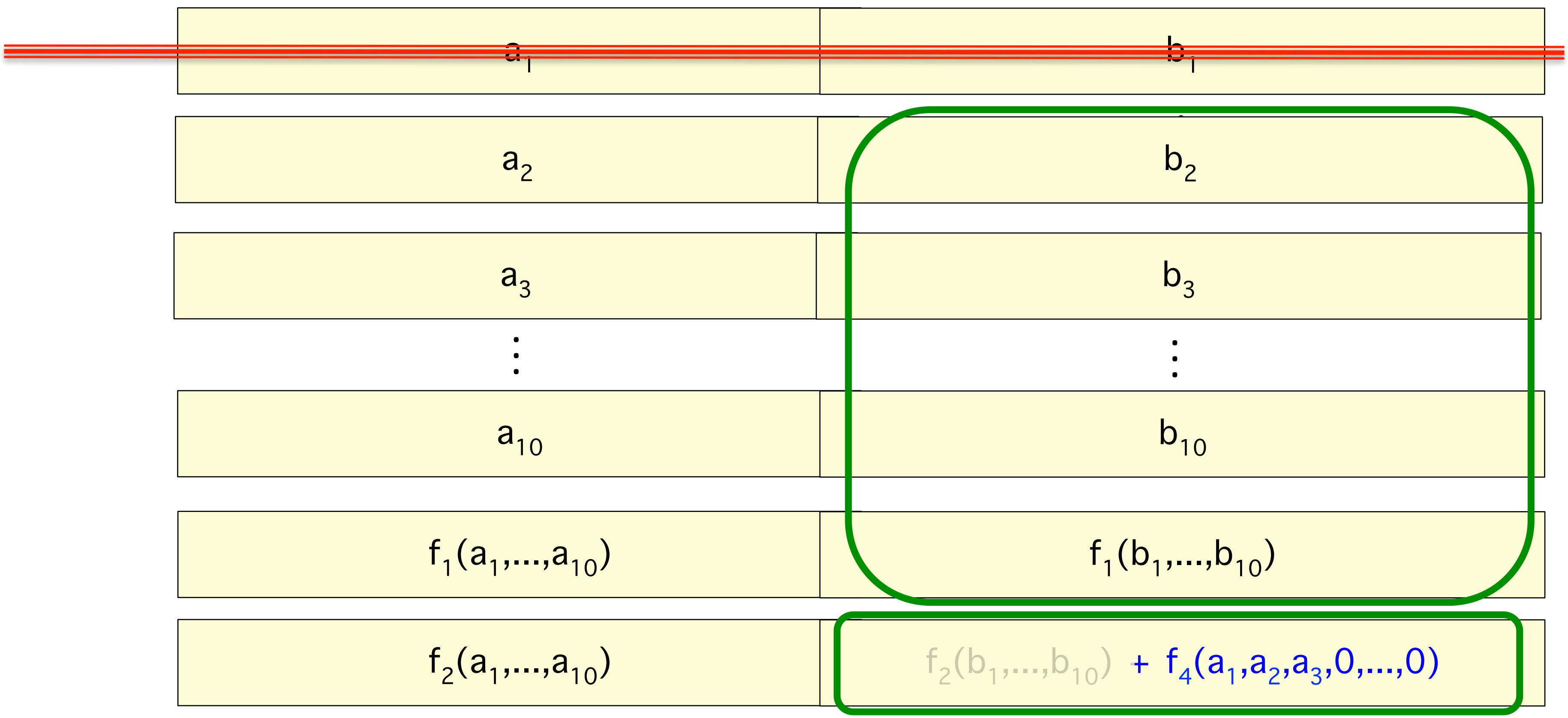
Efficient data-recovery

a_1	b_1
a_2	b_2
a_3	b_3
⋮	⋮
a_{10}	b_{10}
$f_1(a_1,\ldots,a_{10})$	$f_1(b_1,\ldots,b_{10})$
$f_2(a_1,\ldots,a_{10})$	$f_2(b_1,\ldots,b_{10}) + f_4(a_1,a_2,a_3,0,\ldots,0)$

