Яндекс

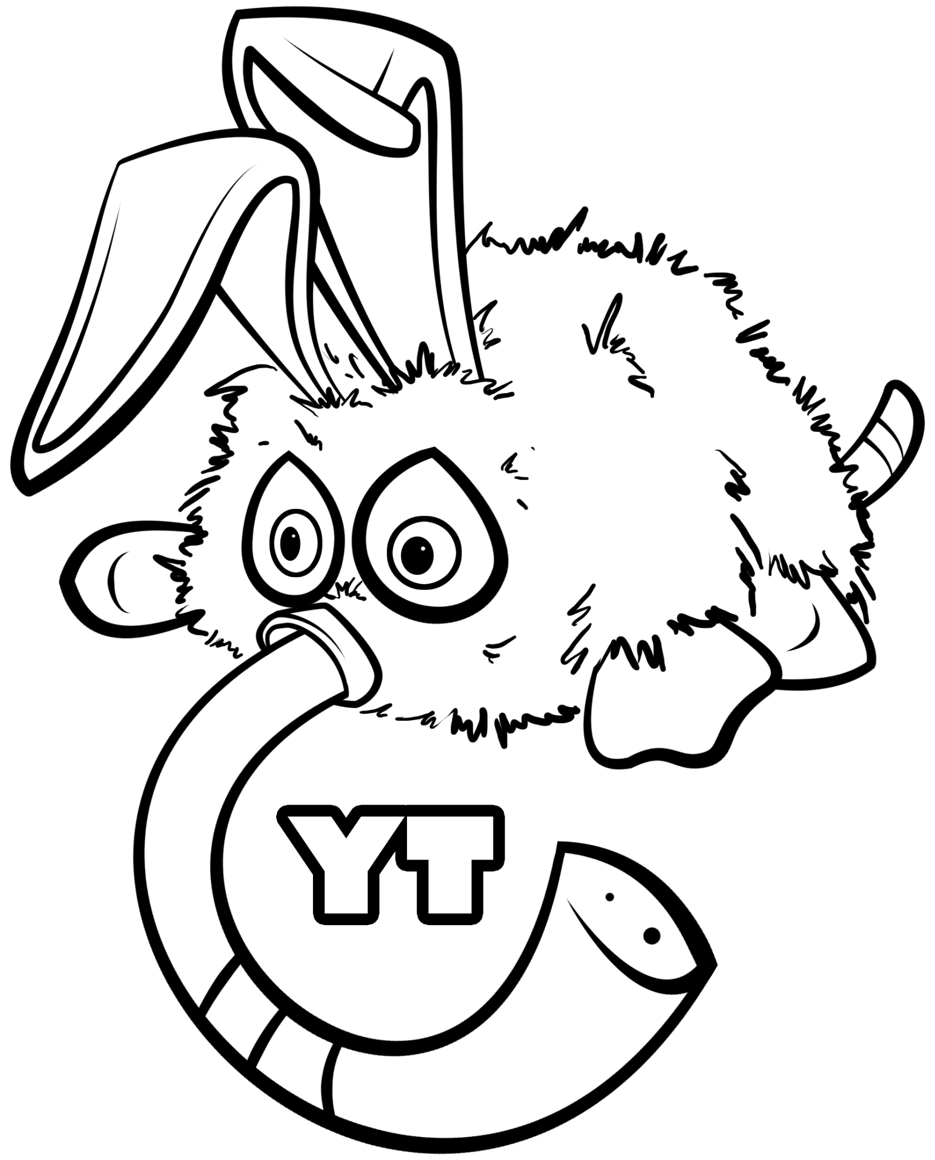
Erasure Coding at Scale

Maxim Babenko

YT Storage Subsystem

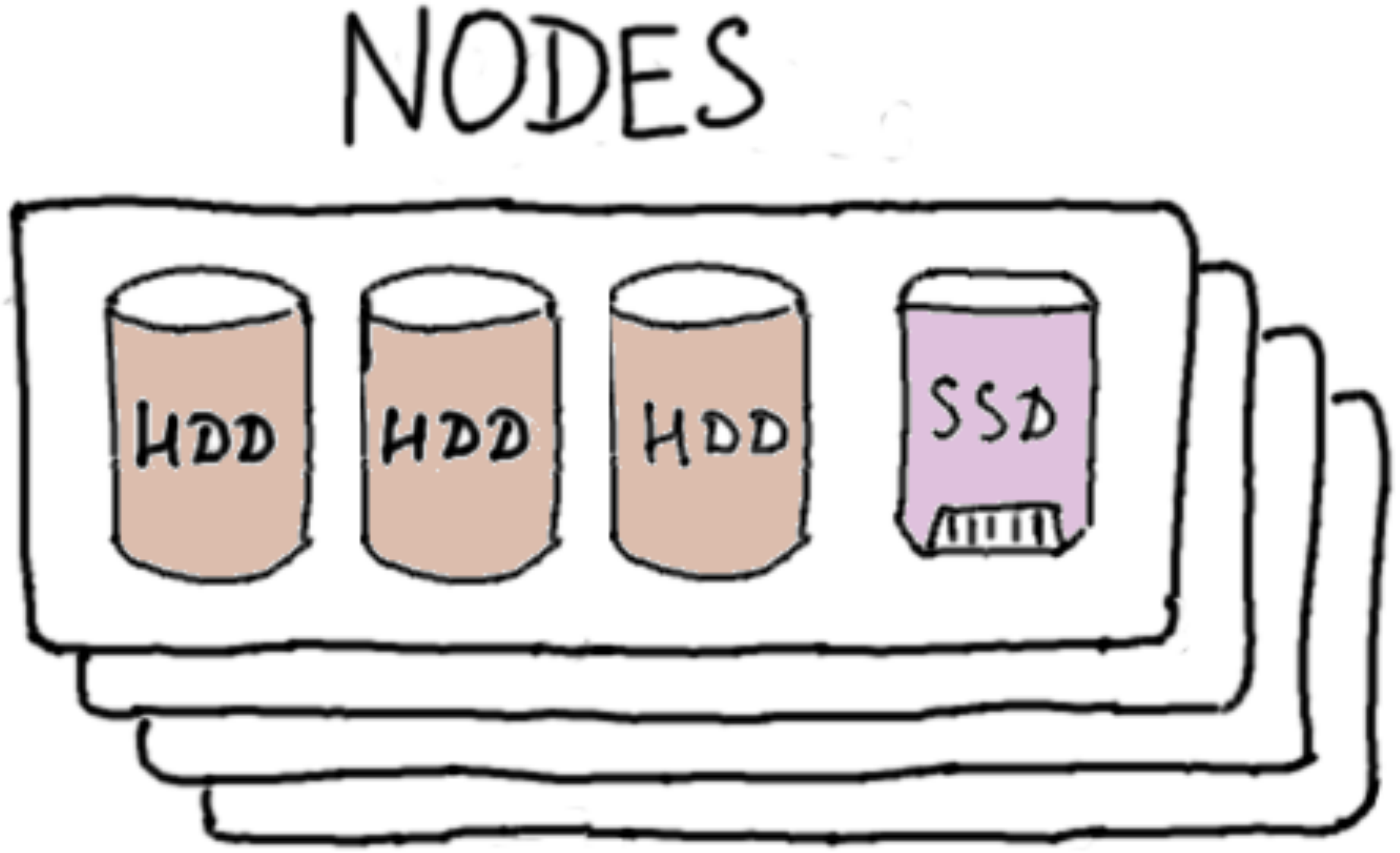
What is YT?

- The primary storage and compute platform at Yandex
- Stores data of various kinds (logs, ML models, ads info, crawler indexes etc)
- Runs on (mostly) commodity hardware in our own datacenters
- Provides MapReduce-like APIs (with vast extensions) for long-running batch operations
- Provides low-latency key-value storage with multi-row commits and snapshot isolation semantics
- Multi-tentant: supports running both ad-hoc and production workloads within the same cluster



Storage Overview

- Data is stored at **nodes** (~10-100K)
- Each node has ~10 **disks**, 1-10T each
- Various physical disk **types**: HDD, SDD, NVMe
- Total **capacity** ~1EB



Storage Overview

Most disk capacity is occupied by tables

Strongly-typed schema

Collection of rows

Row order is important

Sorted tables enable fast reduce (join) operations

Tables are split into (blob) chunks

Logical unit (portion) of data

Blob chunks are immutable

1G is a good size for a chunk

Metadata and Control Plane

Namespace tree (Cypress)

Not covered by this talk

Chunk metadata

~1B of chunks in a large cluster

Sequence of chunks for each table

Replicas of each chunk

Metadata and Control Plane

Metadata

TBs of metadata for large clusters

Purely in-memory data structures

Metadata sharding: master cells

Masters

Separate group of machines handling metadata

RAFT-like consensus protocol for fault tolerance

~10-20 of RAFT quorum groups (**cells**) for large clusters

Typically 5-7 masters in each group

Metadata and Control Plane

Chunk replica orchestration

Decide which nodes should receive replicas of new chunks

Handle replica loss due for failures

Schedule and track chunk jobs: replication, removal, repair, seal

Orchestration involves some transient data structures

Handled at RAFT group leaders

Reconstructed upon re-election

Replicated Blob Chunks

Why replicate?

