Next Generation Erasure Coding

Bill Scales

Vision

Optimize I/O performance for Erasure Coded Pools

- Enhance performance to be similar to Replicated Pools

  but with lower Total Cost of Ownership (TCO)

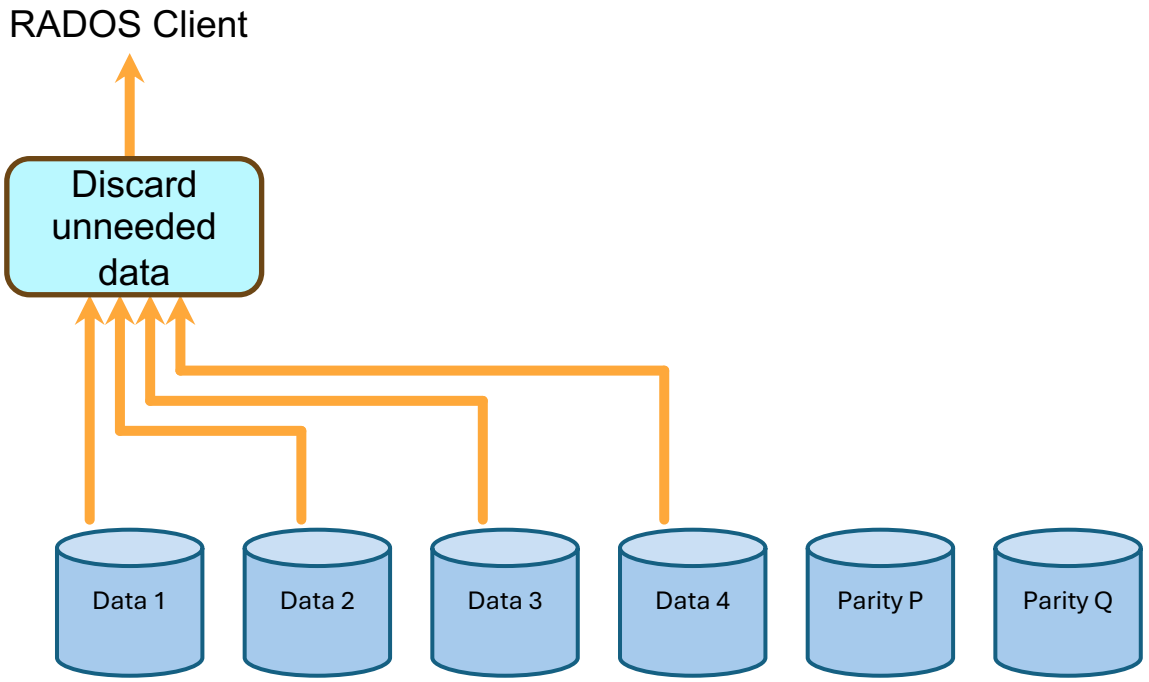
- Make Erasure Coded pools more viable for use with block and file

Performance Bottlenecks

- Bandwidth – typically limited by Network
- I/Os per second (IOPs) – limited by CPU and amplification of 1 client I/O to many backend I/Os
- Latency – increased by extra network hops and extra backend I/Os