recover $b_1,\ldots,b_{10}$
like in RS

(14, 10) Piggybacked-RS code

Efficient data-recovery

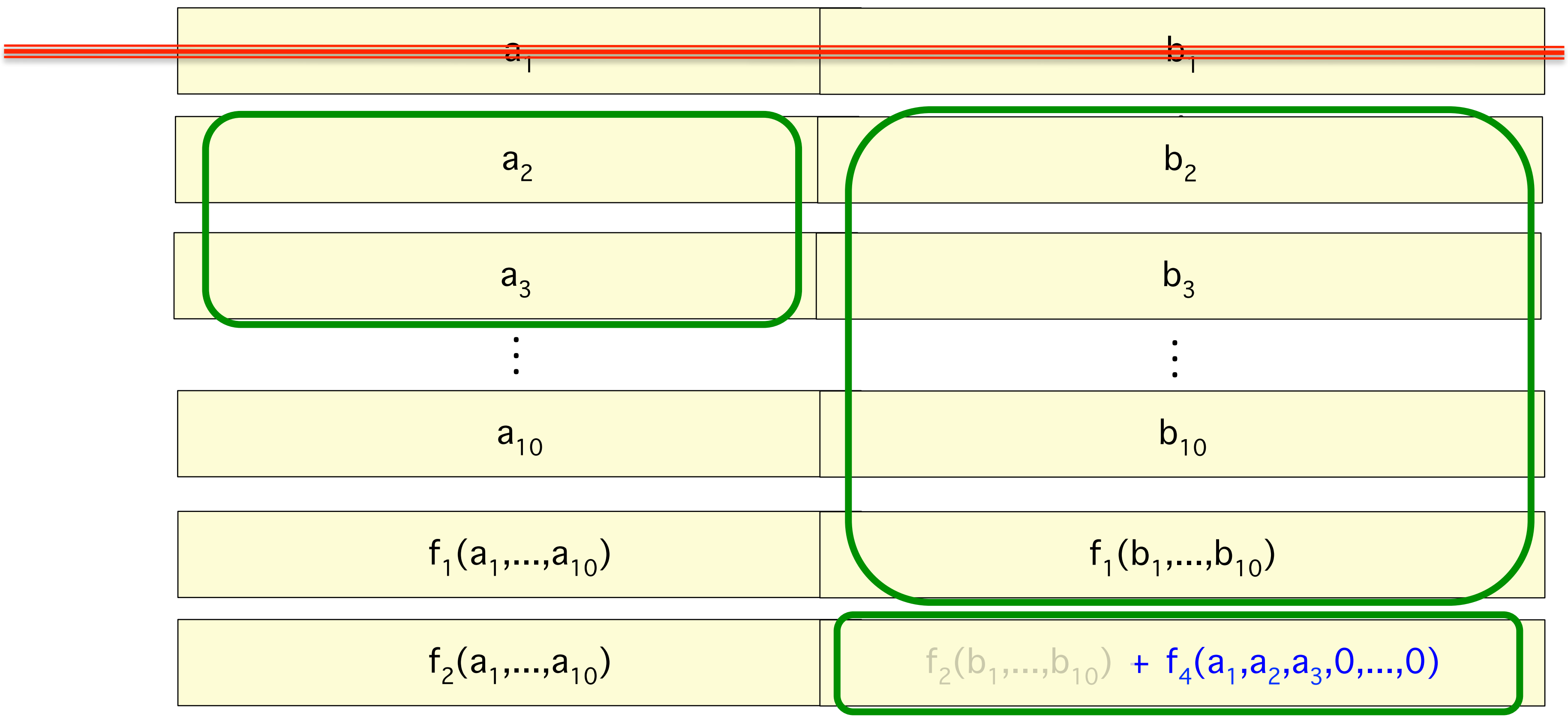


recover $b_1,\ldots,b_{10}$ like in RS

subtract $f_2(b_1,\ldots,b_{10})$

(14, 10) Piggybacked-RS code

Efficient data-recovery

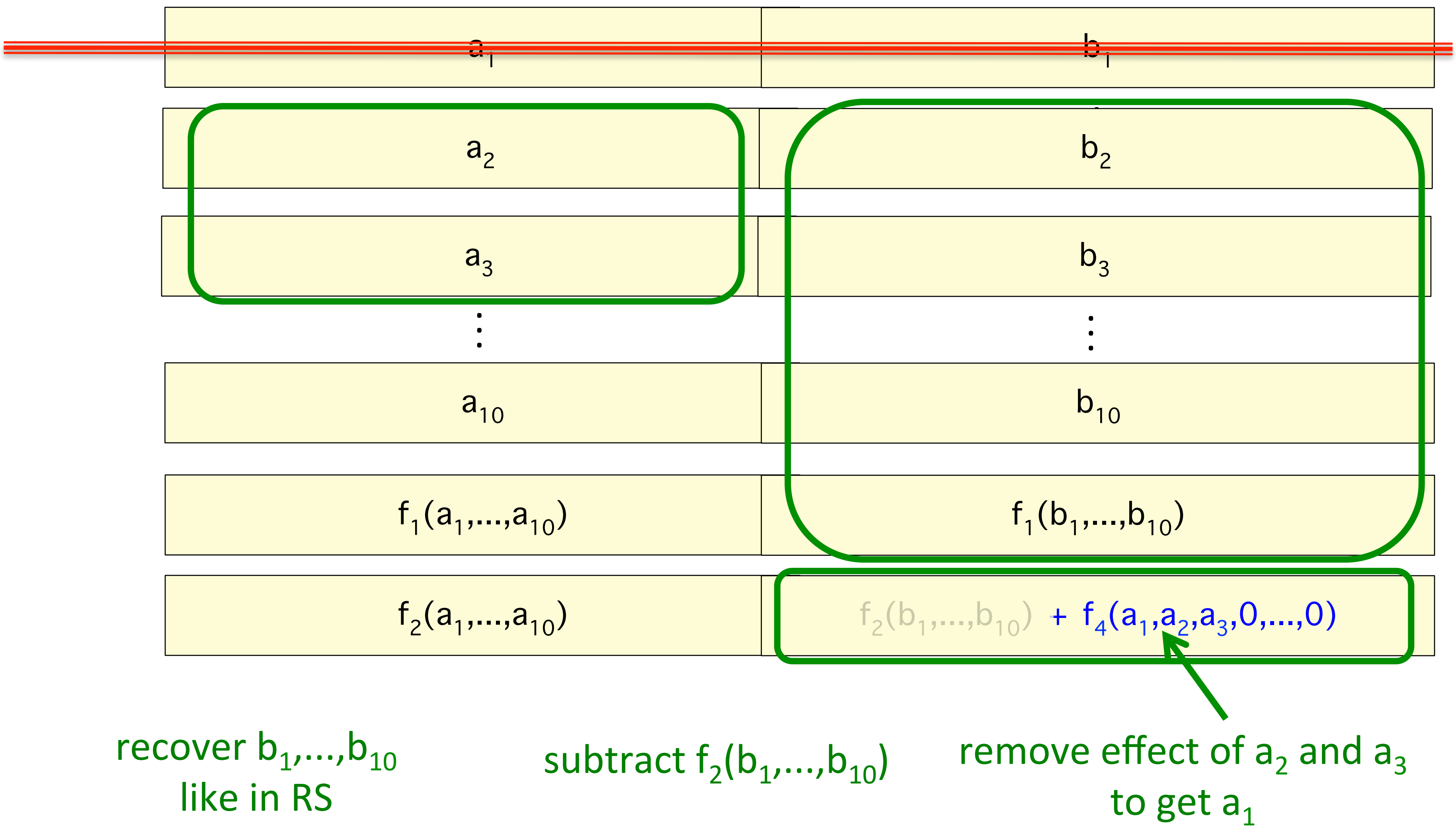


recover $b_1,\dots,b_{10}$ like in RS

subtract $f_2(b_1,\dots,b_{10})$

(14, 10) Piggybacked-RS code

Efficient data-recovery



- (14, 10) Piggyback-RS in HDFS
  - 30% reduction in IO and download on average
  - same storage & fault tolerance

- However, requires connectivity > in RS
  - is this a concern ?

Is connecting to more nodes a concern ?

We performed measurements for various data-sizes in the Facebook Warehouse cluster in production.