10-100 of disk failures per day on large clusters

Spontaneous node outages due to whole node or even rack failures

Rolling restarts

RF=3 is the golden standard for data replication

In other words: x3 disk space overhead

Can tolerate up to 2 simultaneous disk failures

Less overhead?

RF=2 is viable but not very reliable, data loss may occur

RF=1 is totally unreliable (but could be used for temporary data)

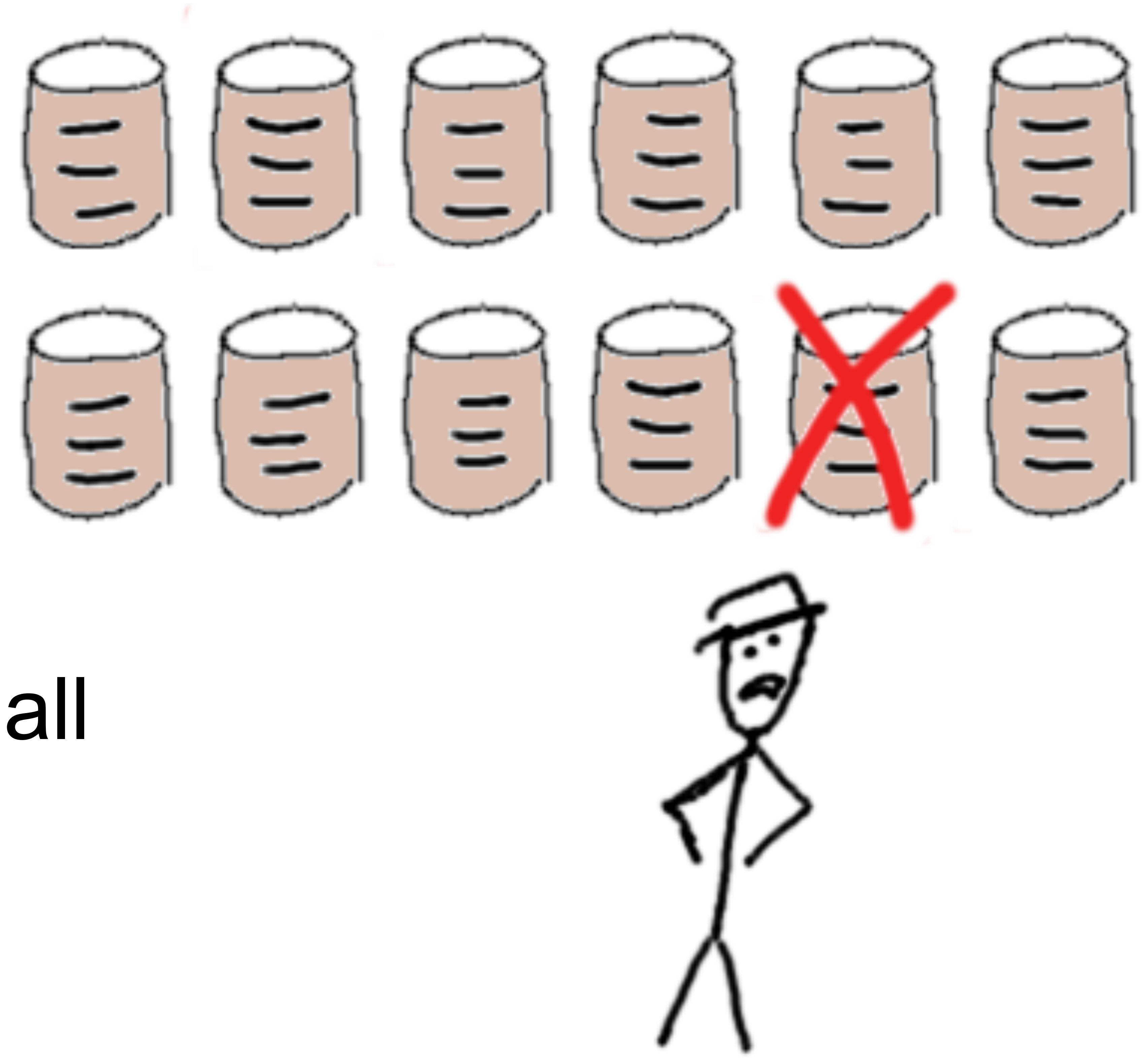
How Strong Are the Guarantees?

- **Is is enough to tolerate 2 disk failures?**

We still have 10-100 of failures per day!

What is “simultaneous failure” exactly?

Do disks actually die “simultaneously”?



- **Timing is important**

Tens of minutes to ensure proper replication of all affected chunks after a single node loss

Chunk Placement Freedom

None

Node-to-node replication

Full

Each chunk replica can be placed anywhere

Something in the middle

Divide disks into logical partitions

Replicate each partition to a number of groups

Somehow assign partition groups to chunks

Free Placement Pros and Cons

Pros

Really fast recovery: the whole cluster is participating

Highly elastic storage: can easily add/remove nodes

Cons

Need for sophisticated metadata storage

Weaker out-of-the-box data locality

Higher network utilization

Other Types of Failures

Single disk failure

No hot-plug as of now, takes the whole node down for maintenance

Node hardware failure (CPU, memory, PSU, NIC etc)