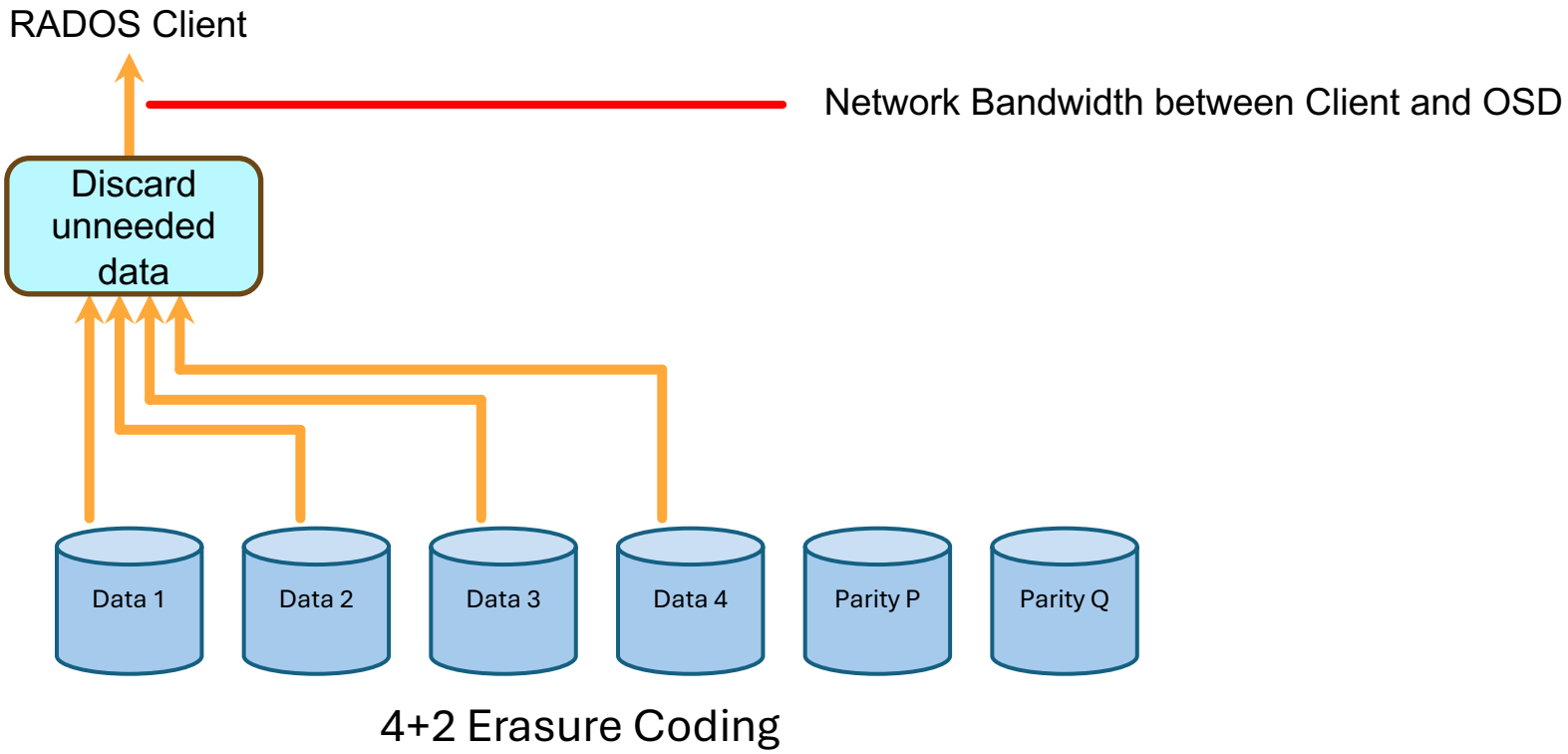
Reads

Current Implementation of Read



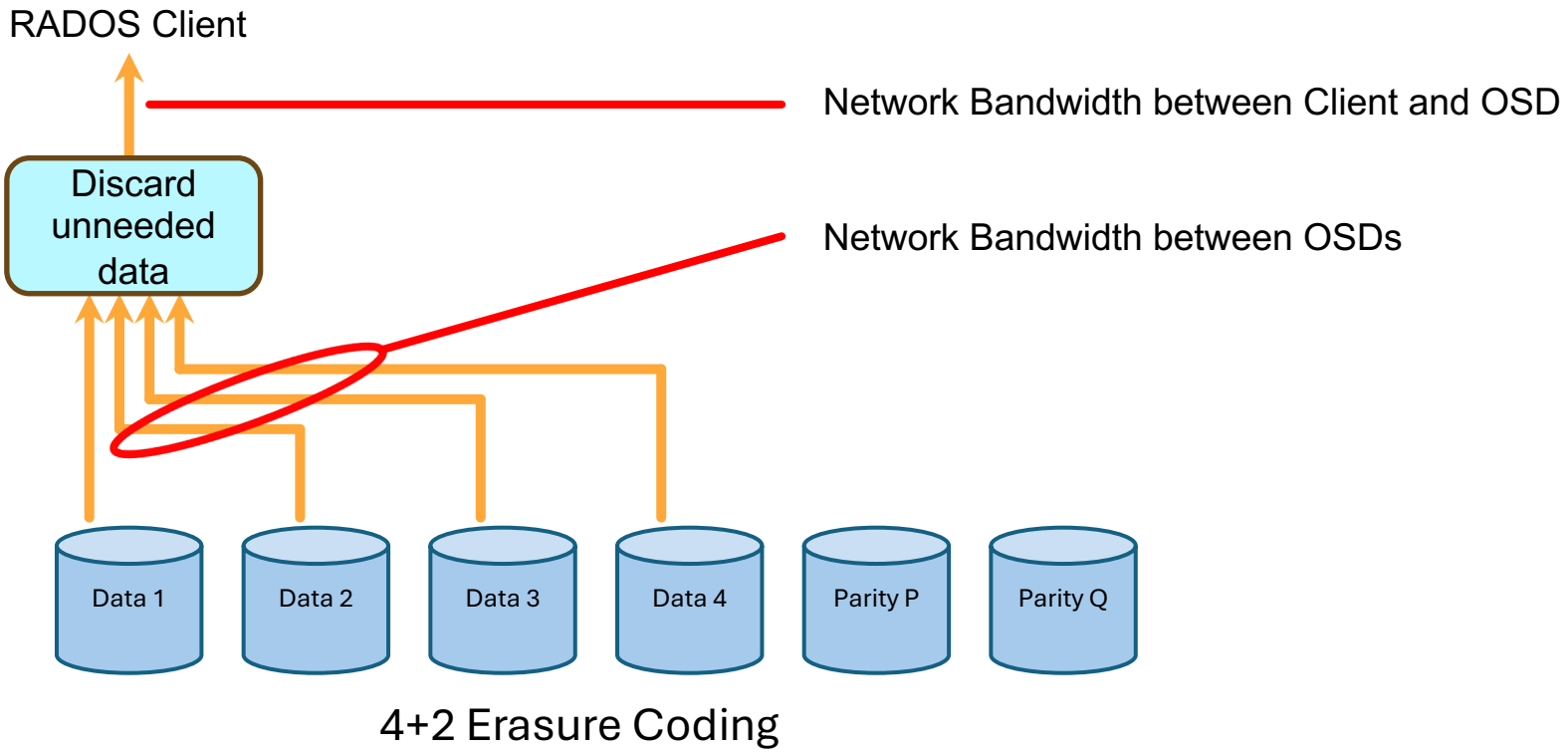
4+2 Erasure Coding

Current Implementation of Read



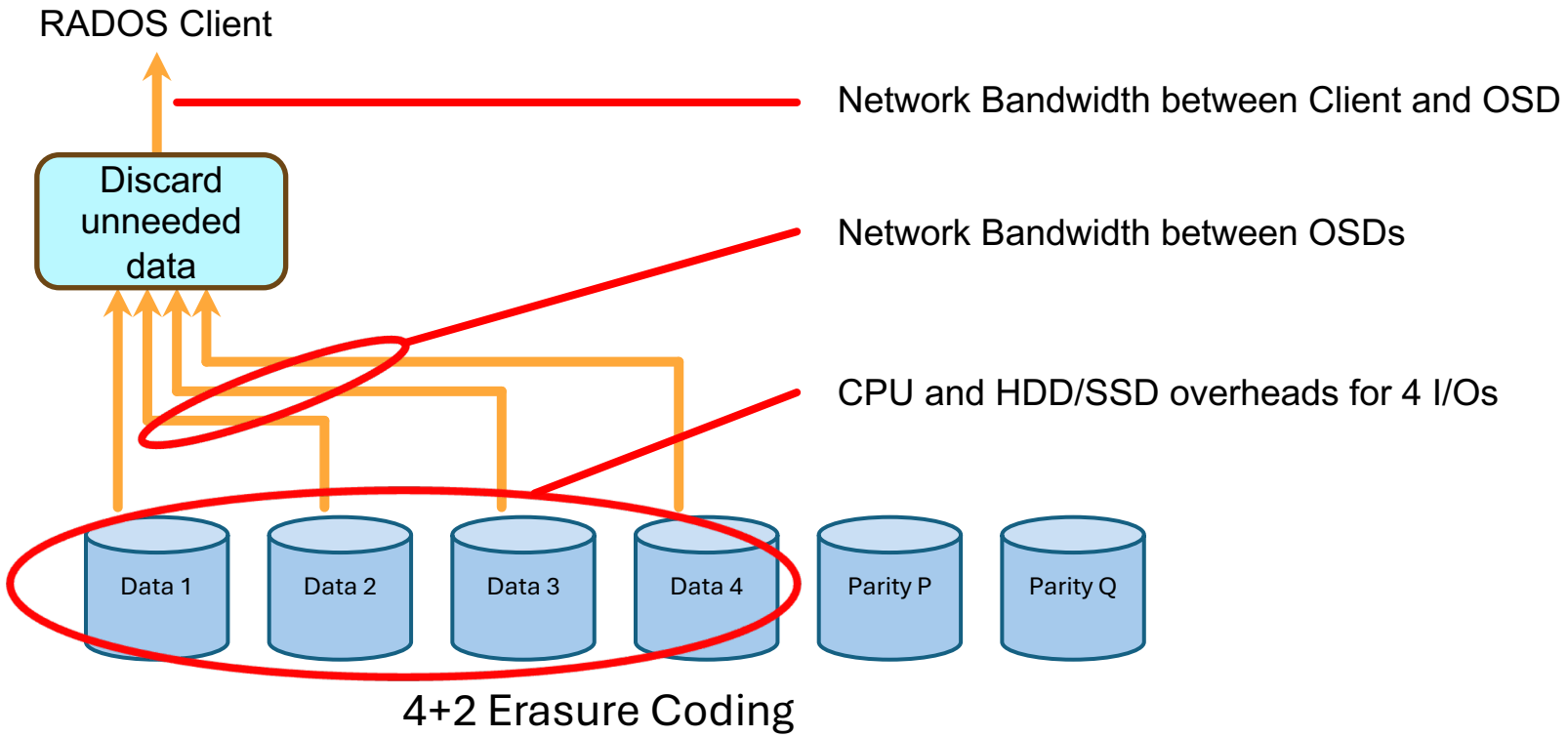
4+2 Erasure Coding

Current Implementation of Read

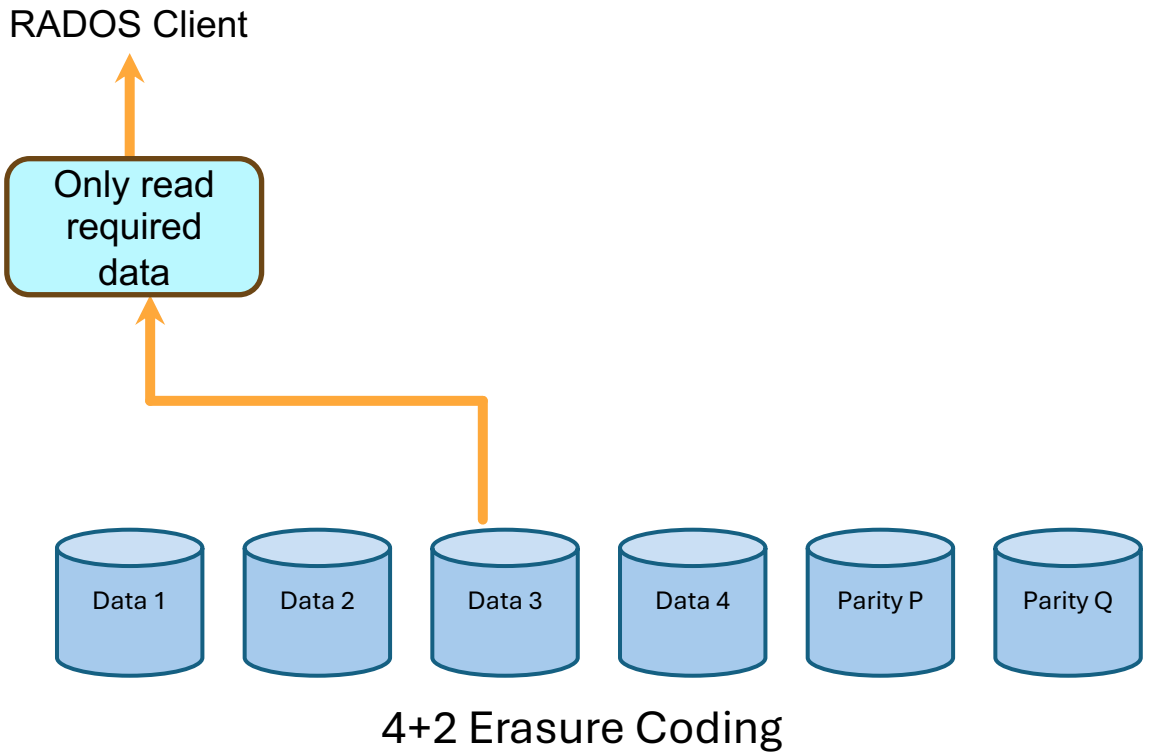


4+2 Erasure Coding

Current Implementation of Read

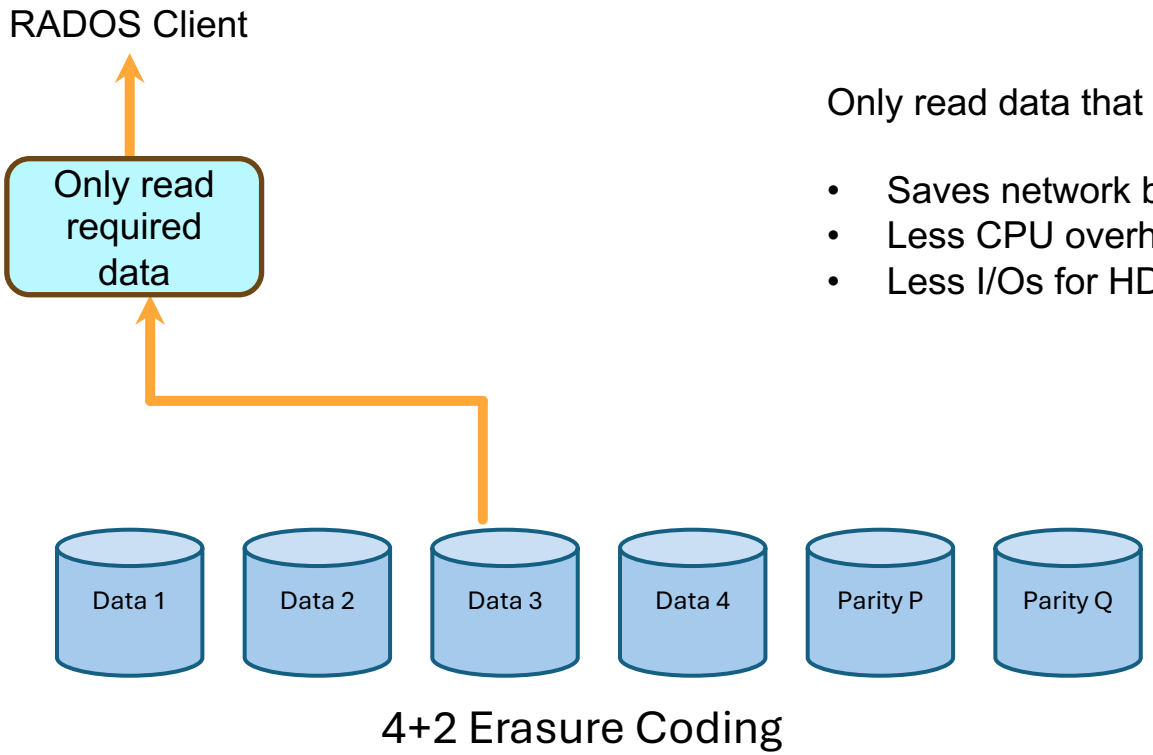


Partial Read



4+2 Erasure Coding

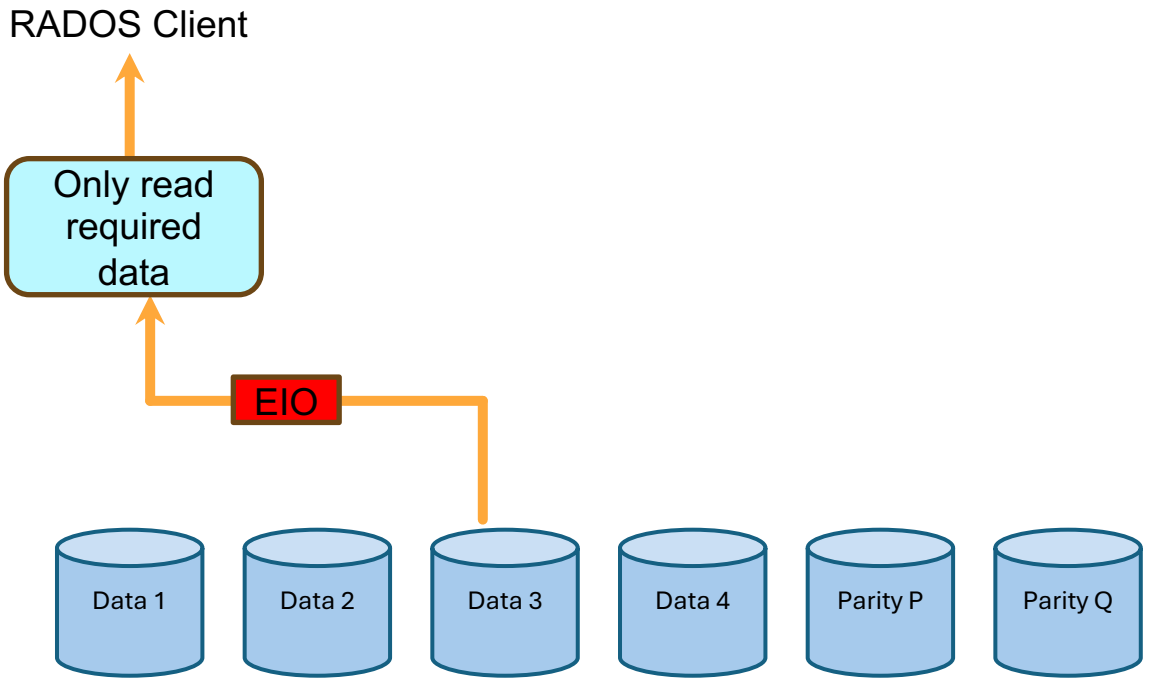
Partial Read



Only read data that is required

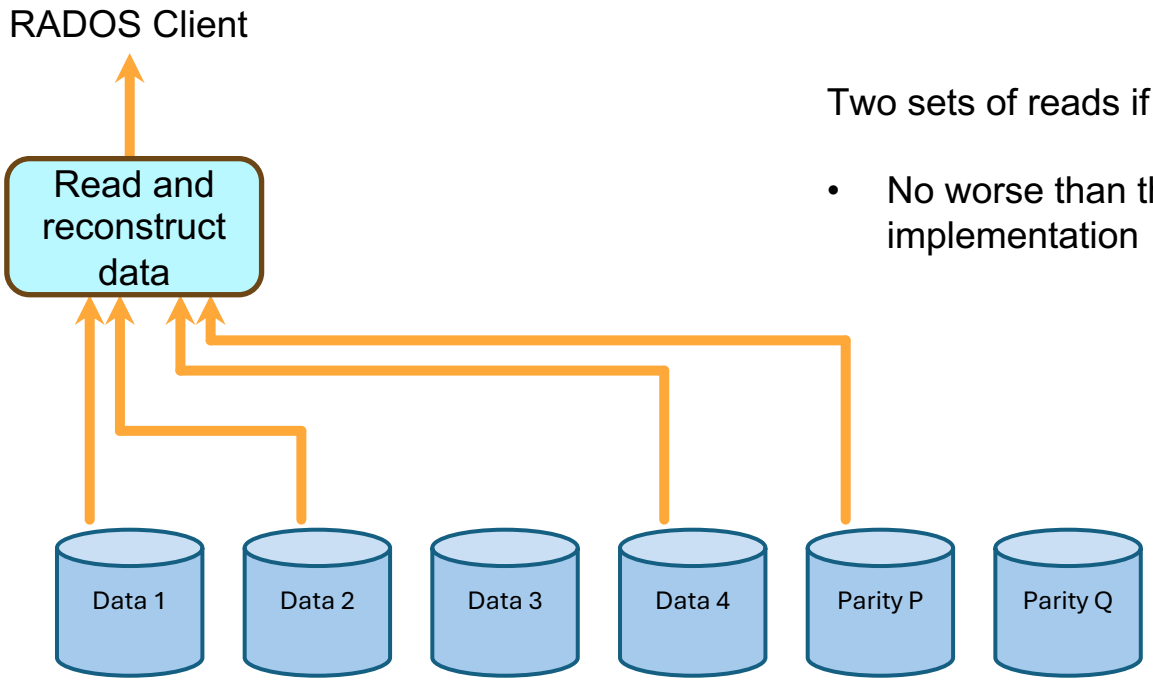