Piggyback-RS codes:

- Reduce primary metrics of IO & download
- Time to repair also reduces upon connecting to more

Locality/Connectivity NOT an issue in this setting

Outline

- Introduction & Motivation
  – Measurements from Facebook's Warehouse cluster
- The Piggybacking framework
- Via the Piggybacking framework
  – Best known codes for several settings
  – Comparison with other codes
  – Preliminary practical experiments
- Summary & future work

Summary

- “Piggybacking” code design framework
- 3 piggyback function designs

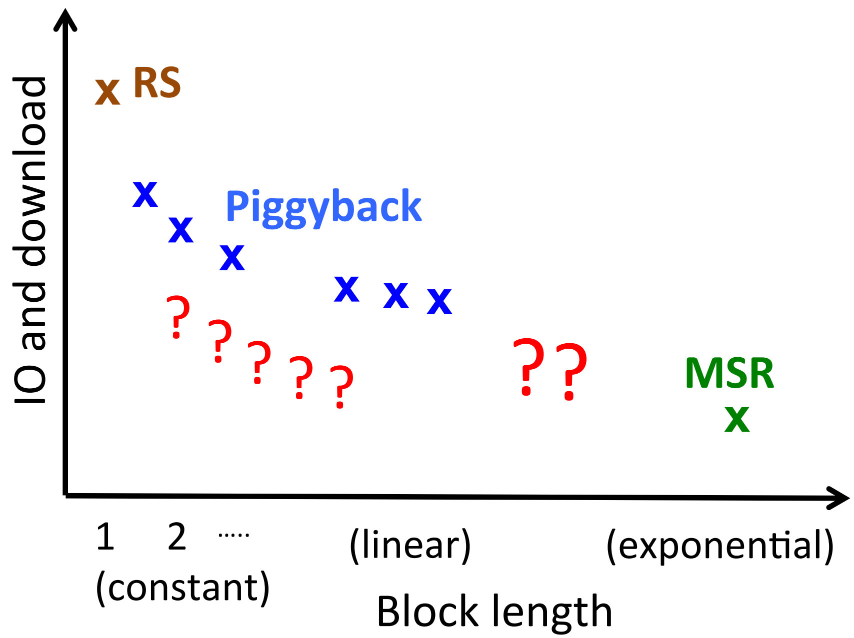
- Best known codes for several settings
  - MDS + high-rate + small block length
  - binary MDS (vector)
  - parity repair in regenerating codes