Same as above

Rack failure

Could be a ToR switch firmware update

Whole DC is down

Not very important for single-DC clusters

Mostly same as rack failure for multi-DC clusters

Placement Anti-Affinity

Failure domains

Subsets of nodes that tend to die simultaneously

Failure domain shares some physical SPoF

Examples

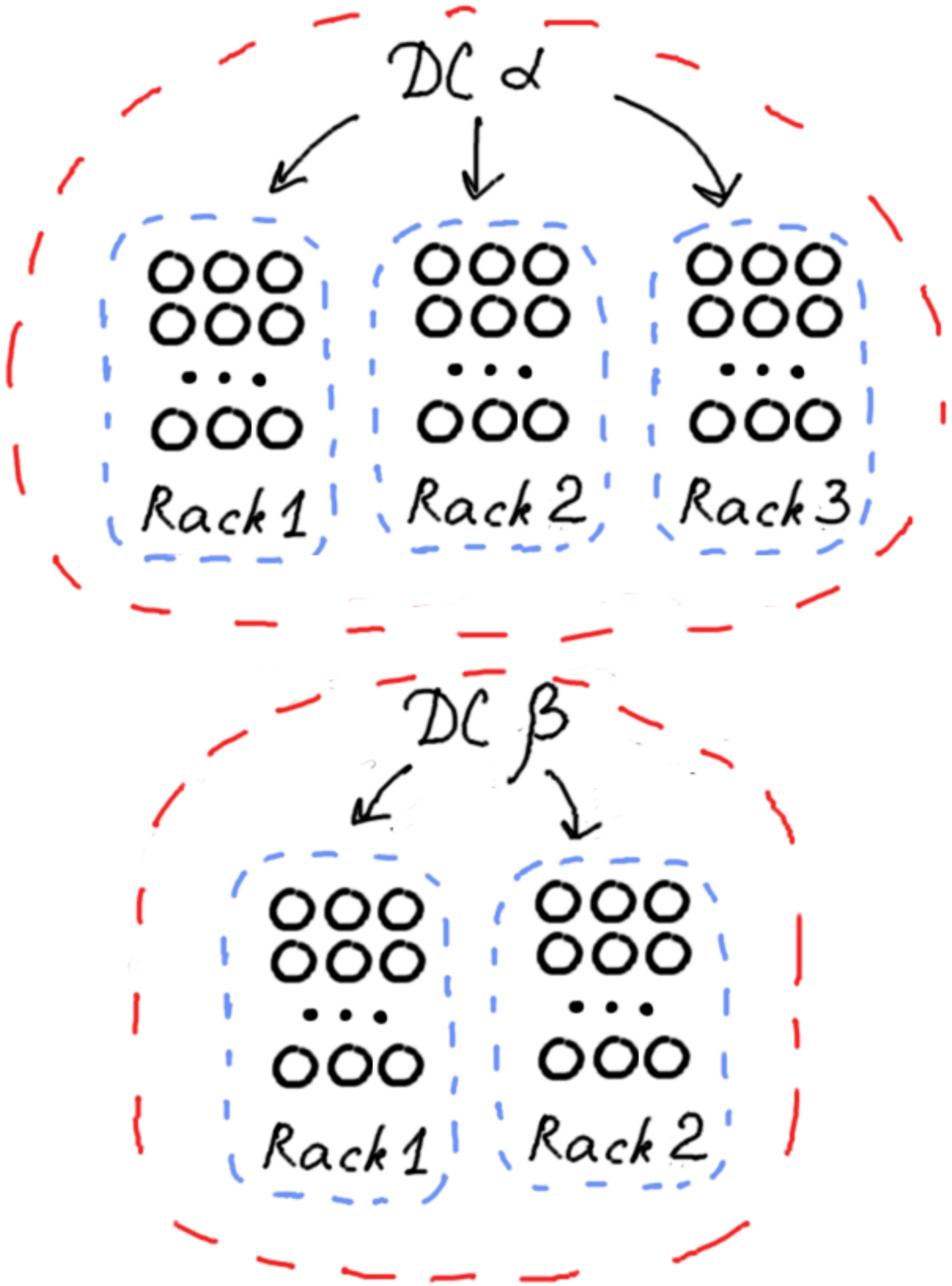
Node

Rack

Datacenter

Anti-affinity constraints

Don’t place too many replicas in the same failure domain!



Single-DC Placement Anti-Affinity

Failure domains

Nodes, racks

Don’t place more than one replica in each failure domain

Sample scenarios

Rack goes down: just one replica is lost

Rack plus an arbitrary node go down: still have a live replica

Rack plus two nodes: some data could become unavailable

Blob Chunk Structure

Metadata

Protobuf with extensible structure

Index for block offsets and lengths in payload

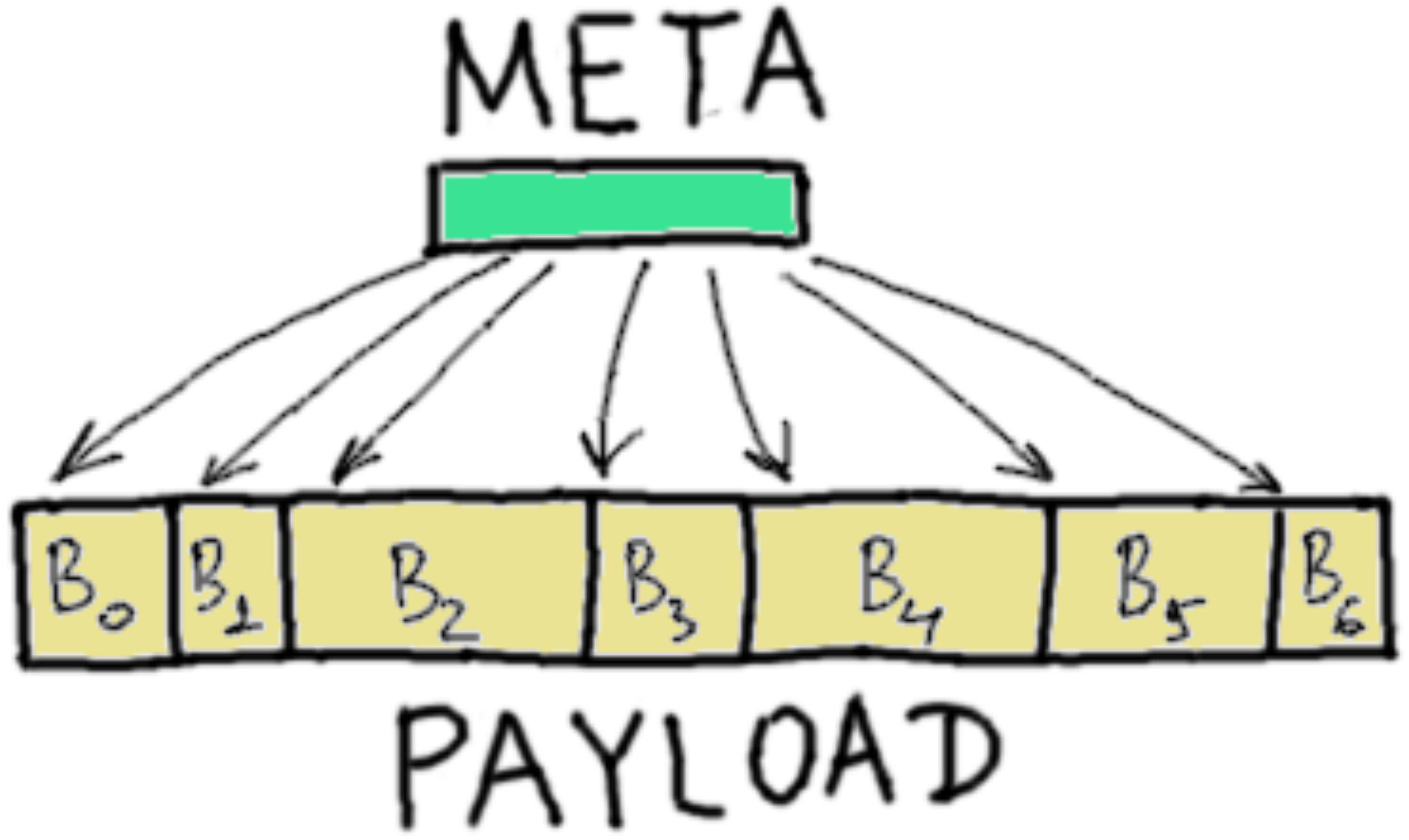
Payload

A sequence of blocks (opaque blobs)

Blocks are compressed/uncompressed as a whole

Block is a unit of read

We don’t want metadata proliferation => we like big chunks (>1GB)