- Saves network bandwidth
- Less CPU overheads
- Less I/Os for HDDs/SSDs

Partial Read with Media Error



4+2 Erasure Coding

Partial Read with Media Error

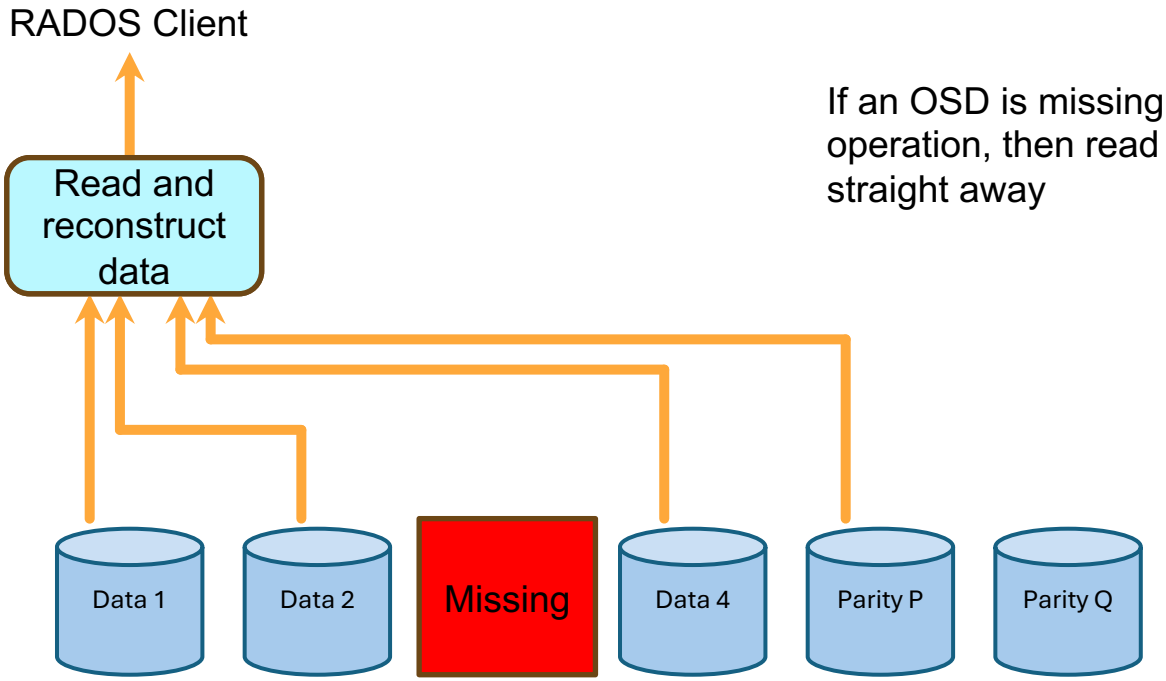


4+2 Erasure Coding

Two sets of reads if an error is encountered

- No worse than the current implementation

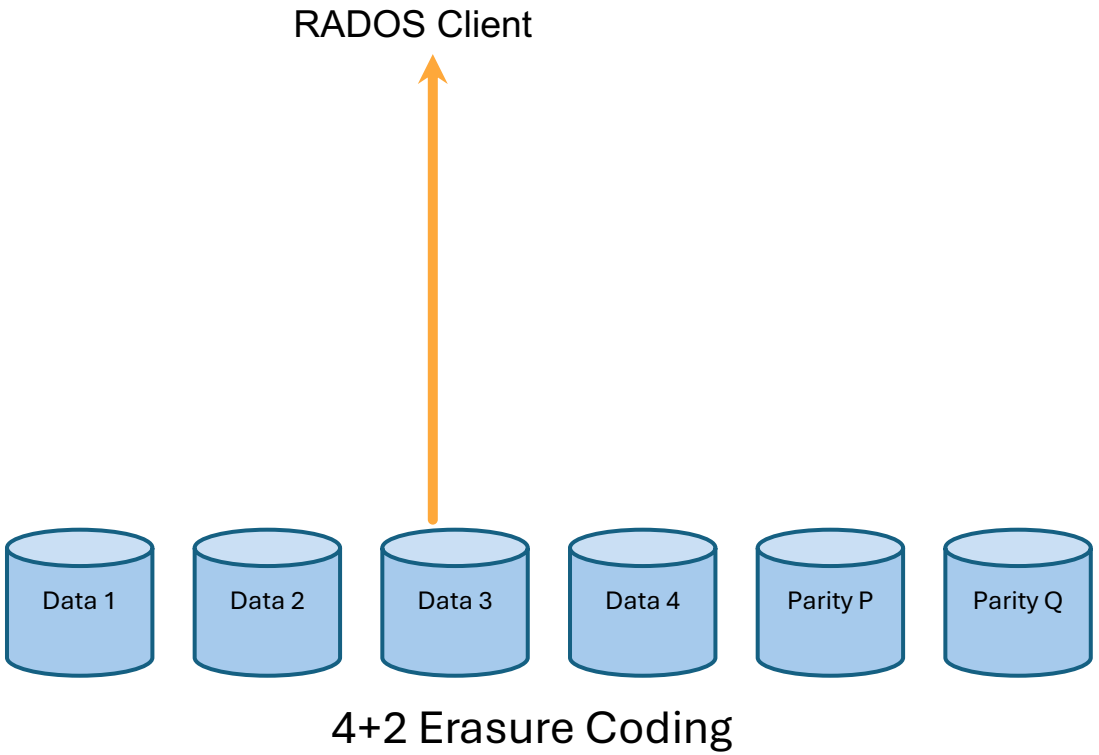
Partial Read with Missing OSD



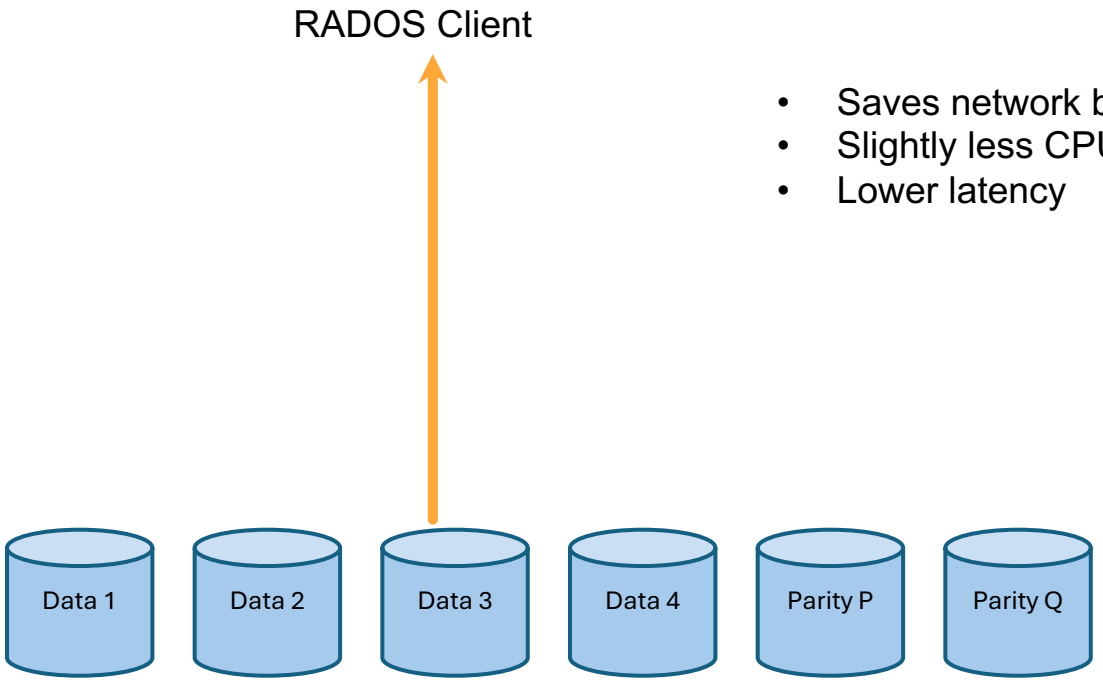
4+2 Erasure Coding

If an OSD is missing at start of read operation, then read and reconstruct the data straight away