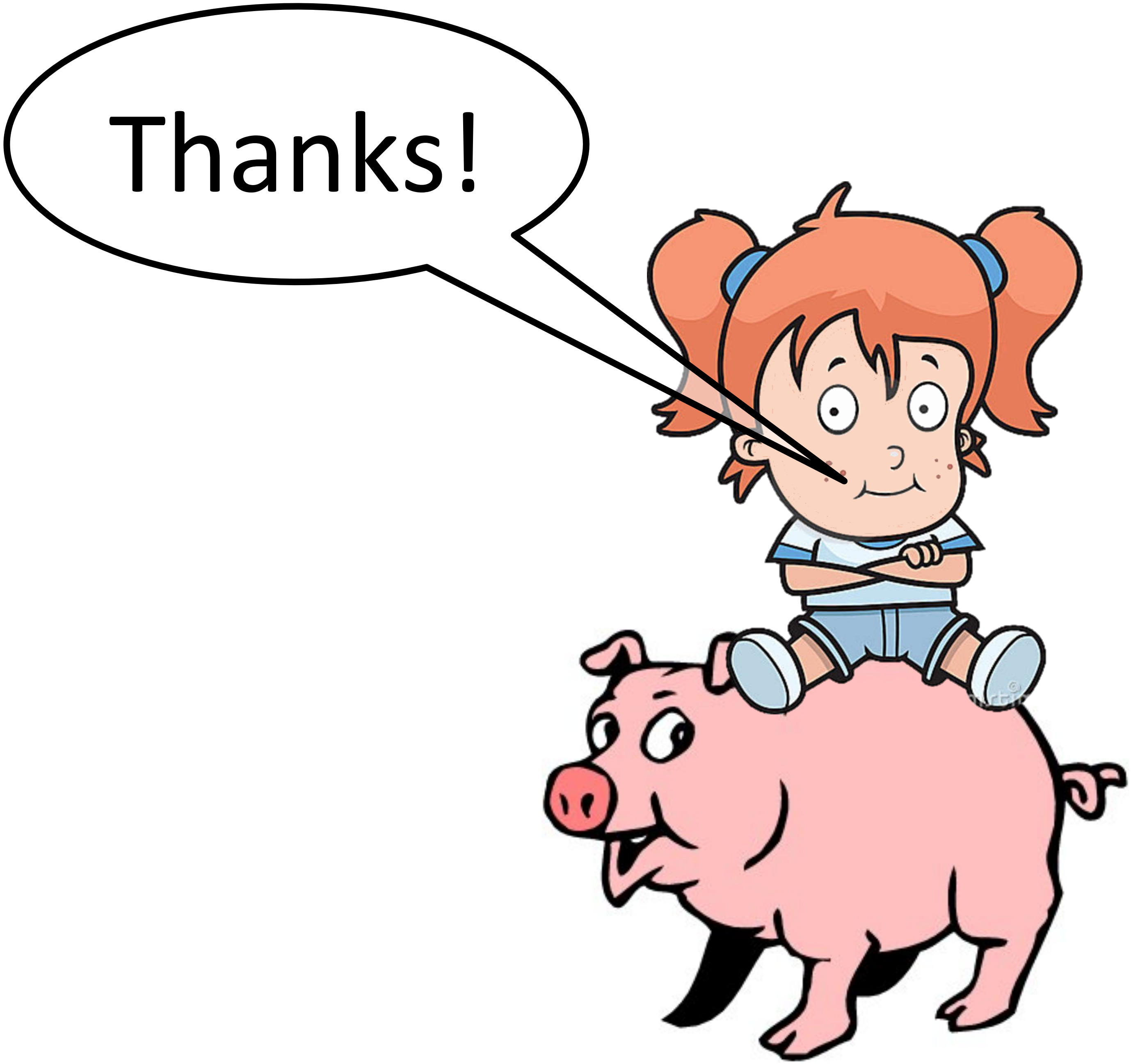
Future work & open problems

- Other Piggybacking designs / applications
- Bounds for Piggybacking approach ?

Future work & open problems

- Other Piggybacking designs / applications
- Bounds for Piggybacking approach ?
- High-rate MDS: Tradeoff between block length & IO/download




Thanks!