Write Pipeline

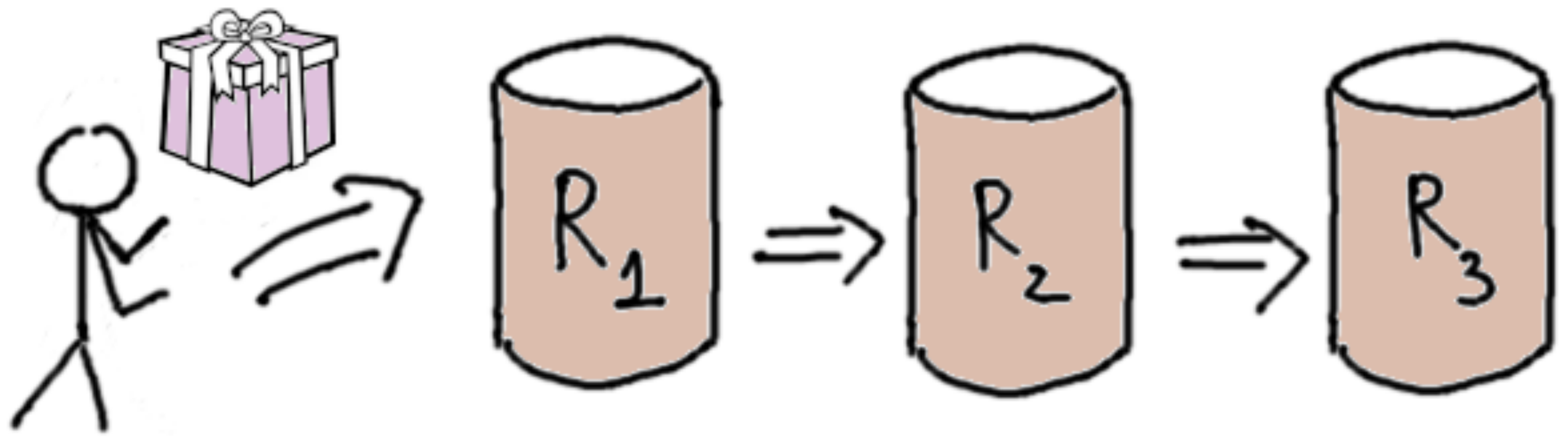
Client is producing data block-by-block

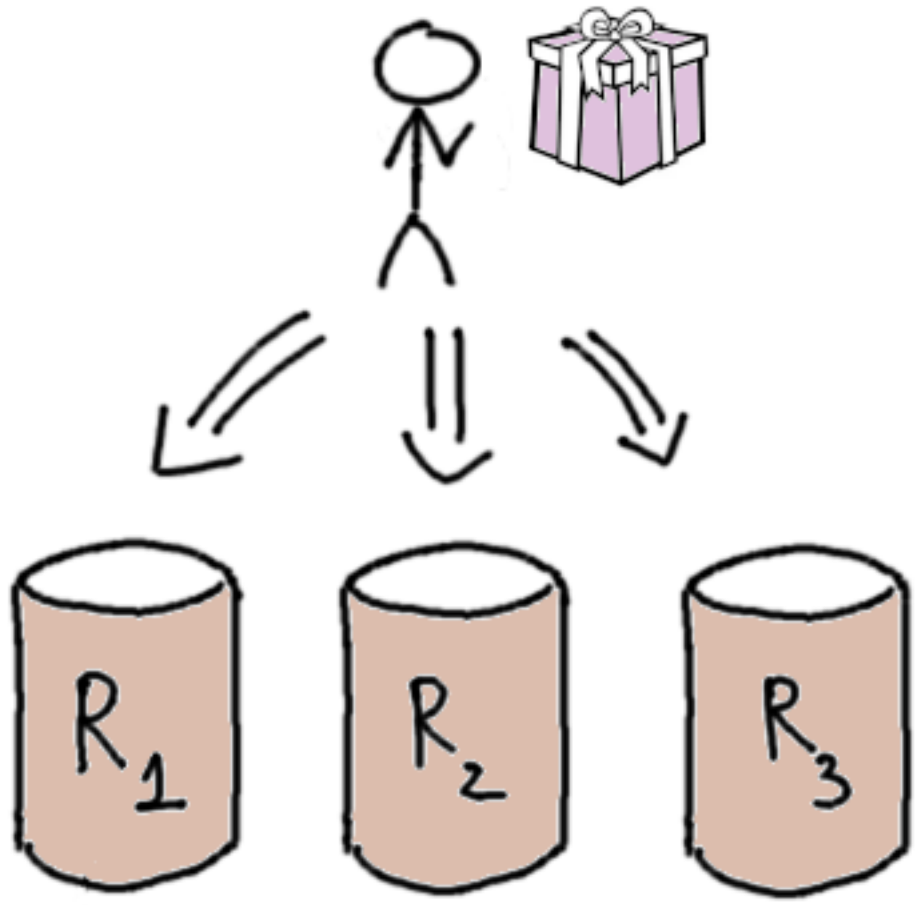
Do not store the whole chunk in memory (can take GBs)

Forward blocks to pre-allocated replica nodes

Pipeline shape

a **chain** or a **star**





Theory vs Practice

Theory

Need RF=3 replicas

All flushed to a persistent medium

Practice

Let’s write URF=2 replicas and extend these to RF=3 at background

Let’s not interrupt the pipeline even if just MURF=1 alive replica remains

Let’s not invoke *fdatasync* and hope the best

One should choose which of the above to apply with care!

Read Pipeline

Primary goal: read fast

Probe replicas before reading, choose the least-loaded one

One the best replica is selected, issue a read request

Replicas may vanish, re-appear and move

Refresh replica placement info from time to time

Read hedging (ask multiple replicas in short succession)

Erasure Blob Chunks

Erasure Coding

Contract

Given: N data parts (blobs, typically of the same length L)

Compute: K parity parts (blobs of length L)

Such that: given any subset of G (<N+K) parts it is possible to reconstruct all the parts

Simple replication

N=1, K=RF-1, G=1

Erasure-coded Chunks

Place all N+K parts at **distinct** nodes (or even **failure domains**)

Fault tolerance: can tolerate up to N+K-G node failures

The Math Beneath

Words

Parts are sequences of **words**

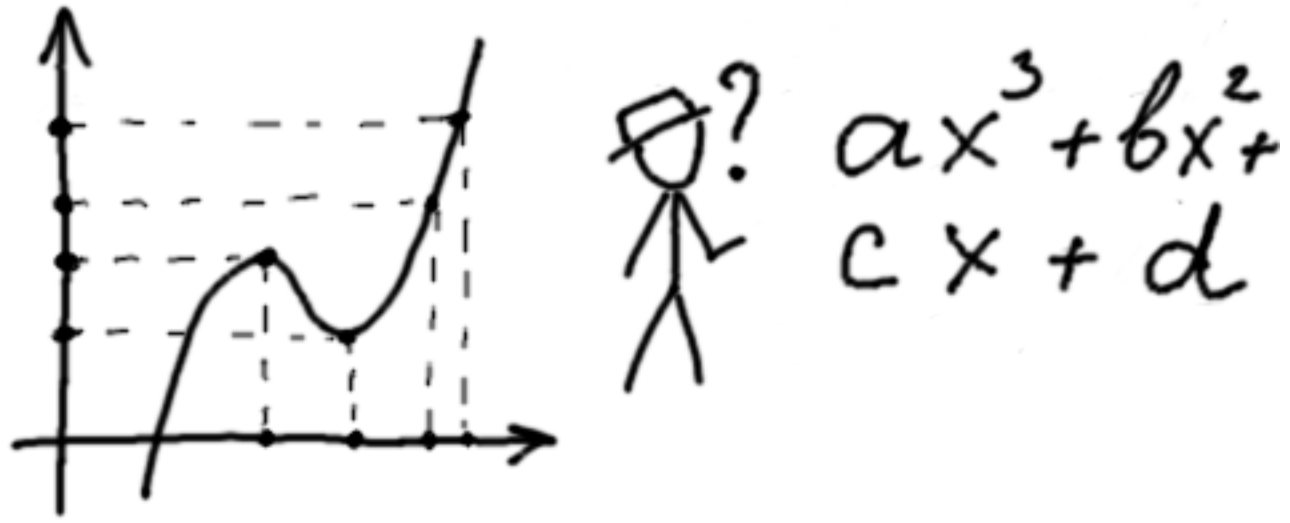
Word is an element of a finite field

Erasure coding

For each position within **data** parts compute from N data words some K **parity** words such that **any G** words of the above are sufficient to reconstruct the whole set of data

Erasure decoding

Resembles reconstructing a polynomial from given values at given points



Erasure Coding in Practice

Reed-Solomon codes are quite popular

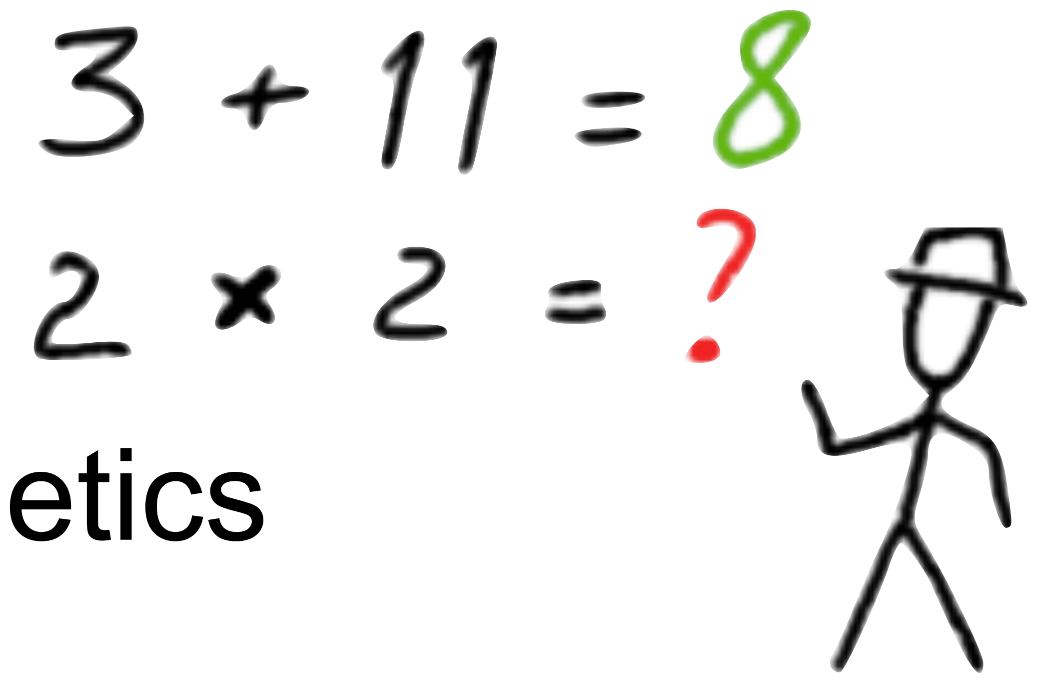
N and K could be arbitrary

Typically words are **bytes**, GF(2^8)