Direct Read


RADOS Client
Data 1
Data 2
Data 3
Data 4
Parity P
Parity Q
4+2 Erasure Coding

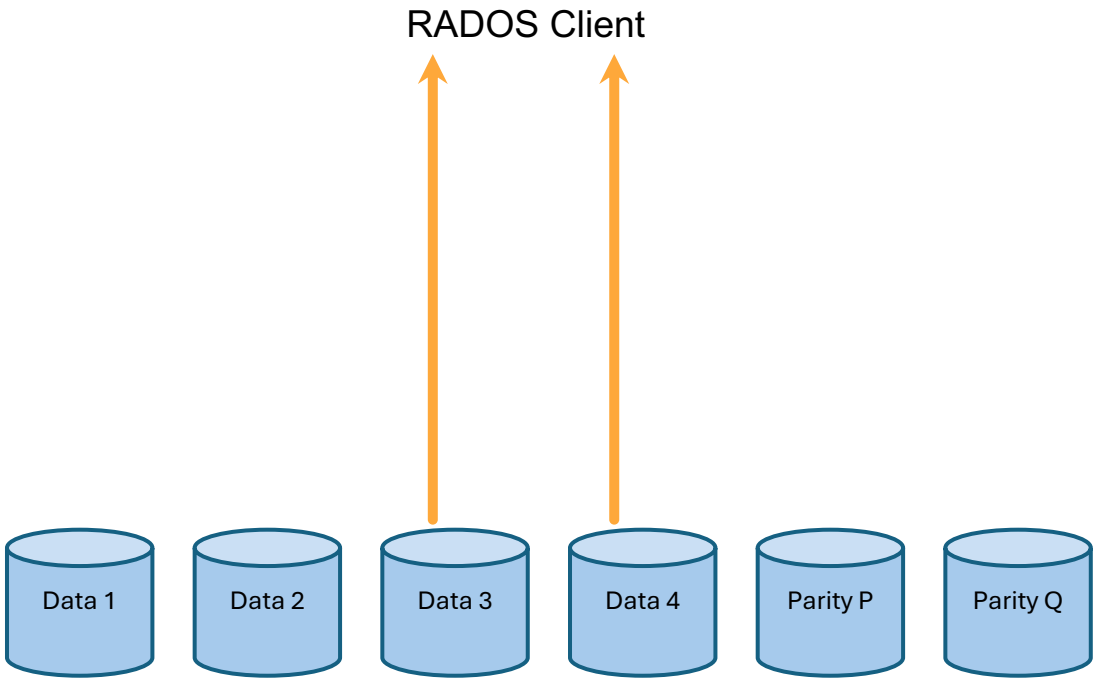
Direct Read



- Saves network bandwidth
- Slightly less CPU overhead
- Lower latency

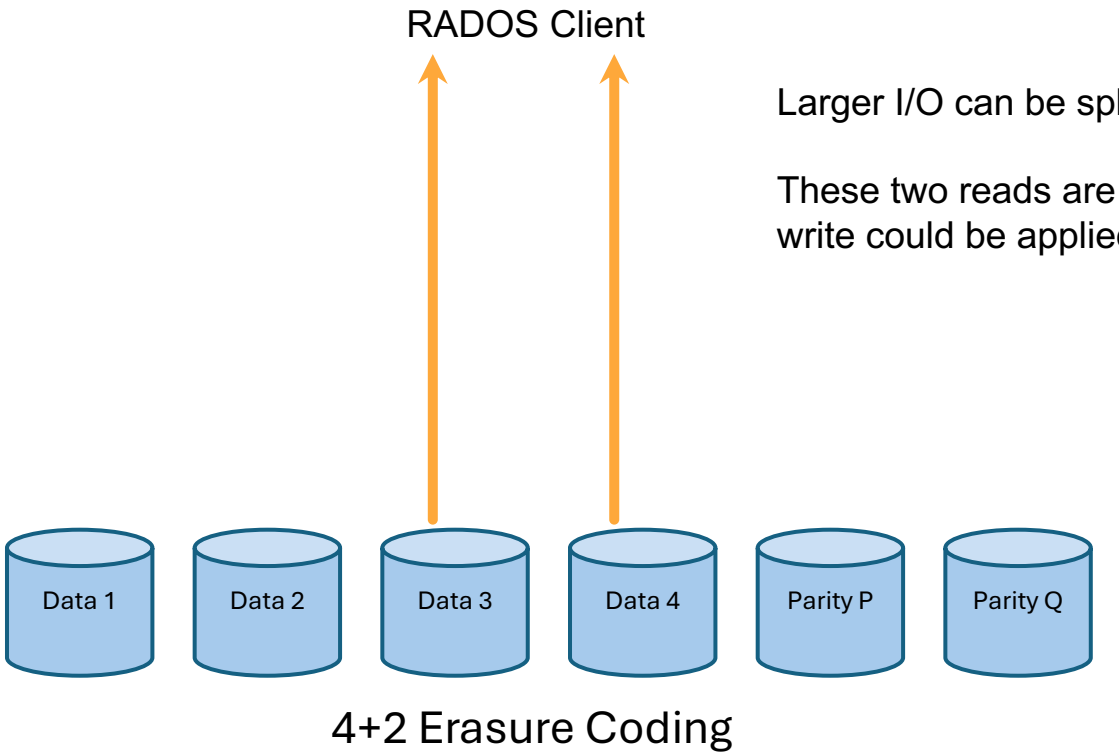
4+2 Erasure Coding

Direct Read



4+2 Erasure Coding

Direct Read

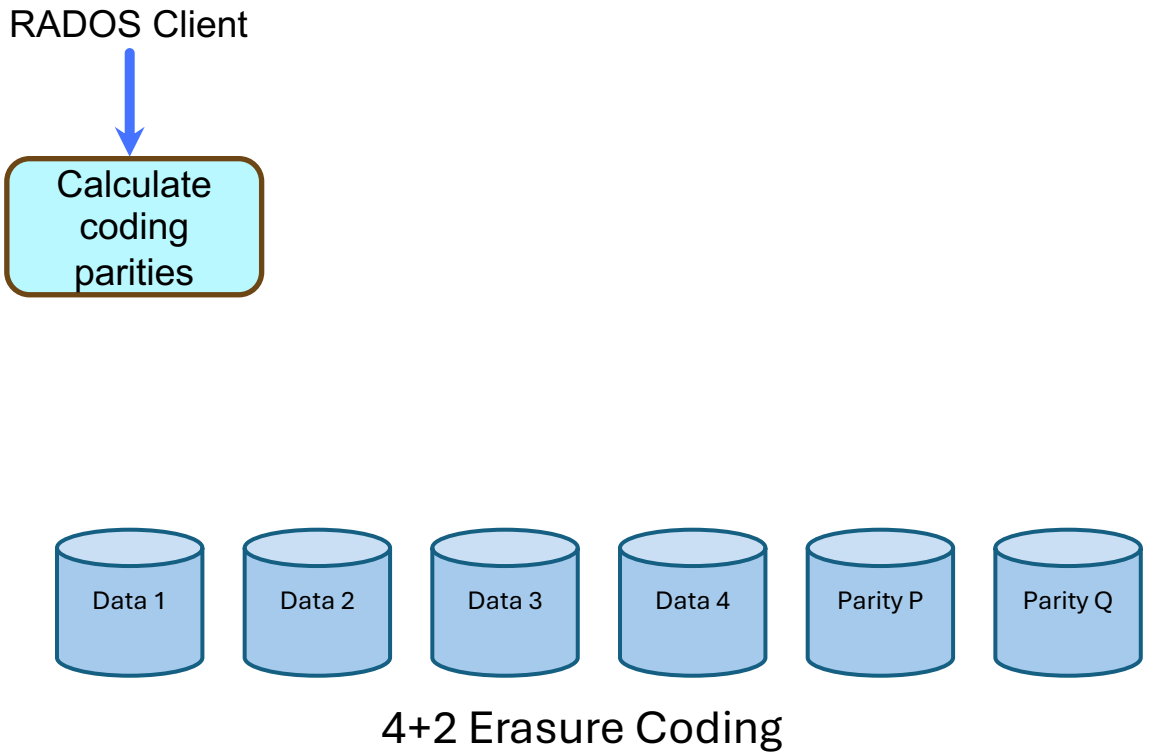


Larger I/O can be split by the client

These two reads are no longer atomic, a write could be applied between the two reads

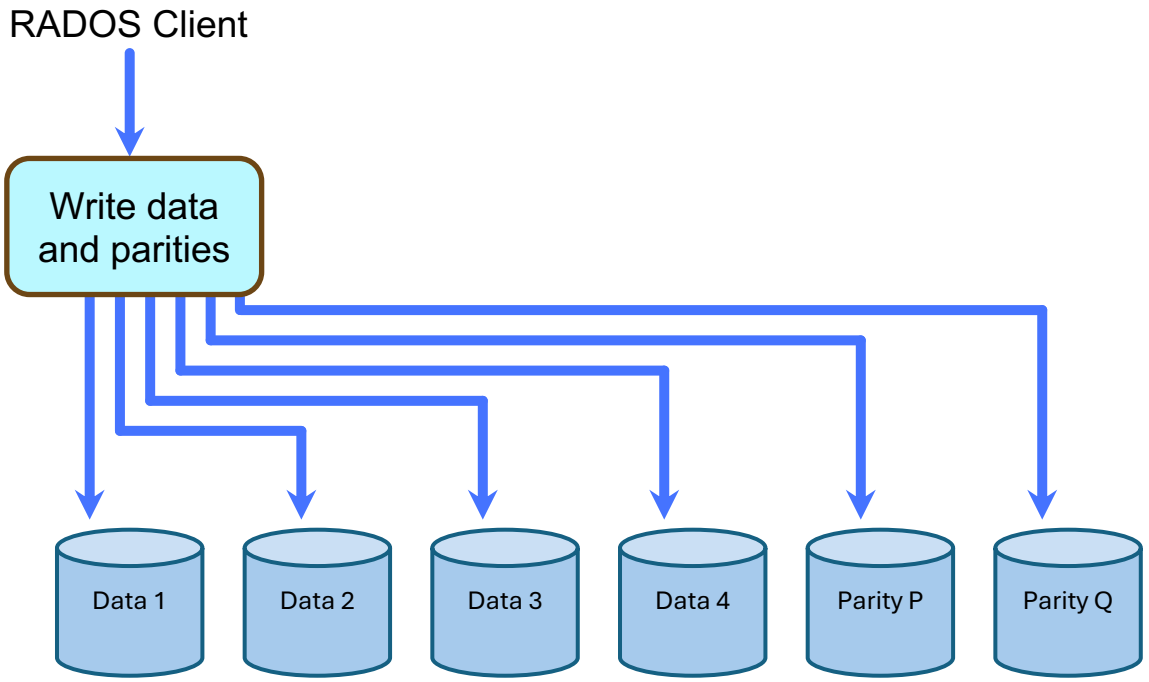
Writes and Overwrites

Current Implementation of Write



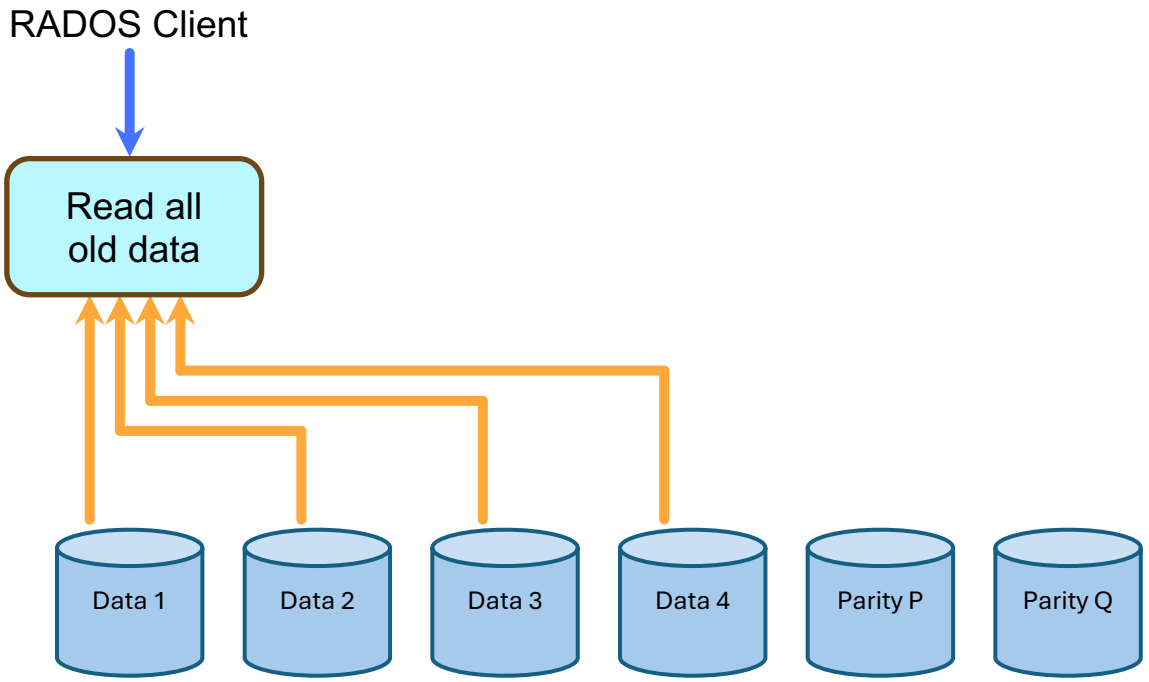
4+2 Erasure Coding

Current Implementation of Write



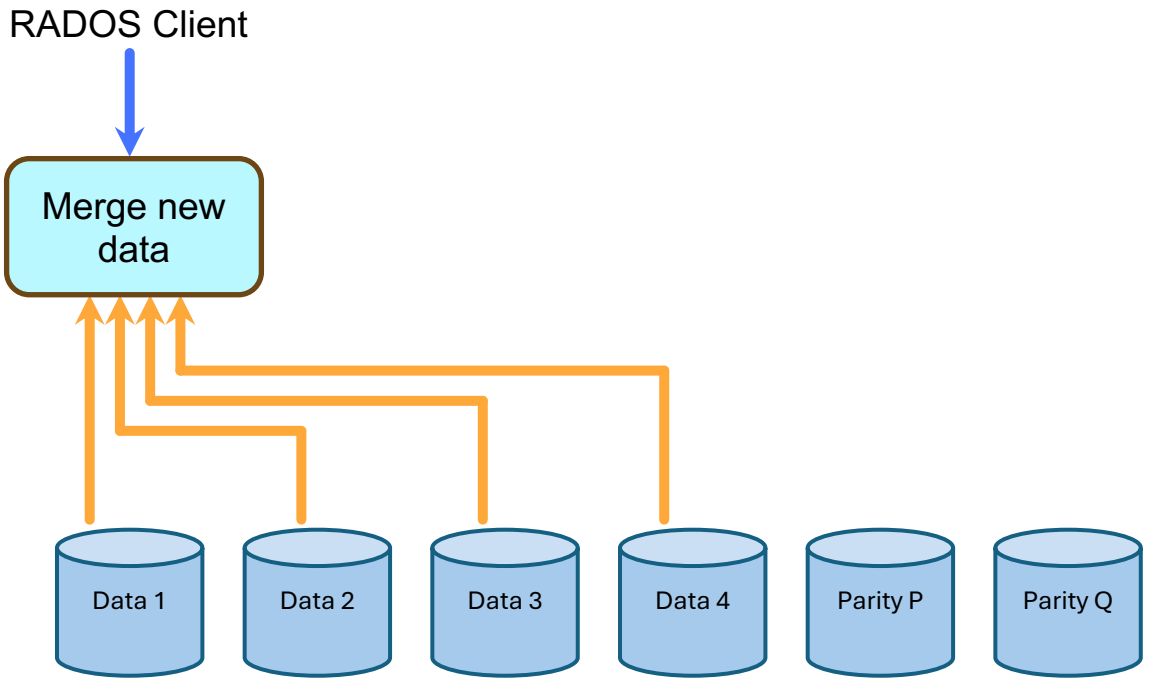
4+2 Erasure Coding

Current Implementation of Overwrite



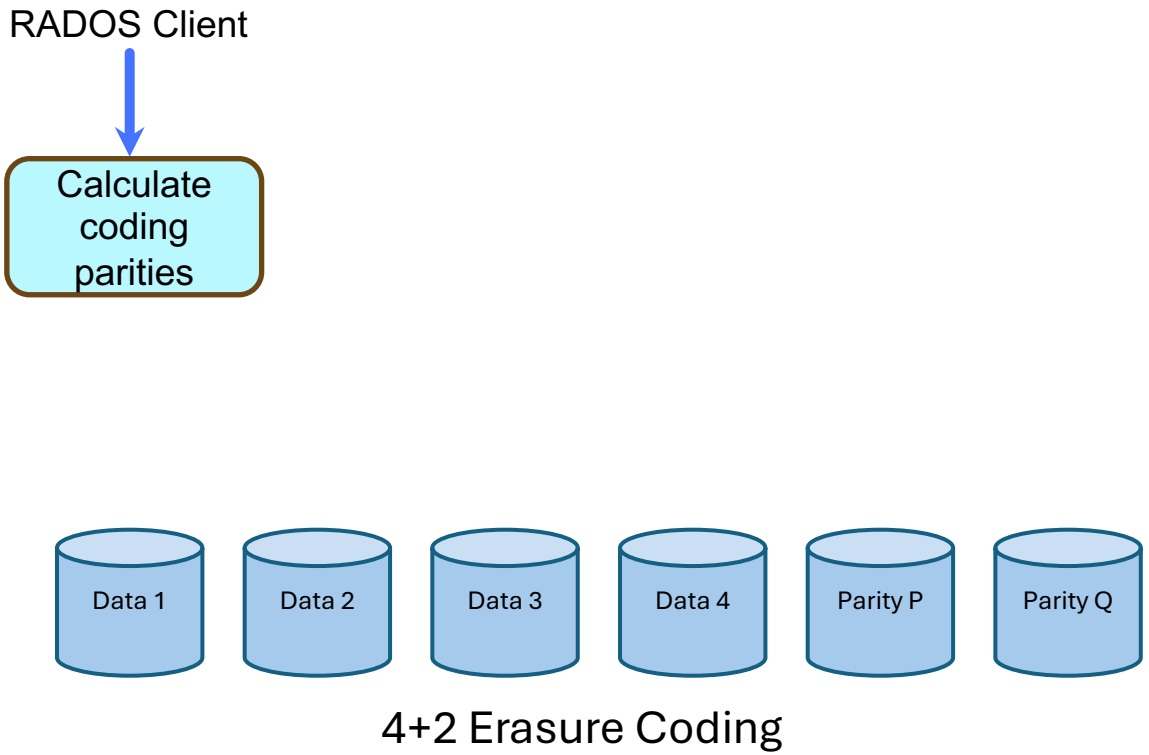
4+2 Erasure Coding

Current Implementation of Overwrite

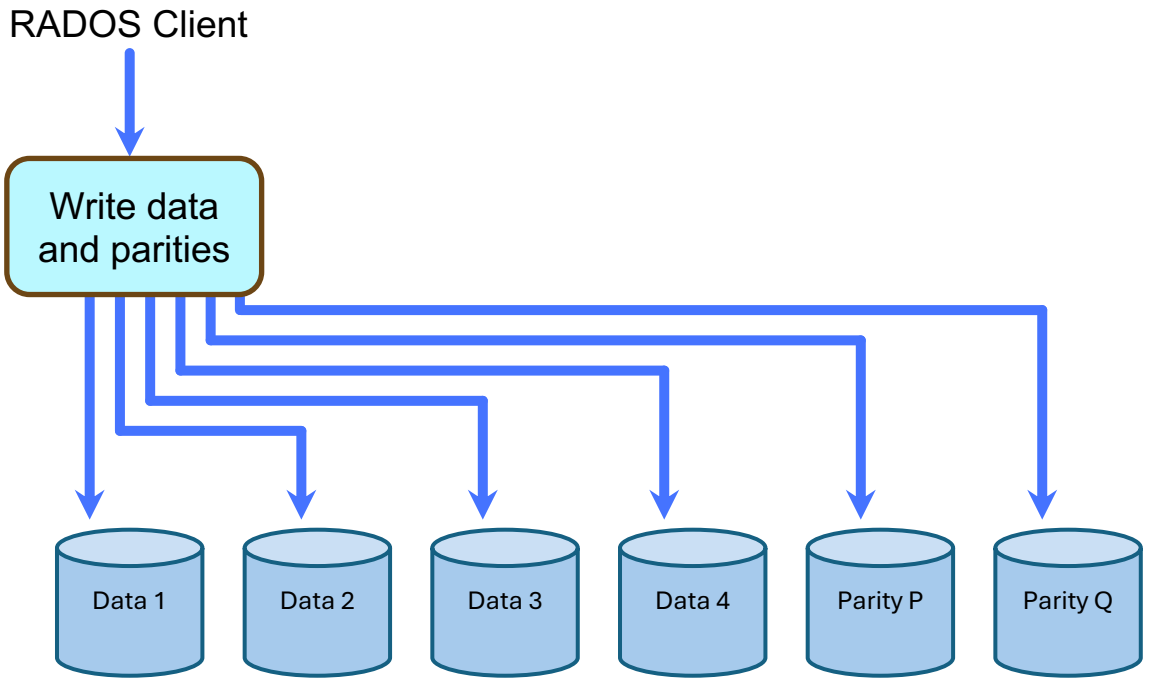


4+2 Erasure Coding

Current Implementation of Overwrite

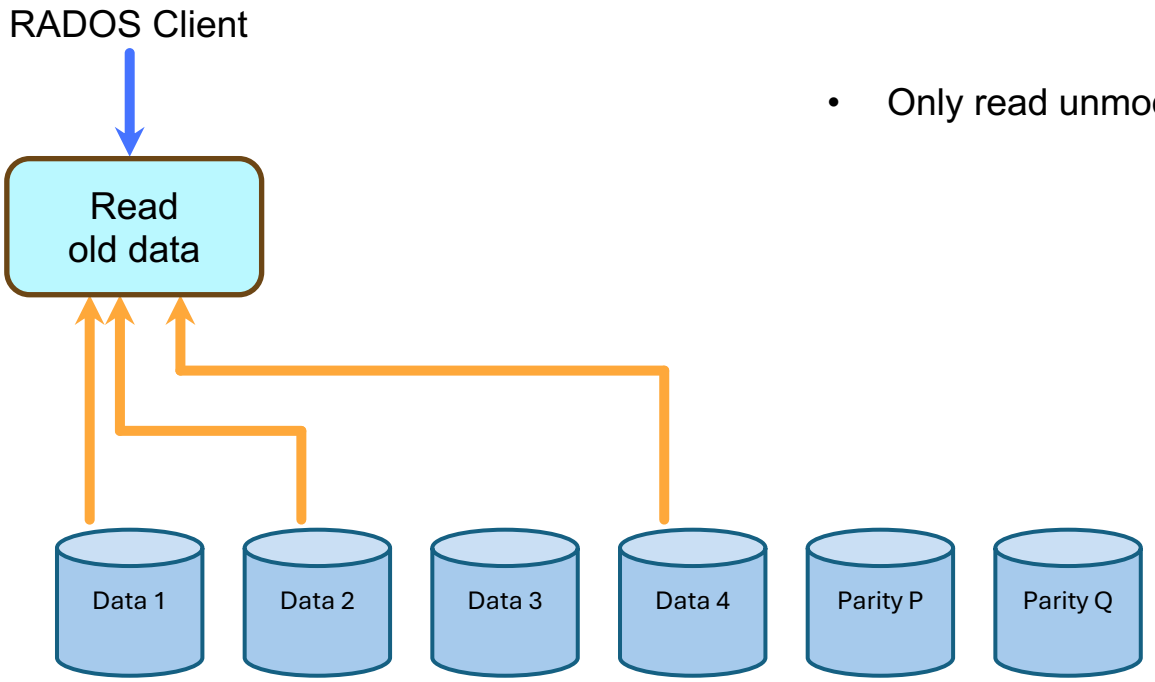


Current Implementation of Overwrite



4+2 Erasure Coding

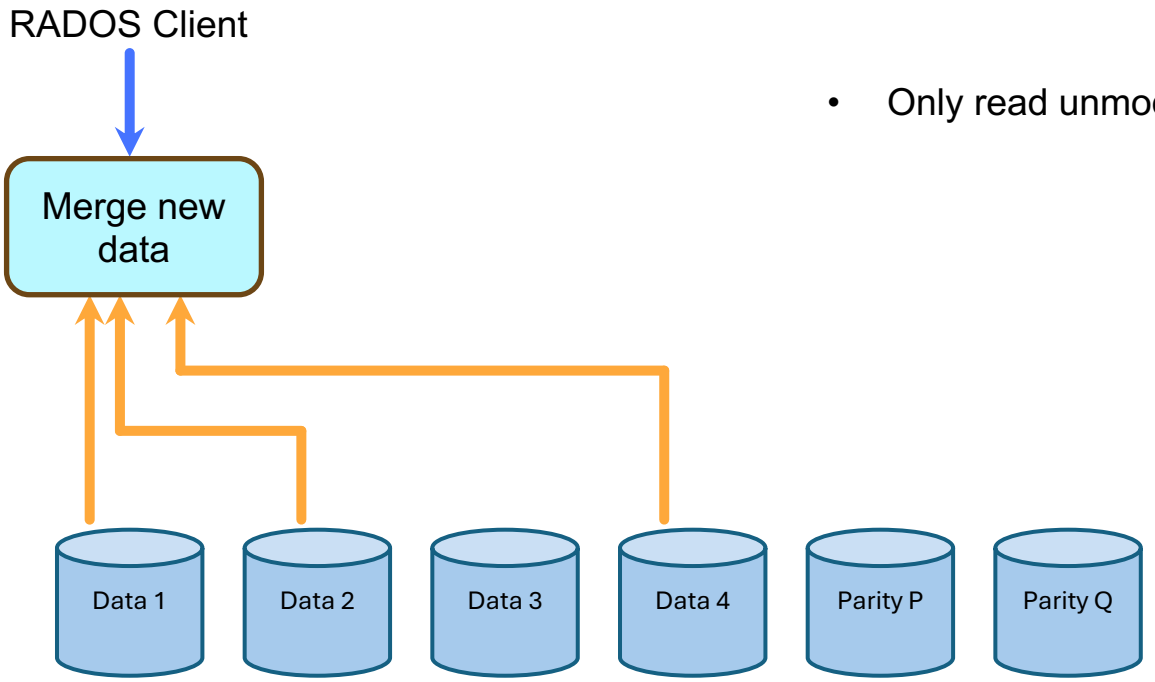
Simple Optimizations for Overwrite



- Only read unmodified data

4+2 Erasure Coding

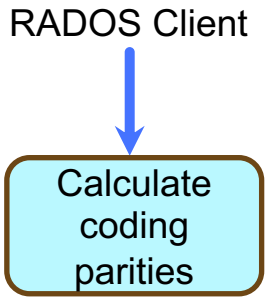
Simple Optimizations for Overwrite



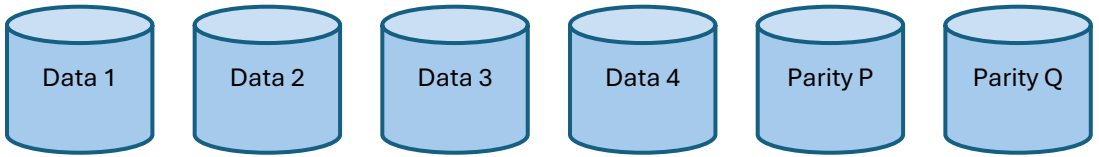
- Only read unmodified data

4+2 Erasure Coding

Simple Optimizations for Overwrite