Addition/subtraction is just XOR and is very fast

Multiplication is hard and reduces to polynomial arithmetics

Division is no better



Popular implementations

Old-and-mature Jerasure library

Pretty modern ISA-L from Intel (requires modern processors with proper SIMD instructions)

Constructing The Parts From Chunk Data

Chunk splitting

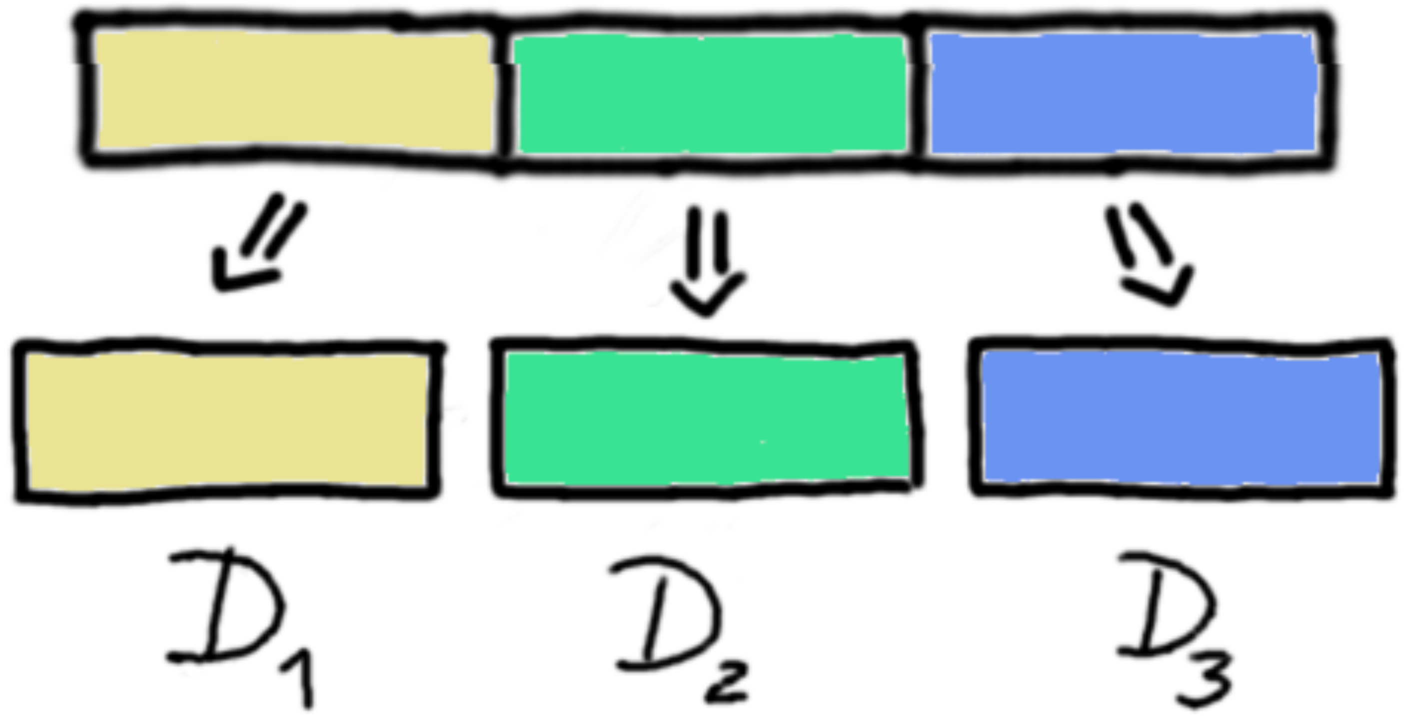
Divide each chunk into N data parts of same length

Compute K additional data parts

Pretty much preserves the block structure

Need to have the whole chunk in memory prior to encoding

Preferred for batch workloads



Constructing The Parts From Chunk Data

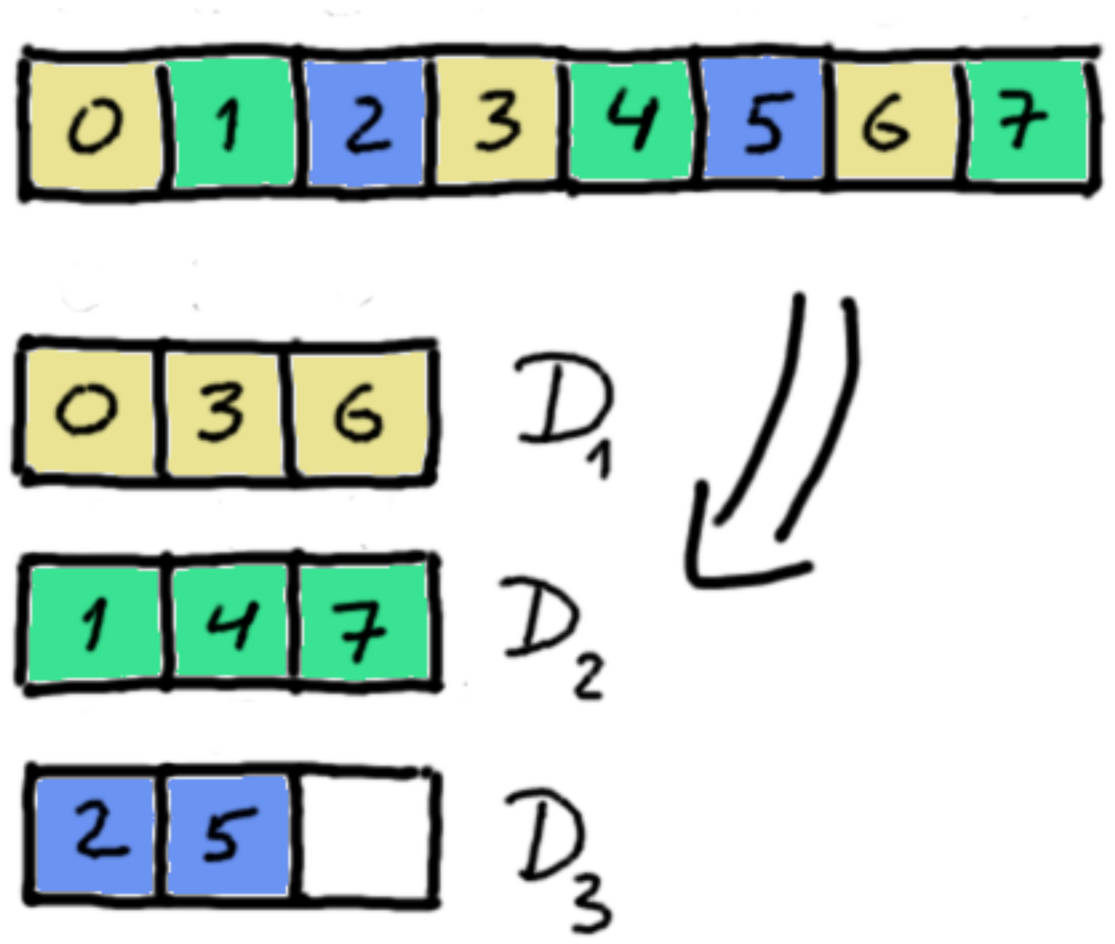
Chunk striping

i^{th} byte goes to $(i \bmod N)^{th}$ part

Can encode data on-the-fly

Reading a range of chunk involves contacting many nodes

Not useful for batch workloads but essential for journals



Erasure Repair: Basics

Node goes down

Masters detect that some chunk parts are missing

Repair jobs are spawned at nodes to run the decode pipelines

Recovering from a node loss is fast since all cluster nodes participate

Readers may…

Wait until data is fully recovered

Run on-the-fly repair

Erasure Repair: More Tricks

Repair order

Masters maintain queues of pending repairs

When client is missing some data parts it contacts masters

These requests promote (lift-to-front) chunks that are being actively read thus prioritizing repairs

Safeguards

Repair traffic and concurrency are throttled not to overload the system

Nice side effect

Node decommission is better handled via repair rather then replication

Erasure Coding Examples

Reed-Solomon with N=6, K=3

Needs 9 nodes (or even racks) to store a chunk

Pros

Incurs x1.5 disk space overhead

Tolerates up to 3 disk failures (better than RF=3)

Cons

Repairing even a single data part requires reading 6 other parts

Decoding always involves GF-multiplication and is CPU-intensive

Erasure Coding Examples

LRC(12,2,2)

LRC stands for **Local Reconstruction Codes**

N=12, K=4, RF=16

Incurs x4/3 disk space overhead

Not **Maximum Distance Separable**: cannot tolerate arbitrary 4 failures

But can tolerate up to 3 arbitrary failures

Single data part failure

Needs just 3 another parts for decoding

Decoding involves just additions (XORs)

Much of Yandex cold data is now stored using this scheme

Текст

Detour: Data Journalling

Chunk Mutability

Table chunks are immutable

Must have all data in memory before writing out to disks

Write pipeline is optimized for throughput rather than latency

What about OLTP?

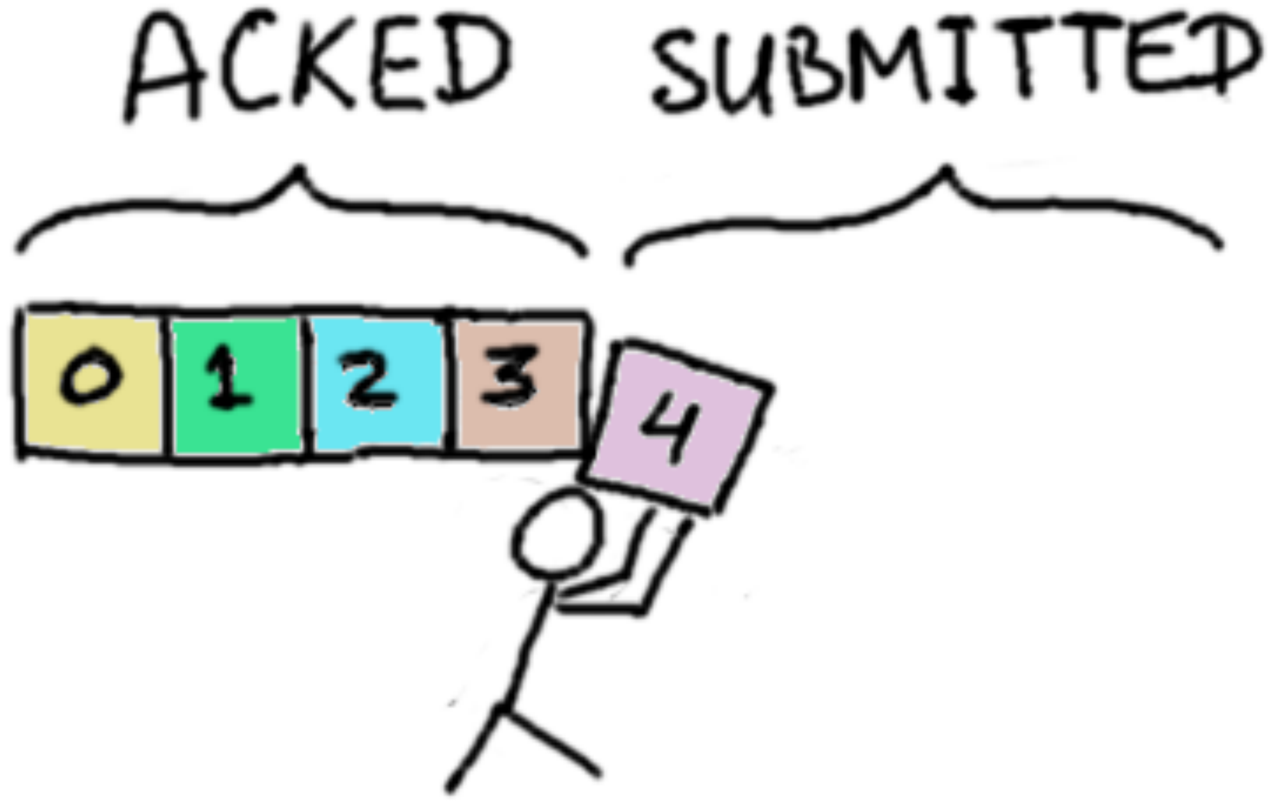
KV storages need place to store changesets (WALs)

KV storages struggle for low latency

Can pack changesets into chunks but cannot wait until chunk is full before flushing it to disks

Journal Chunks: Contract

- **Journal** chunks (AKA **append-only** chunks)
- Contains a sequence of (typically small) **records**
- When record is appended, is receives a **record number** (LSN)
- Records are not reordered (w.r.t. LSN)
- Once a record is **acked** by write pipeline it and all its predecessors are reliably stored
- System never loses any **acked records**
- During a crash, system may discard some records that were submitted (but were **not acked** yet)
- Ack **latencies** are low (tens of ms)



Journal Chunks: Implementation

Basic notions

RF = total number of replicas to store

WQ = write quorum, number of replicas to wait before acking the records

RQ = read quorum, number of alive replicas needed to determine the number of acked records

Safety

RQ + WQ > RF (or each read quorum intersects each write quorum)

Example

RQ=3, WQ=2, RF=4

Journal Chunks: Write Pipeline

Write pipeline

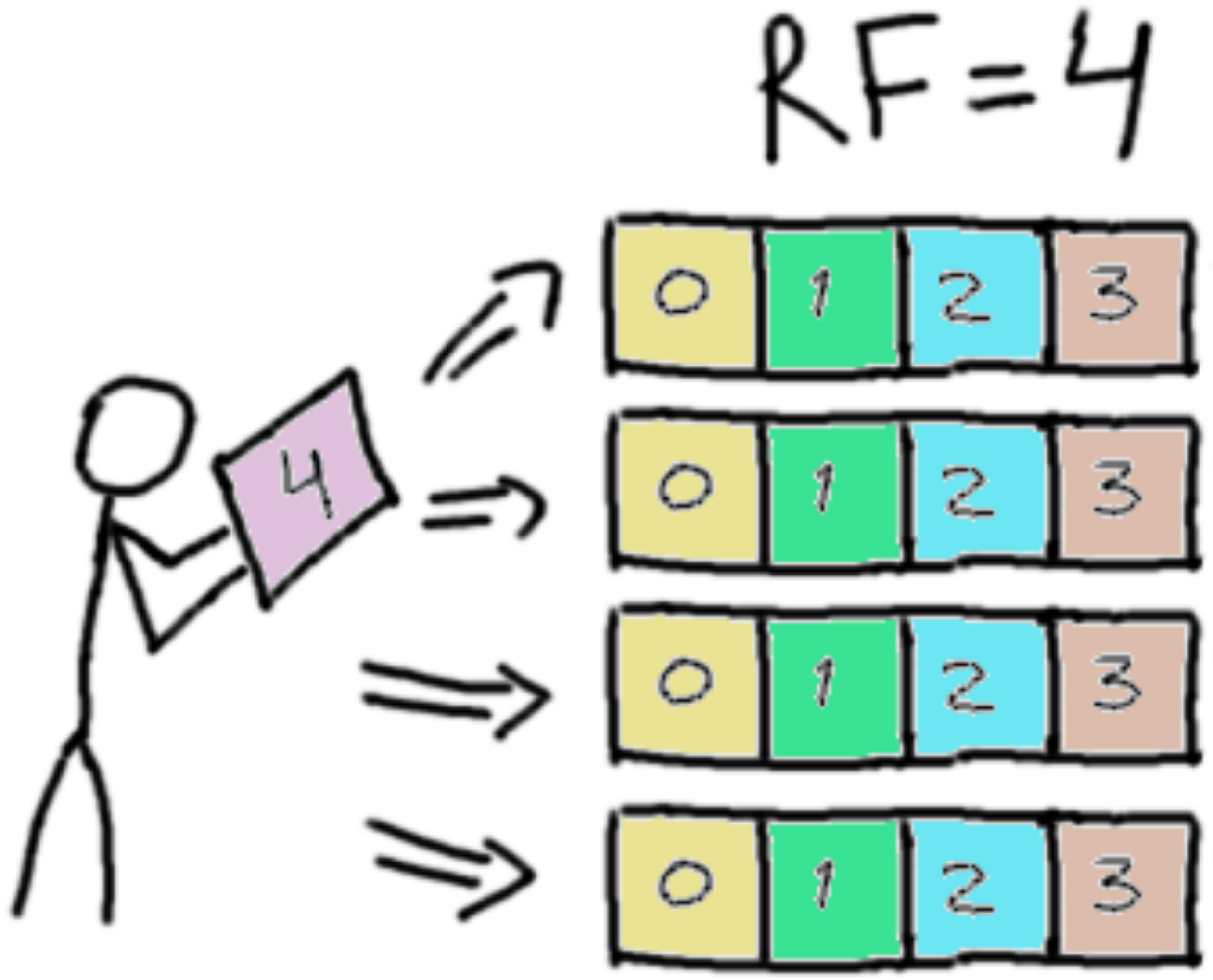
Send new records to all replicas
Ack when WQ replicas are flushed