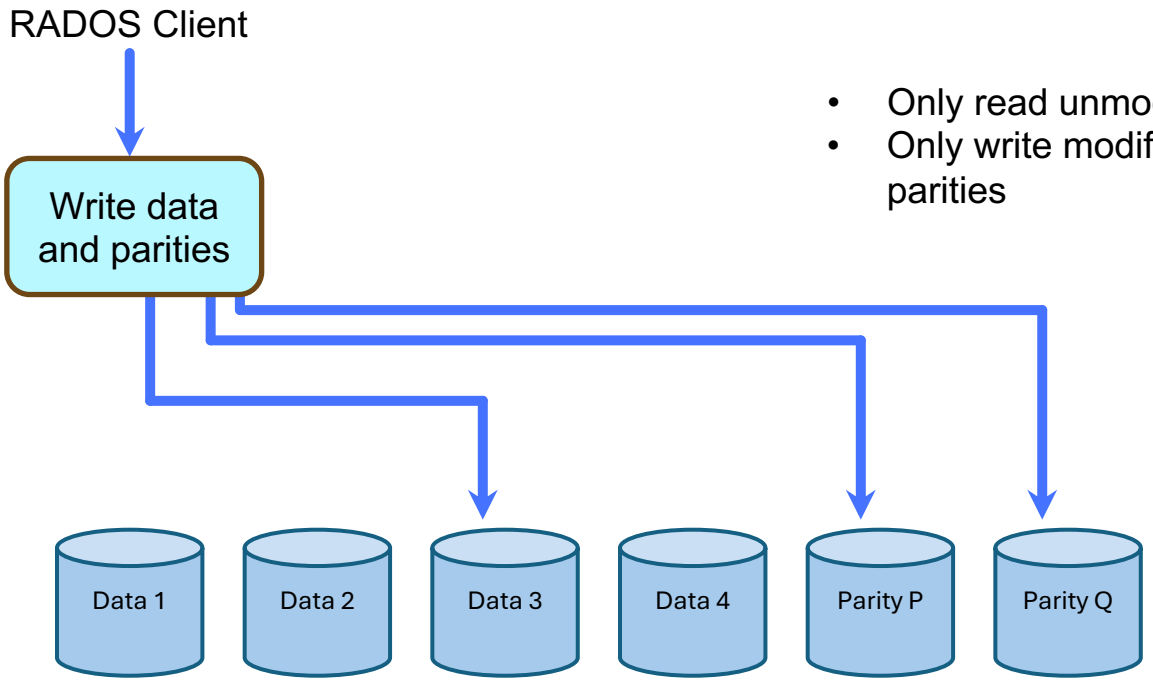


- Only read unmodified data



4+2 Erasure Coding

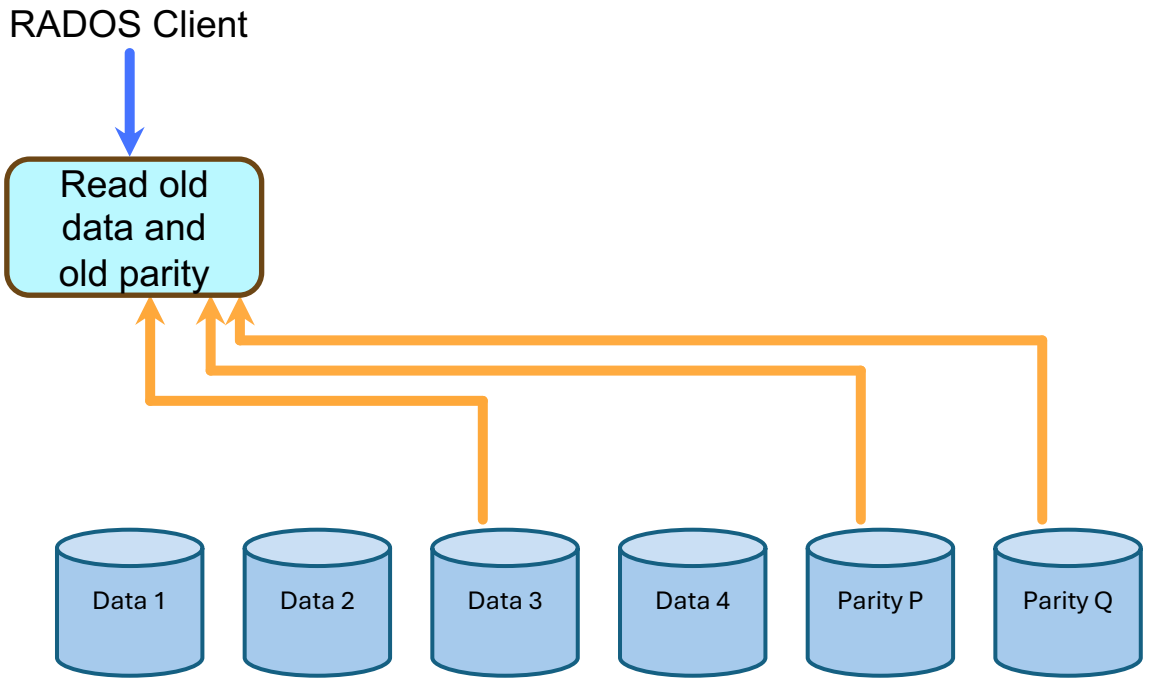
Simple Optimizations for Overwrite



- Only read unmodified data
- Only write modified data and coding parities

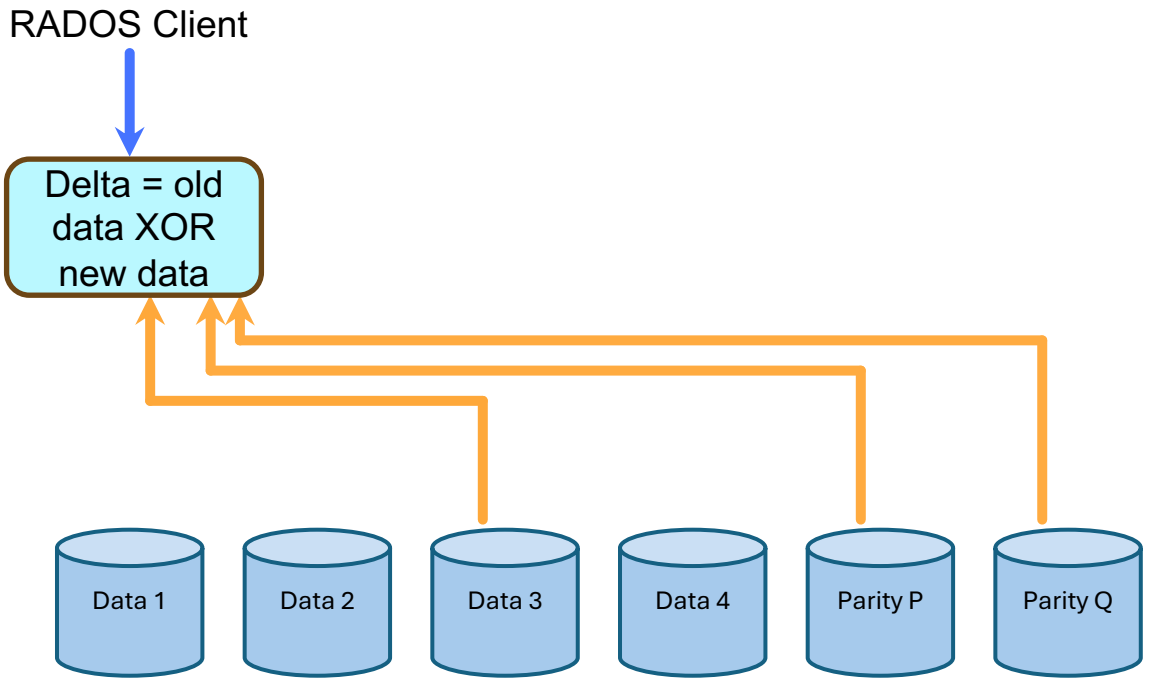
4+2 Erasure Coding

Parity-Delta-Write Optimization



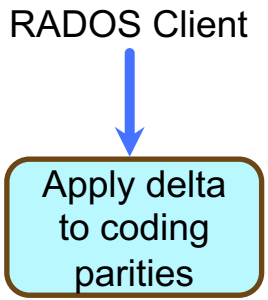
4+2 Erasure Coding

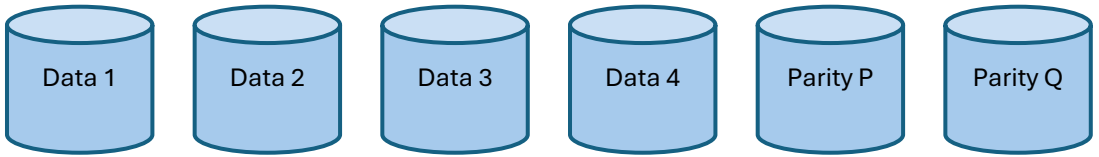
Parity-Delta-Write Optimization



4+2 Erasure Coding

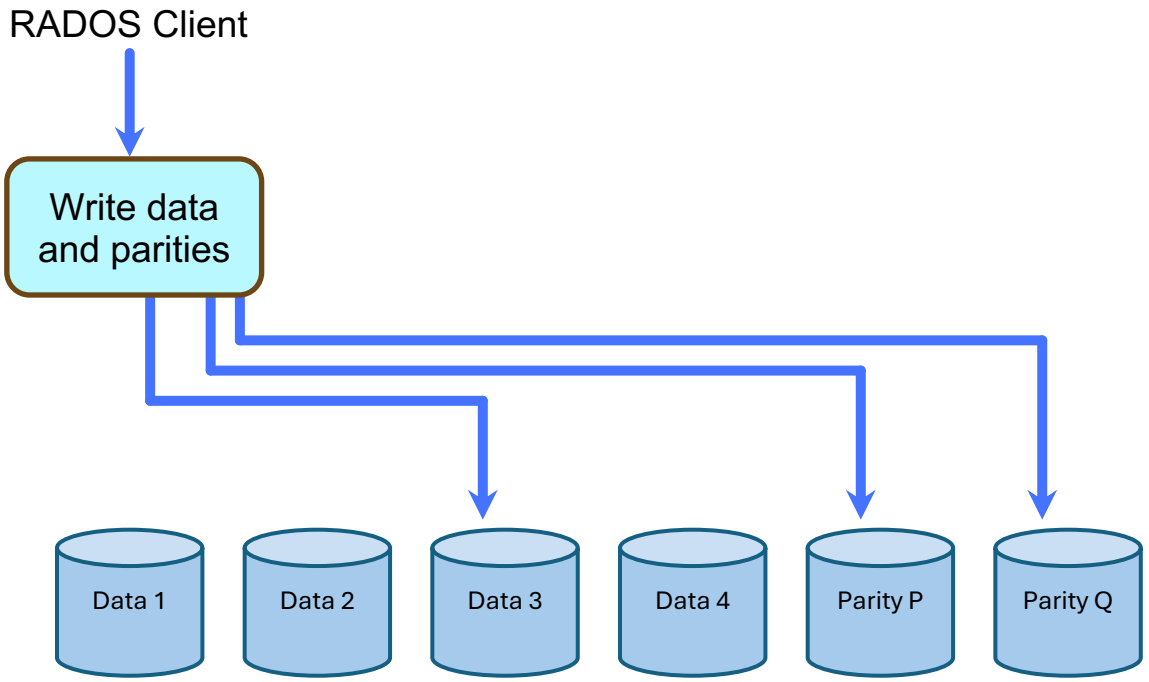
Parity-Delta-Write Optimization





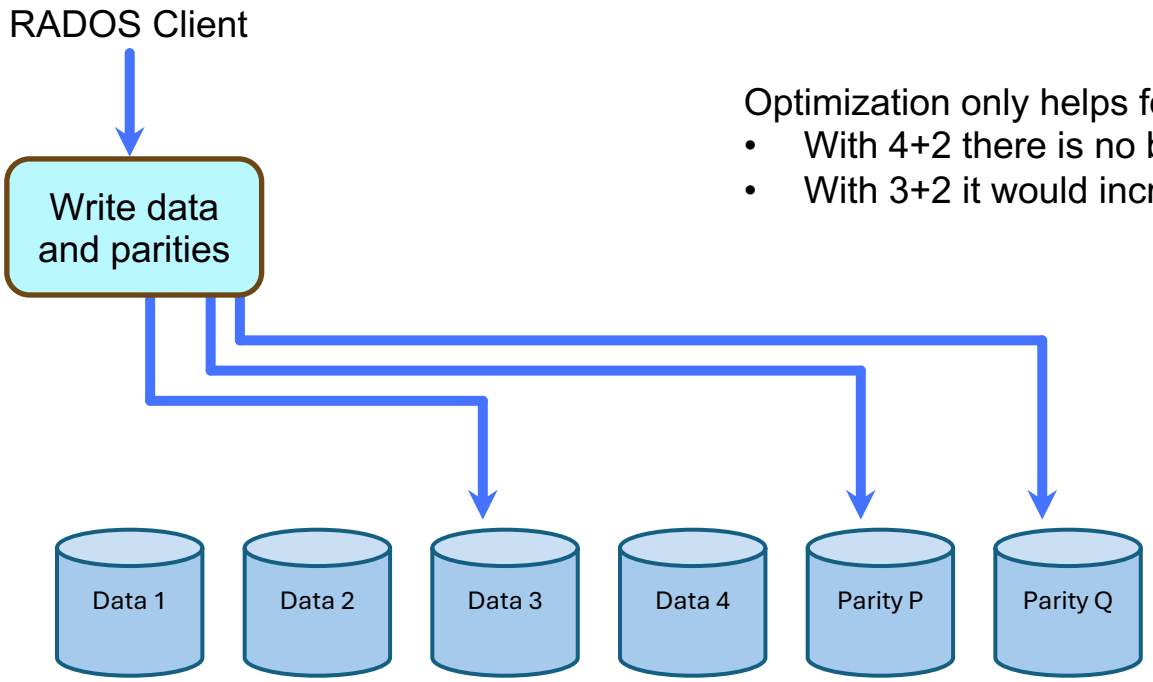
4+2 Erasure Coding

Parity-Delta-Write Optimization



4+2 Erasure Coding

Parity-Delta-Write Optimization

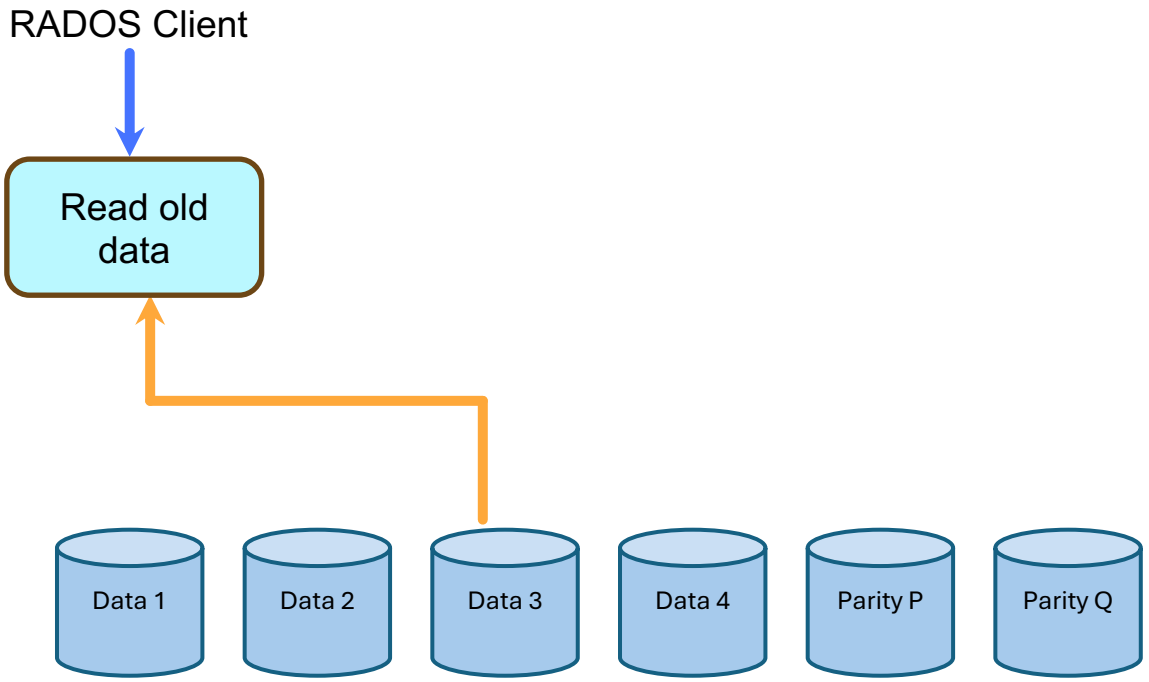


4+2 Erasure Coding

Optimization only helps for larger values of K

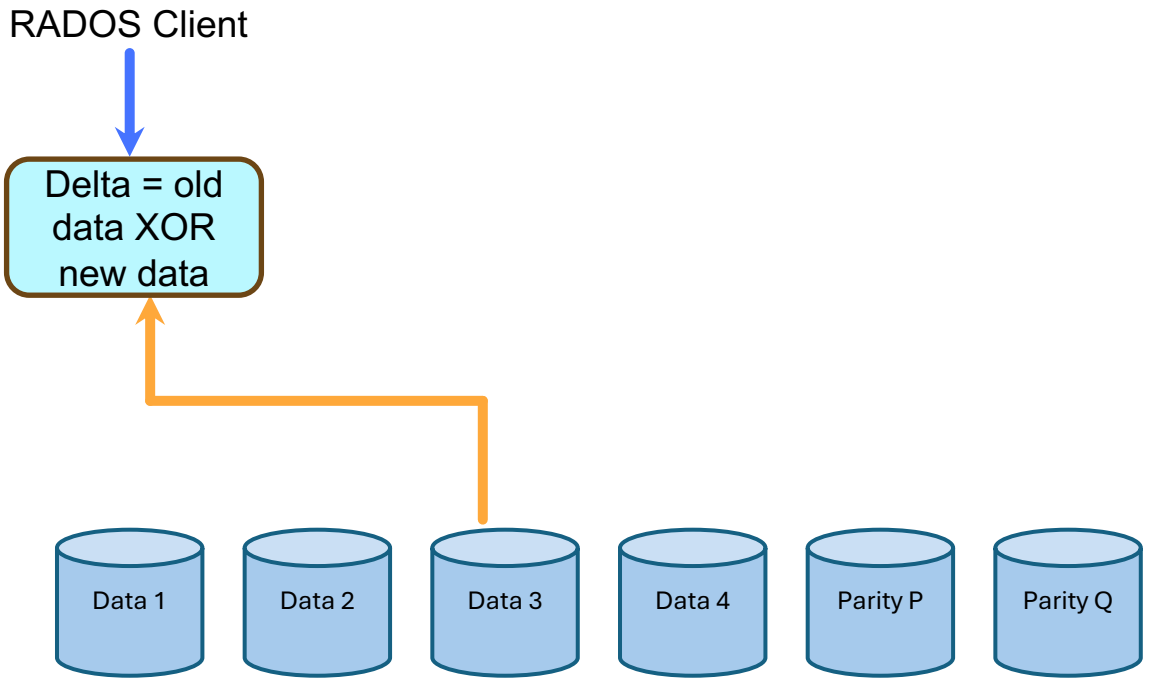
- With 4+2 there is no benefit
- With 3+2 it would increase the amount of I/O