Consistency

Replicas are prefixes of each other

Handling replica failures

If less than WQ replicas were successfully written then switch to another chunk



Journal Chunks: Read Pipeline

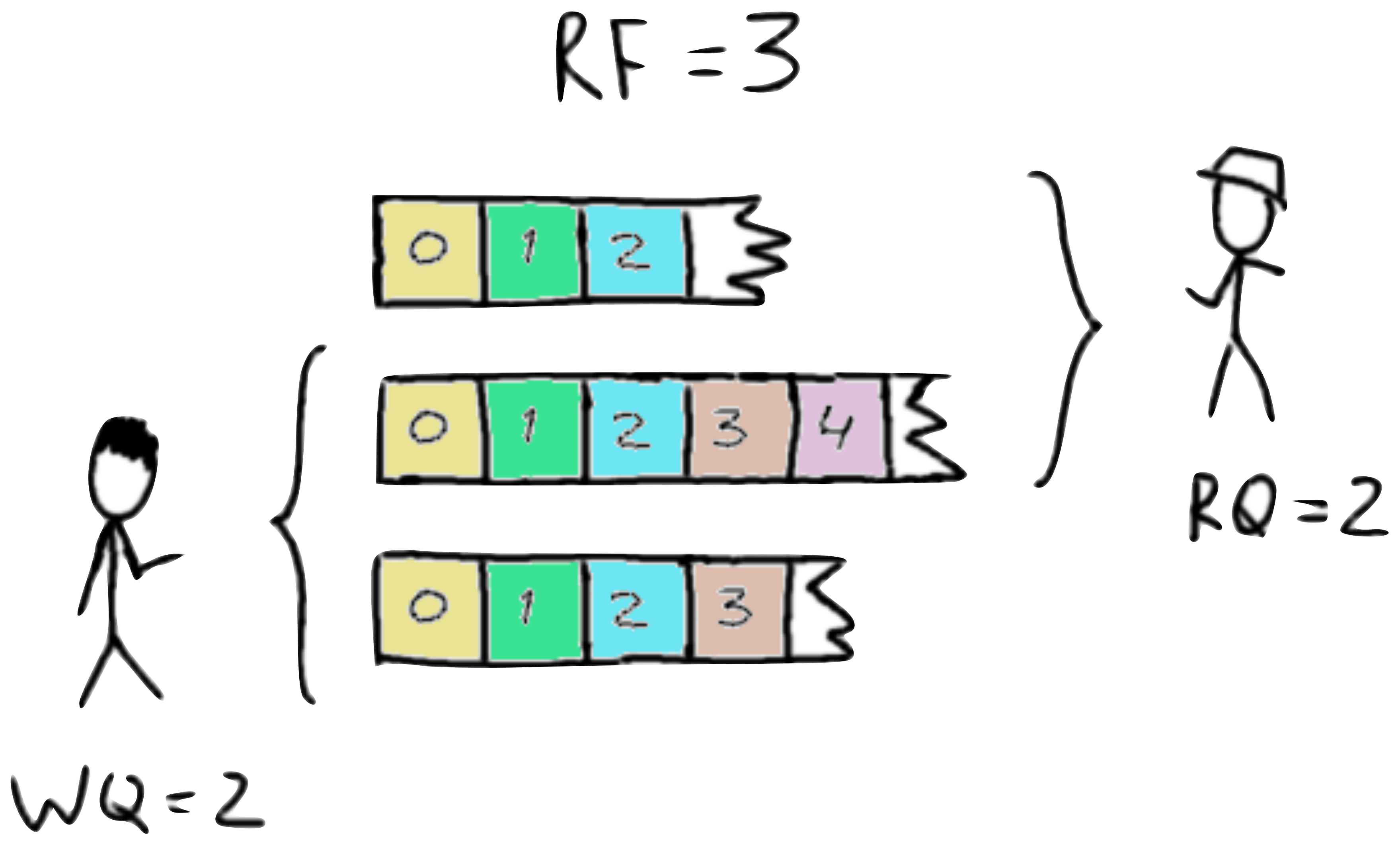
Given a subset of replicas, can we read the data?

Replica consistency => the longest replica is always enough

But how can we be sure about the number of acked records?

If <RQ replicas are alive, all of them may lack the last acked records

Otherwise the longest replica contains all the acked records



Journal Chunks: Analysis

Pros

Can handle mutable (append-only) data structures

Provides strong safety guarantees

Cons

Much higher disk and network bandwidth footprint

RF=3 provides 2-node fault tolerance for **immutable** chunks

RF=3, RQ=WQ=2 provides 1-node fault tolerance for **mutable** chunks

RF=3, RQ=1, WQ=3 provides 2-node fault tolerance for **mutable** chunks
but at the cost of increased latency

RF=5, RQ=WQ=3 provides 2-node fault tolerance
but at the cost of x5 bandwidth usage

Disk storage space?

Not actually a concern for WALs

Erasure Journal Chunks

Motivation

Erasure-coded blob chunk can save on disk space

But also on disk and network bandwidth

The latter are typically limiting factors for WALs

Also mind SSD wear (DWPDs are not that big nowadays)

Let’s write erasure-coded journals!

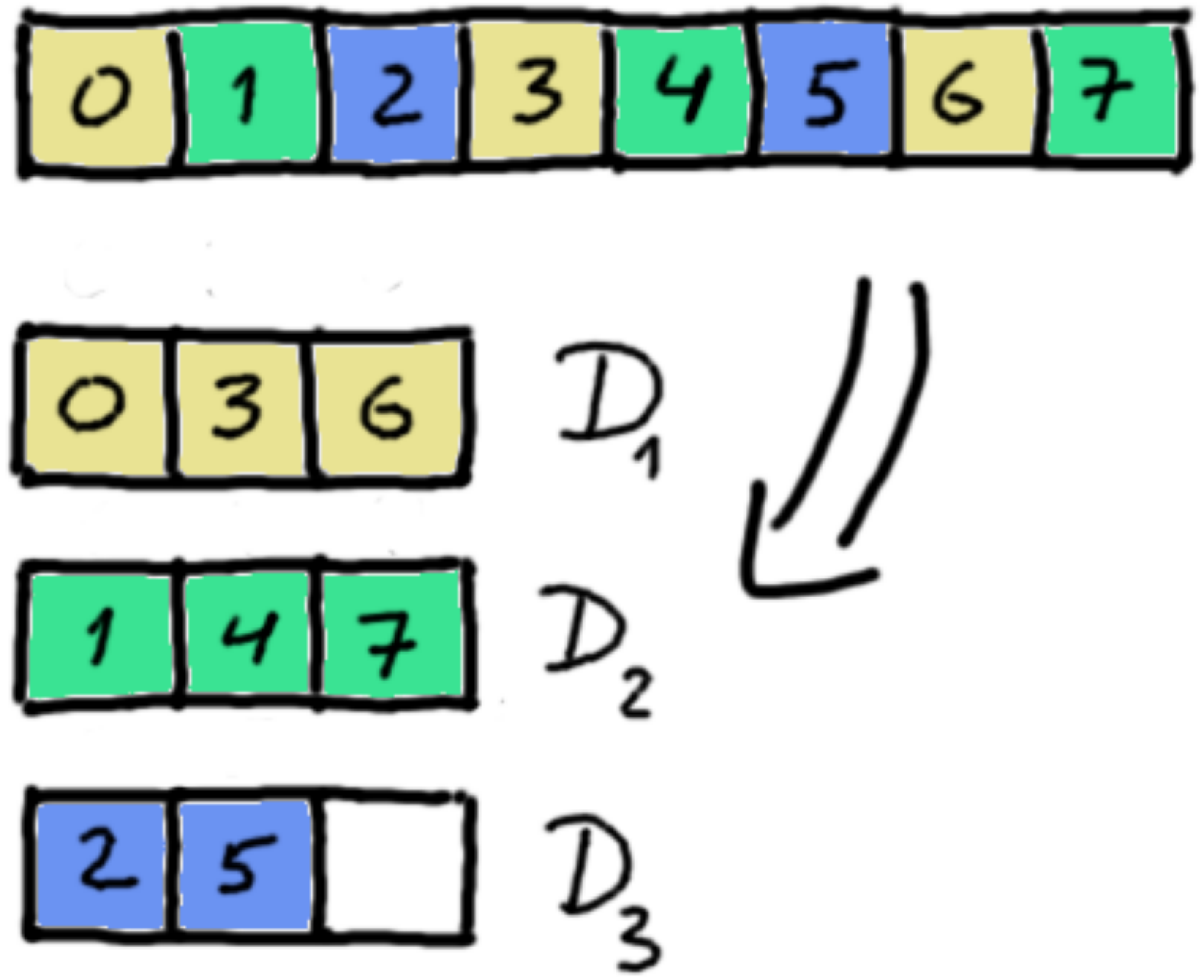
Splitting vs Striping

Recall that we did splitting for erasure-coded blobs

Reading a small range from an erasure-coded blob touches just one part

But we need the whole chunk data to be known in advance

Splitting is not an option for journals => will do striping



Quorums for Erasure-Coded Journals

ReedSolomon(N,K) is used for coding

RF = N + K (the total number of replicas)

Safety

RQ + WQ > RF for **replicated** journals

RQ + WQ > RF + N - 1 for **erasure-coded** journals

RHS declares the needed size of intersection between RQ and WQ

Higher RHS means more alive replicas are needed

Some Practical Scenario

Practical scenario

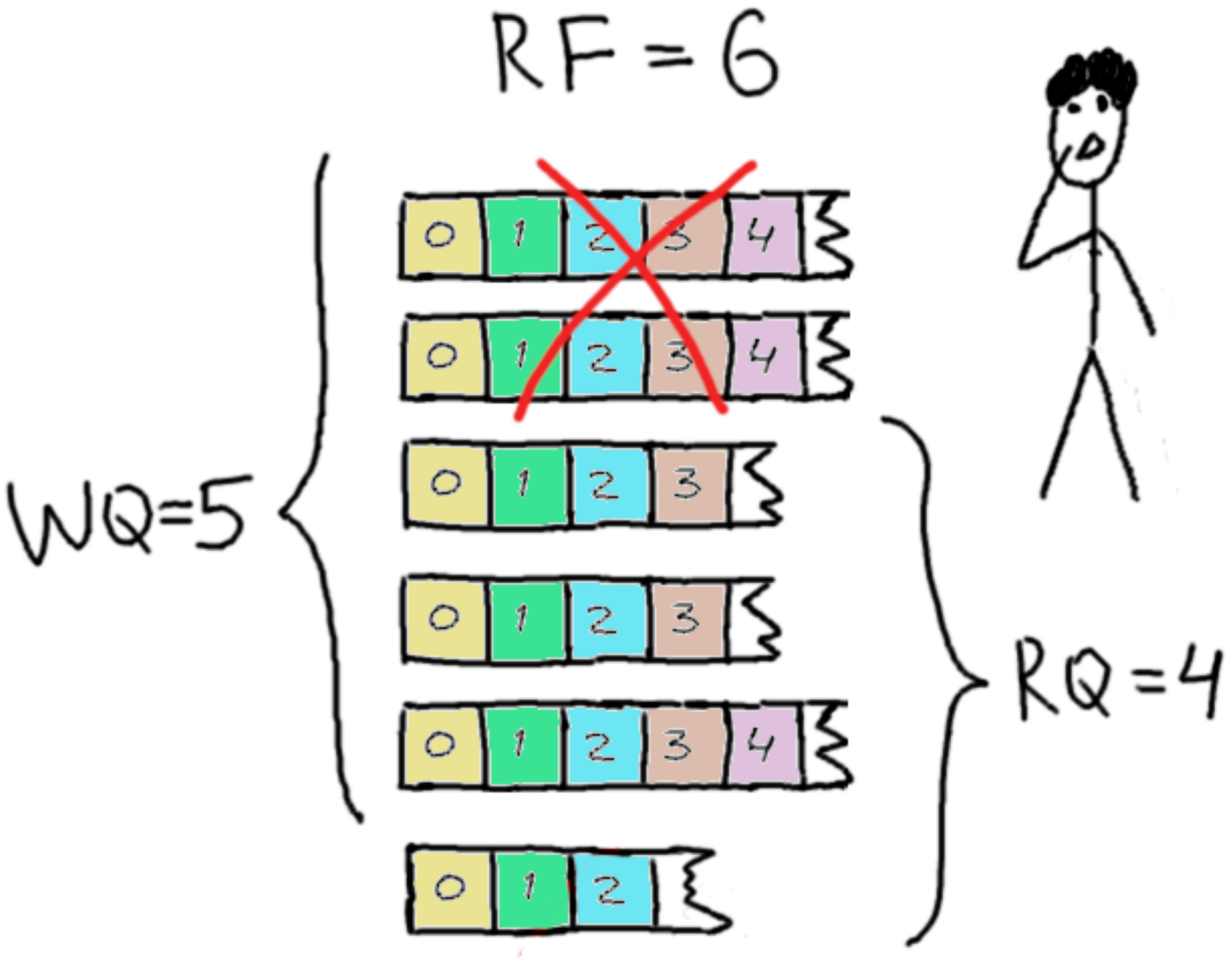
Reed-Solomon with N=3, K=3, RF=6, WQ = 5, RQ = 4
Tolerates up to 2 node failures
x2 bandwidth overhead

Previously for replicated journals

RF = 5, WQ = RQ = 3
Tolerates up to 2 node failures
x5 bandwidth overhead

Huge savings on bandwidth!

Safety argument



The Perils of Jerasure

- **Reed-Solomon with N=6, K=3**

```
DATA[0]   <00000000000000000000000000000000>
DATA[1]   <01010101010101010101010101010101>
DATA[2]   <02020202020202020202020202020202>
DATA[3]   <03030303030303030303030303030303>
DATA[4]   <04040404040404040404040404040404>
DATA[5]   <05050505050505050505050505050505>
```

The Perils of Jerasure

Reed-Solomon with N=6, K=3, Jerasure encoder

```
DATA[0]    <00000000000000000000000000000000>
DATA[1]    <01010101010101010101010101010101>
DATA[2]    <02020202020202020202020202020202>
DATA[3]    <03030303030303030303030303030303>
DATA[4]    <04040404040404040404040404040404>
DATA[5]    <05050505050505050505050505050505>
PARITY[0]  <01010101010101010101010101010101>
PARITY[1]  <03030303030303030606060606060606>
PARITY[2]  <04040404040404040505050505050505>
```

The Perils of Jerasure

Reed-Solomon with N=6, K=3, ISA-L encoder

```
DATA[0]   <00000000000000000000000000000000>
DATA[1]   <01010101010101010101010101010101>
DATA[2]   <02020202020202020202020202020202>
DATA[3]   <03030303030303030303030303030303>
DATA[4]   <04040404040404040404040404040404>
DATA[5]   <05050505050505050505050505050505>
PARITY[0] <03030303030303030303030303030303>
PARITY[1] <0a0a0a0a0a0a0a0a0a0a0a0a0a0a0a0a>
PARITY[2] <24242424242424242424242424242424
```

The Perils of Jerasure

Jerasure does not act byte-per-byte

For erasure journals, read and write portions are typically not aligned

This complicates reading and decoding arbitrary ranges

ISA-L is a “pure” Reed-Solomon encoder

For erasure-coded journals, we exclusively use ISA-L

Also ISA-L is just faster

Old chunks are still Jerasure-encoded and will remain as such

Cross-DC Case

ReedSolomon(3,3) works nicely for 3 DCs

Place 6 replicas in 2+2+2 crossDC arrangement

Any 2 nodes can go down simultaneously

Any DC can go down

Also nice

Cross-DC links are not cheap

Disk bandwidth savings imply cross-DC network bandwidth savings

Текст

Conclusions

Things to Remember

- Erasure coding is a real thing, you can rely on it
- You can store most of your cold data in erasure
- Erasure writes are not very cheap but modern processors can run encoding-decoding really fast
- Reads may suffer hotspots but on-the-fly repair may help (but mind the increased bandwidth usage and CPU utilization)
- Erasure coding not only saves on disk space but also on bandwidth

Яндекс

Thank you

Maxim Babenko

Head of distributed computing technologies team

✉ babenko@yandex-team.ru

@maxim_babenko