Optimize Network Bandwidth



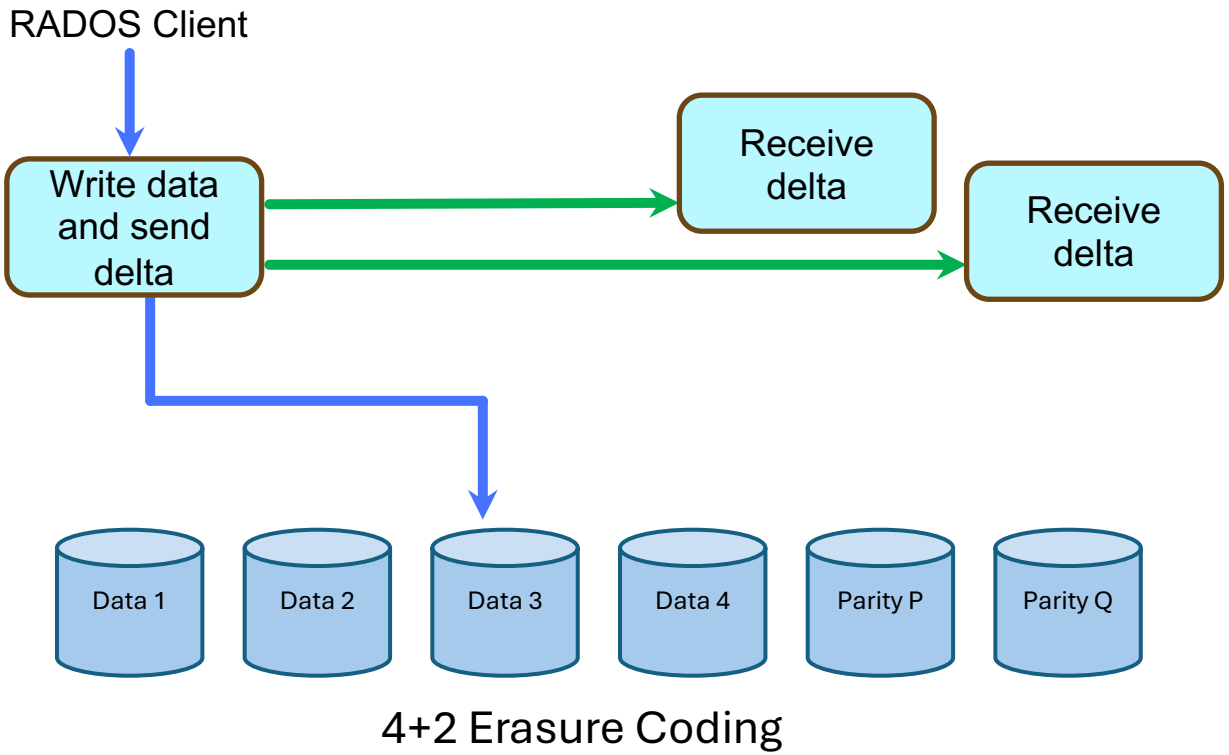
4+2 Erasure Coding

Optimize Network Bandwidth



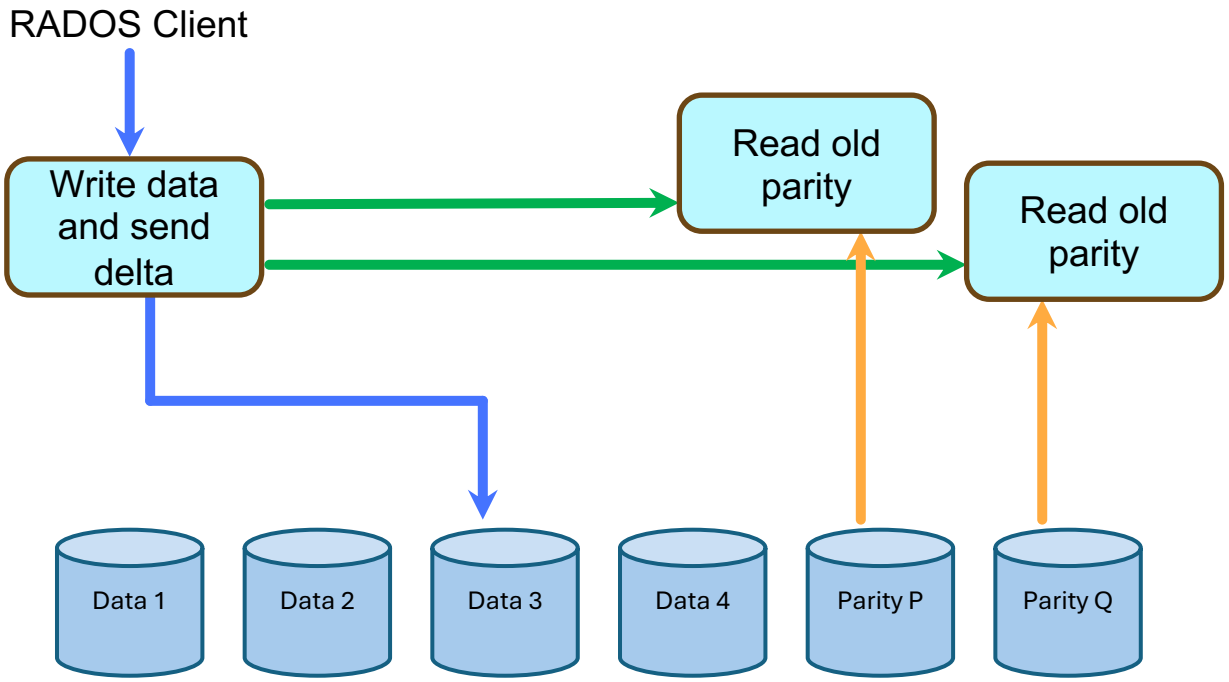
4+2 Erasure Coding

Optimize Network Bandwidth



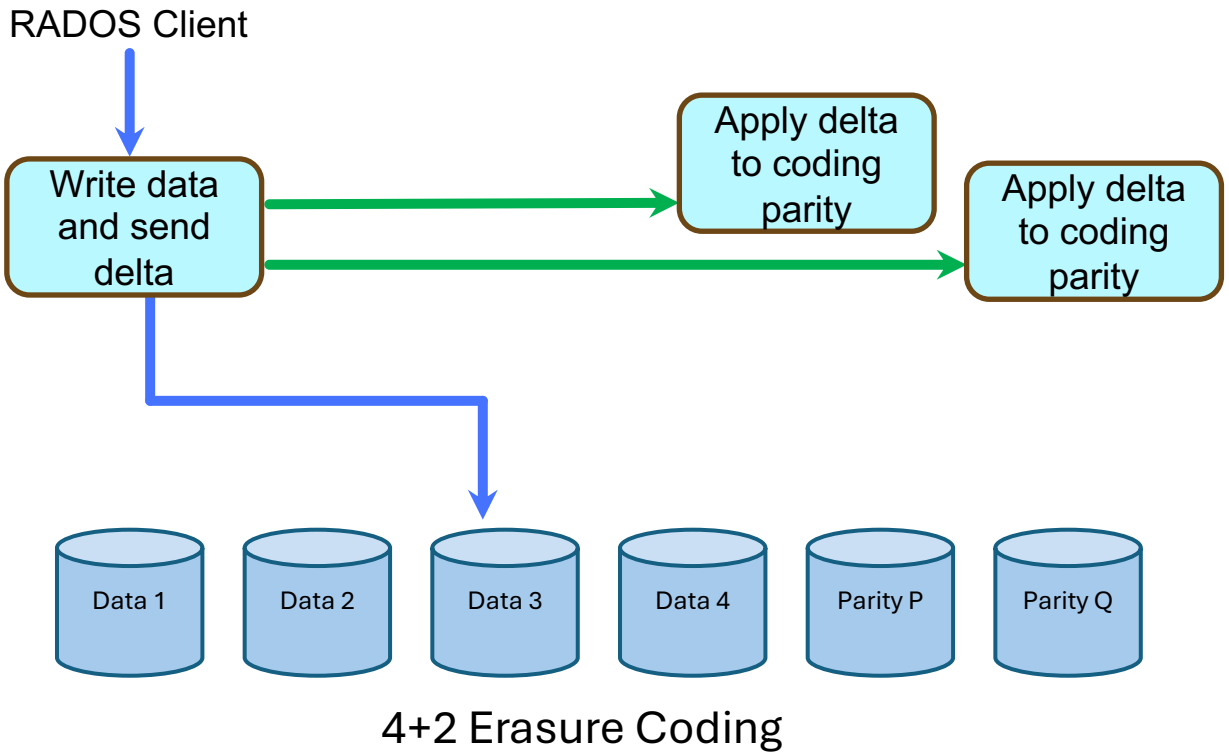
4+2 Erasure Coding

Optimize Network Bandwidth



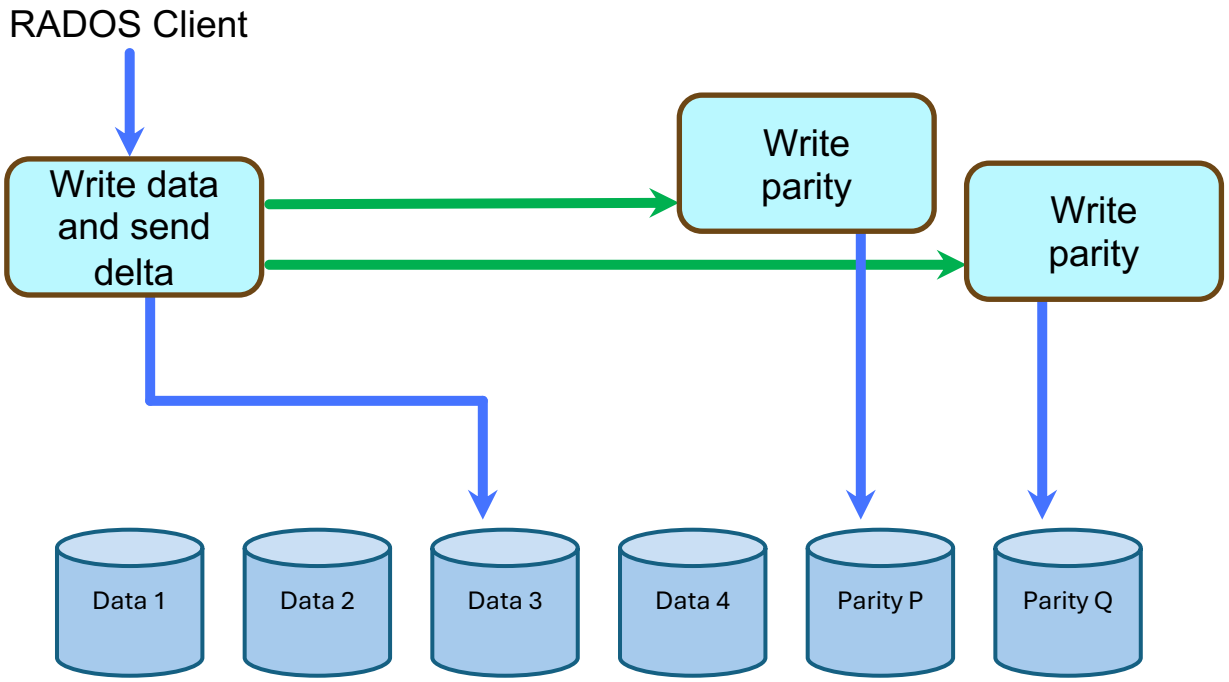
4+2 Erasure Coding

Optimize Network Bandwidth



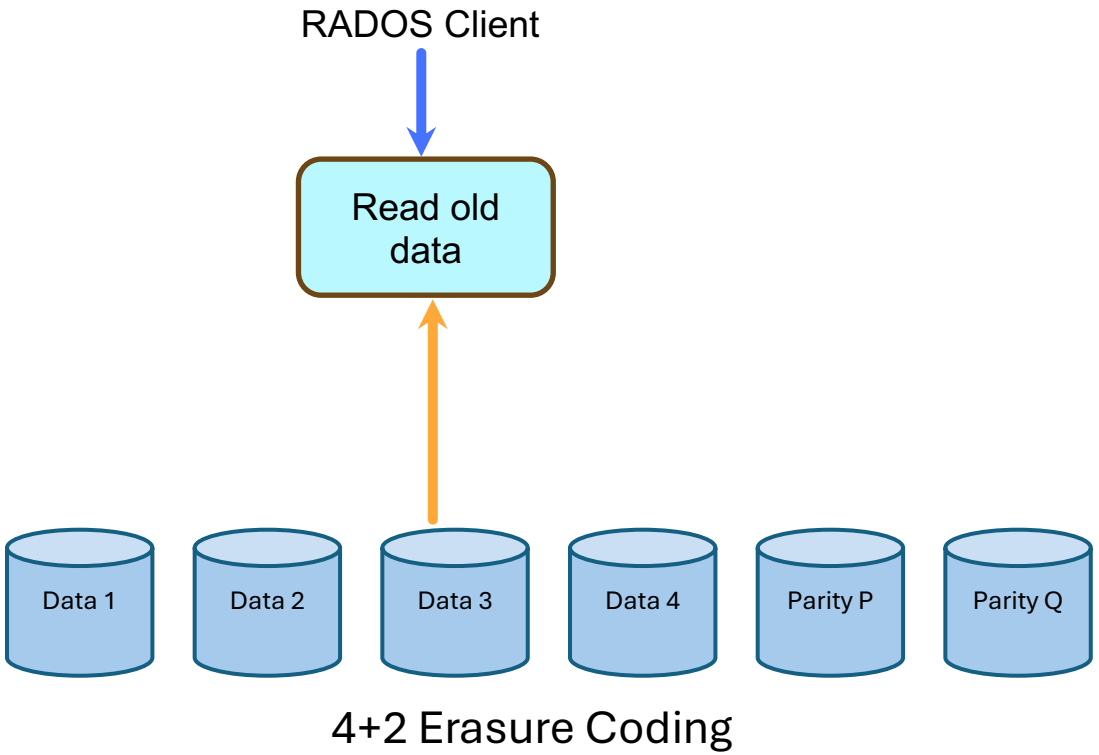
4+2 Erasure Coding

Optimize Network Bandwidth



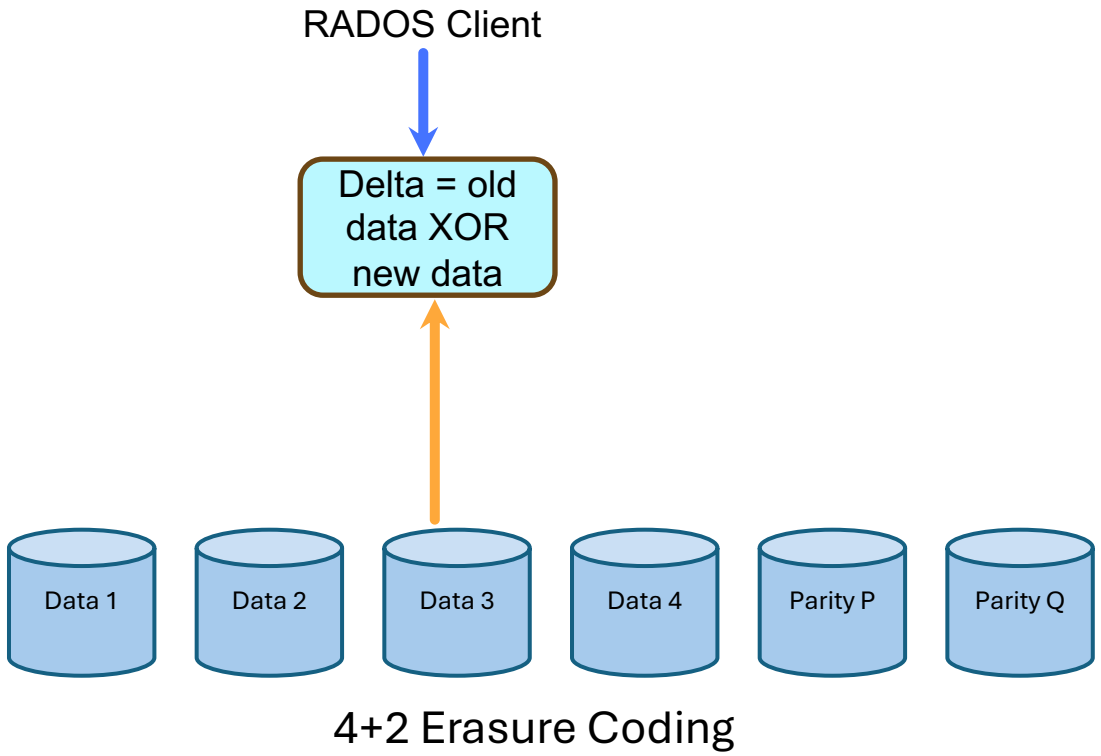
4+2 Erasure Coding

Direct Write



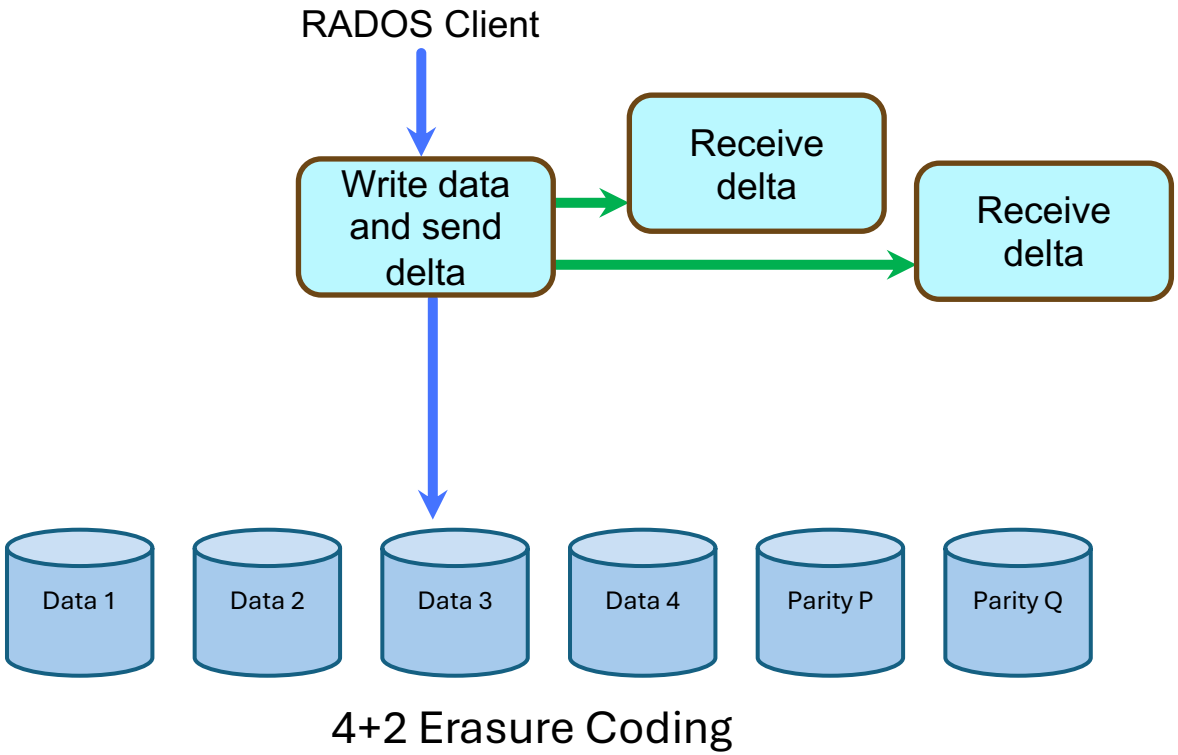
4+2 Erasure Coding

Direct Write



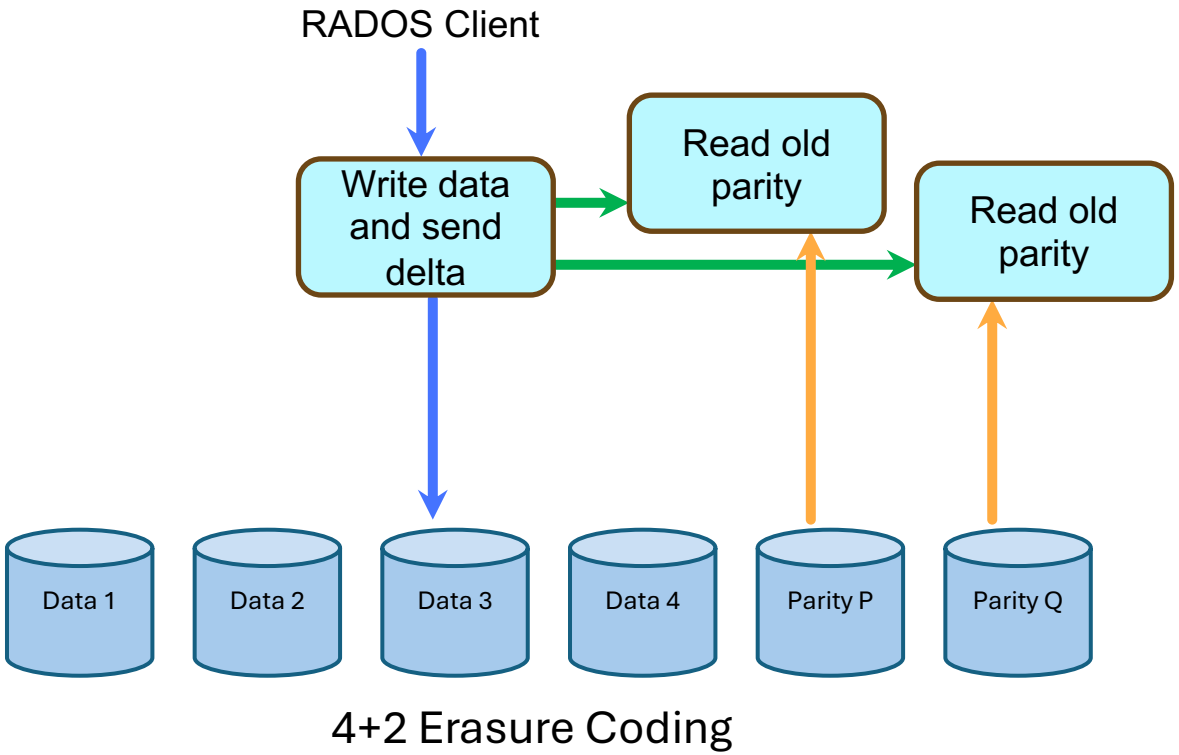
4+2 Erasure Coding

Direct Write



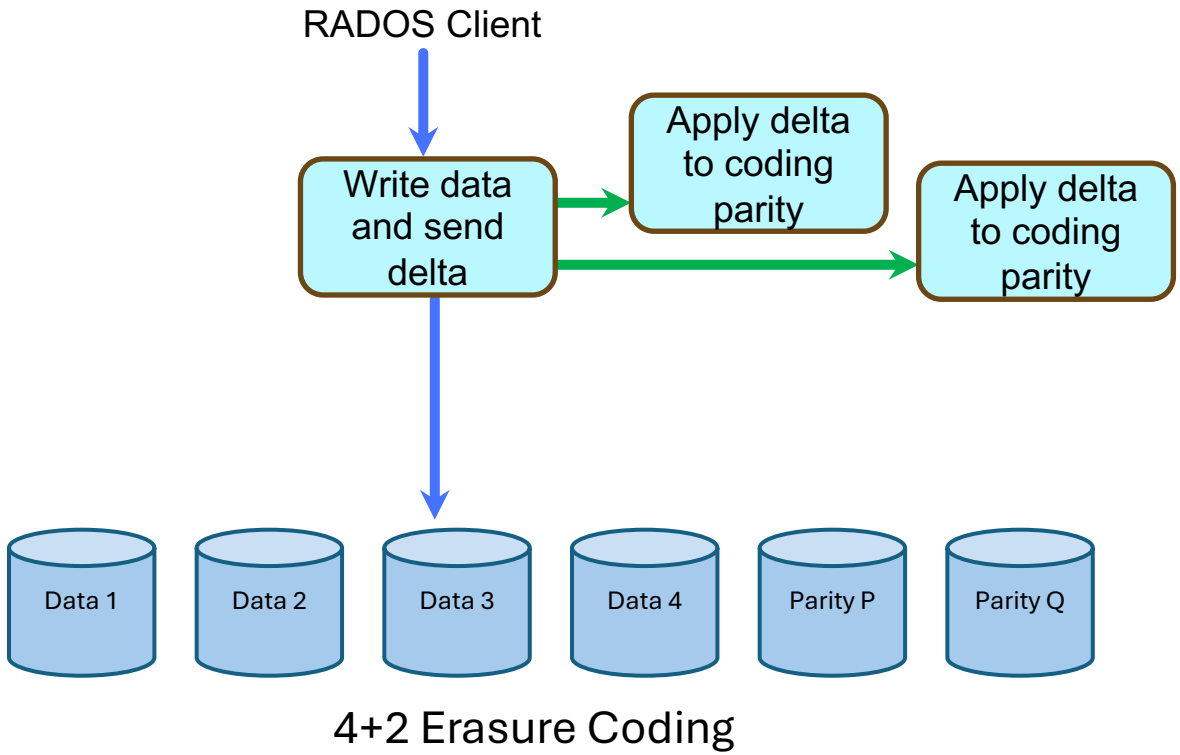
4+2 Erasure Coding

Direct Write



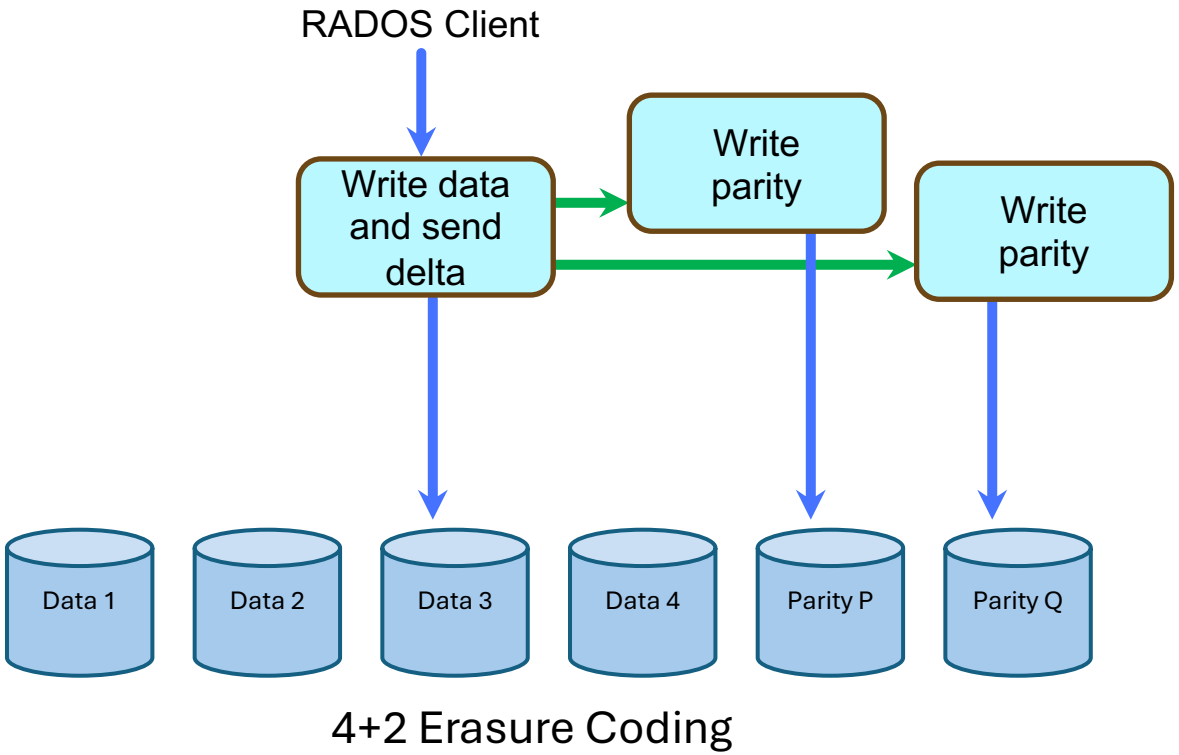
4+2 Erasure Coding

Direct Write

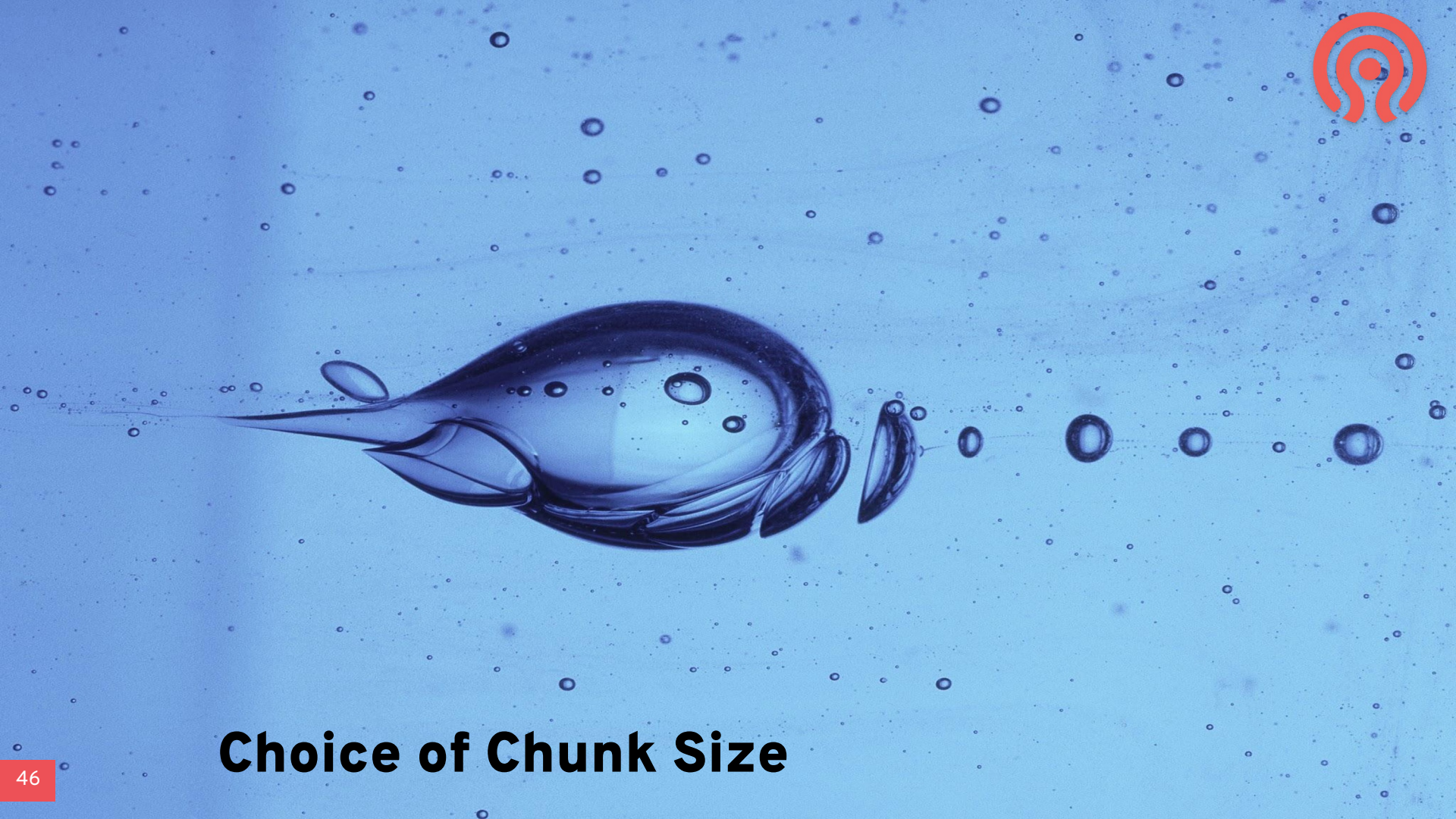


4+2 Erasure Coding

Direct Write

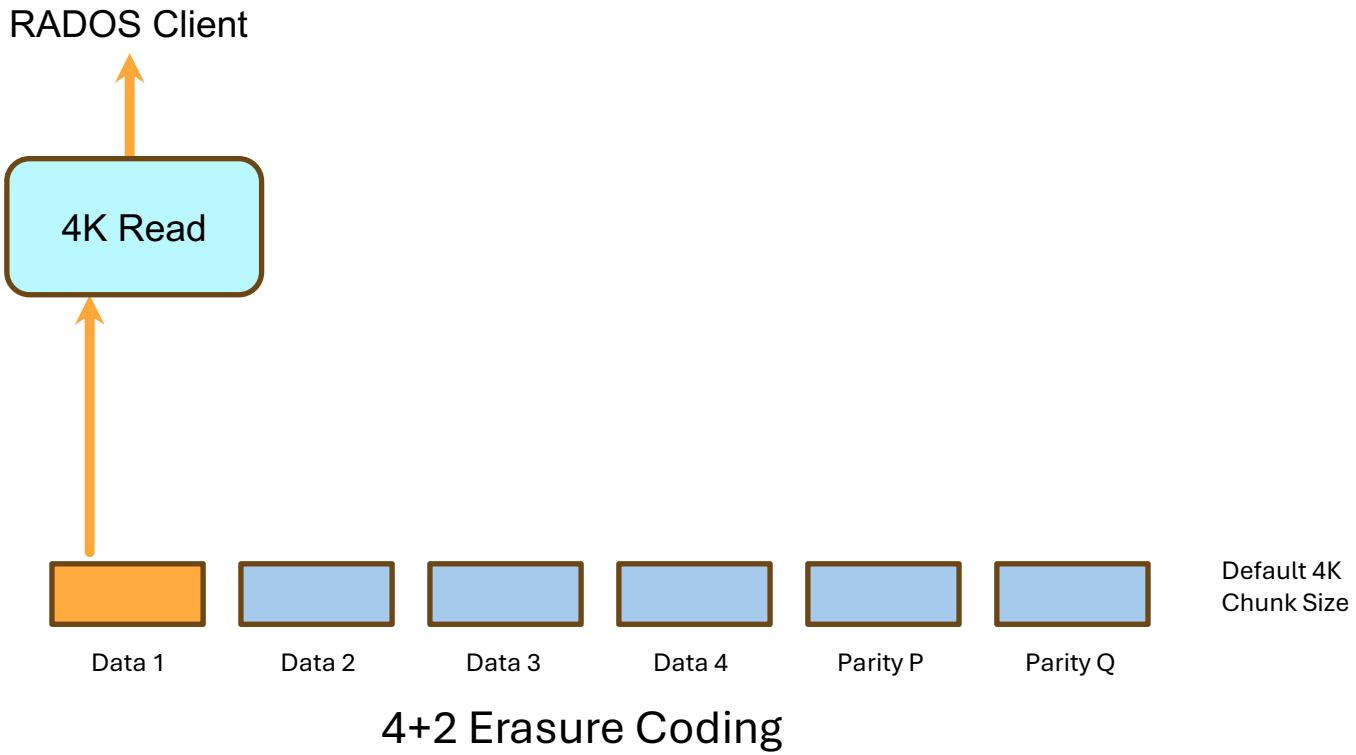


4+2 Erasure Coding

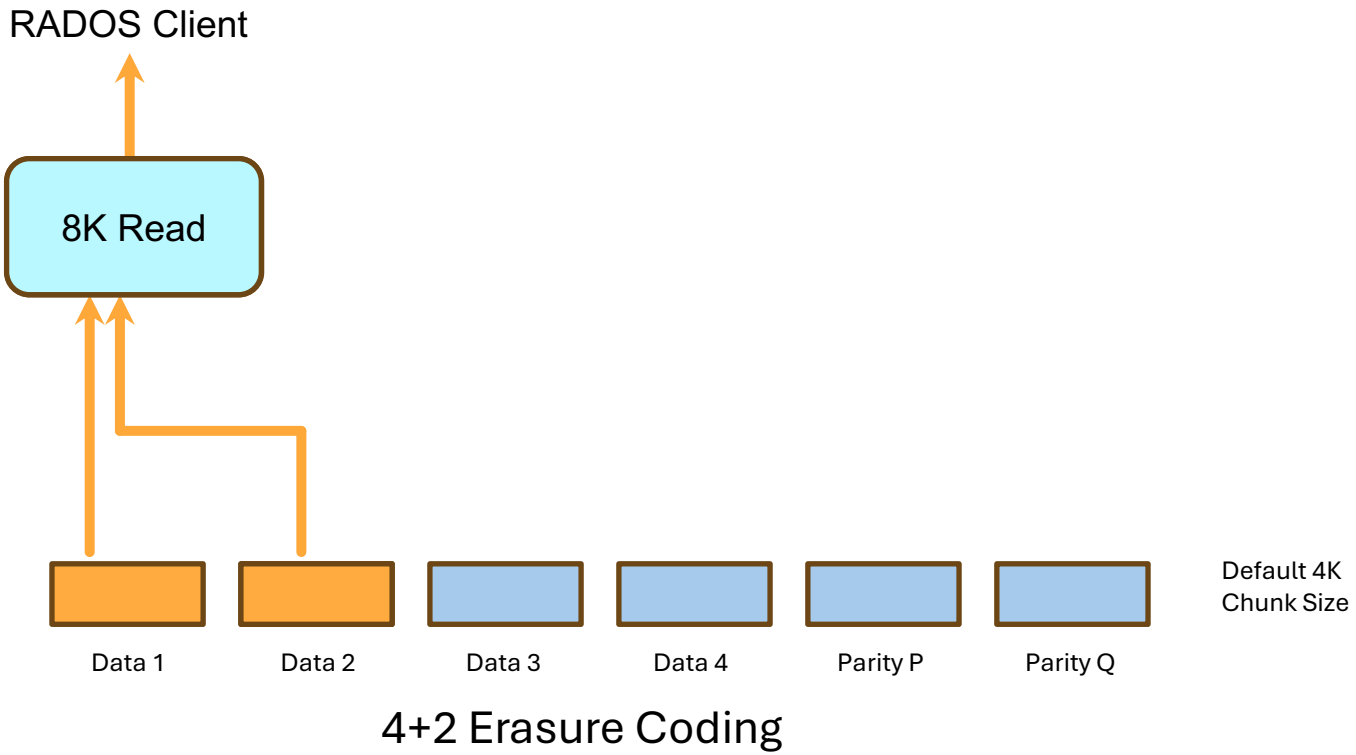


Choice of Chunk Size

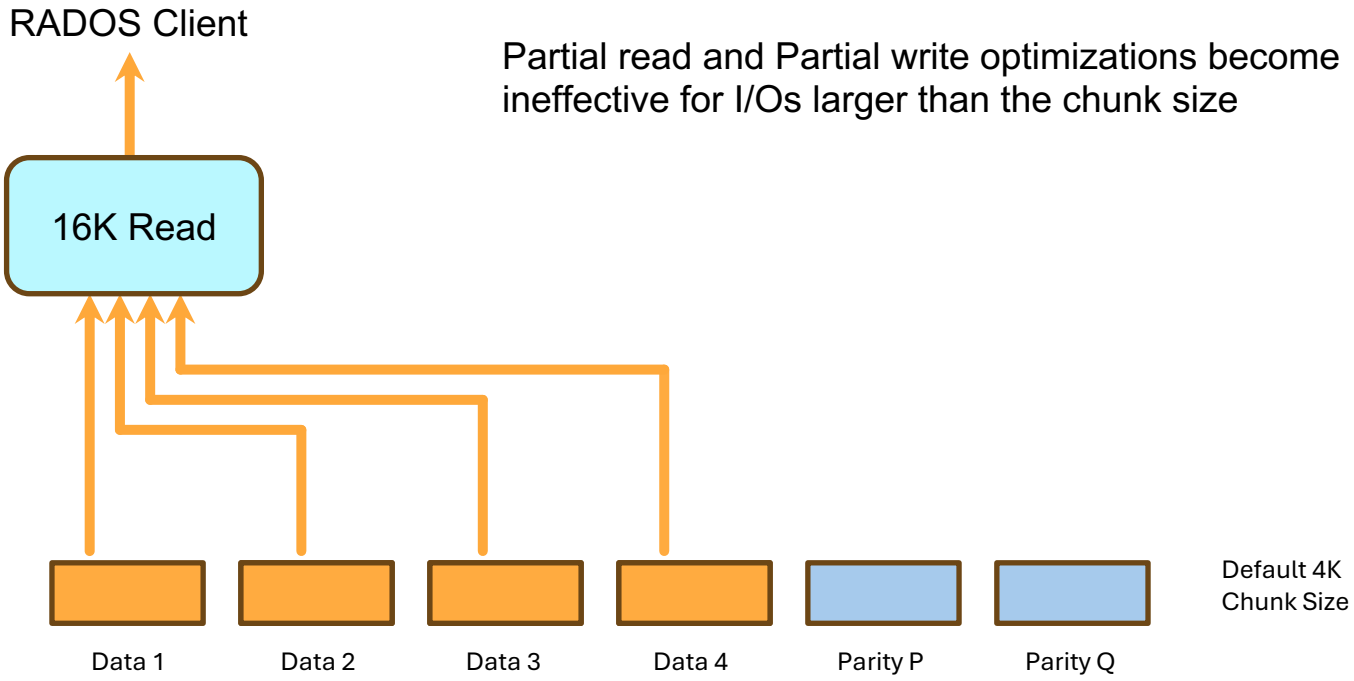
Default 4K Chunk Size



Default 4K Chunk Size



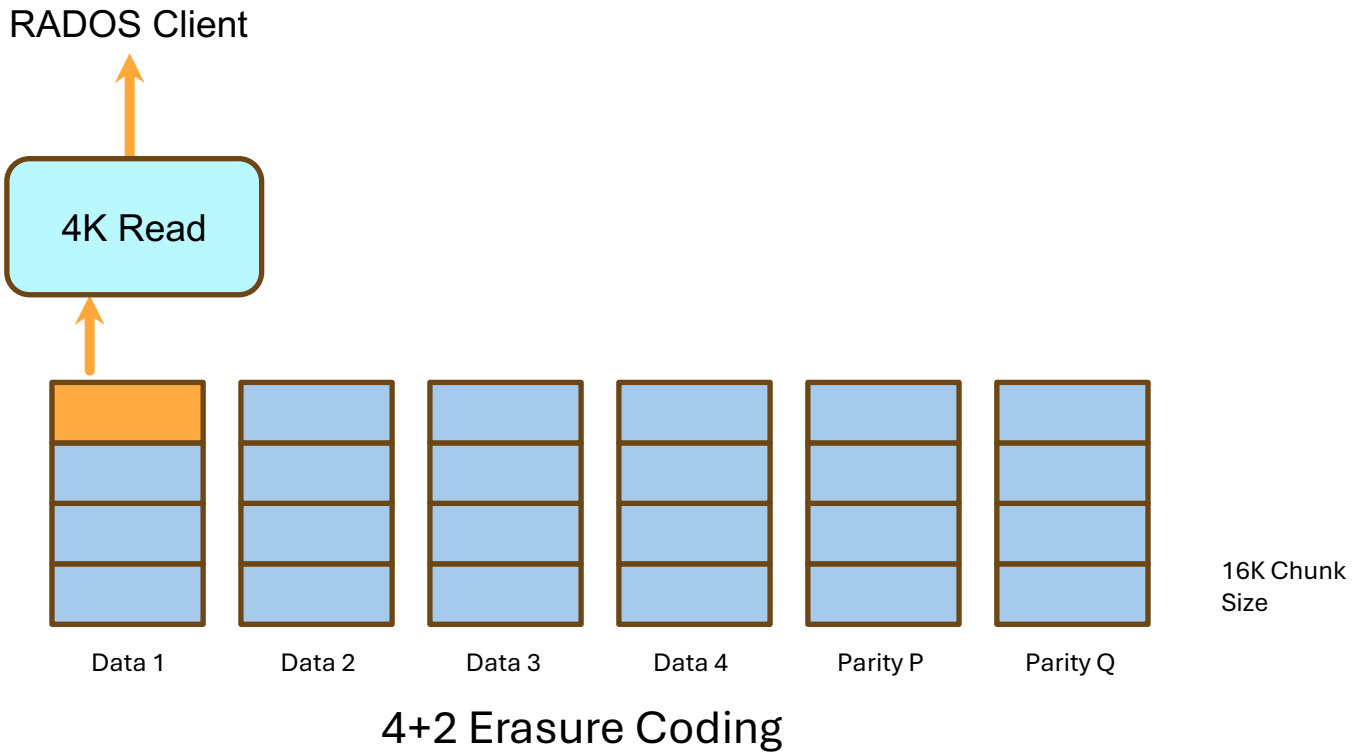
Default 4K Chunk Size



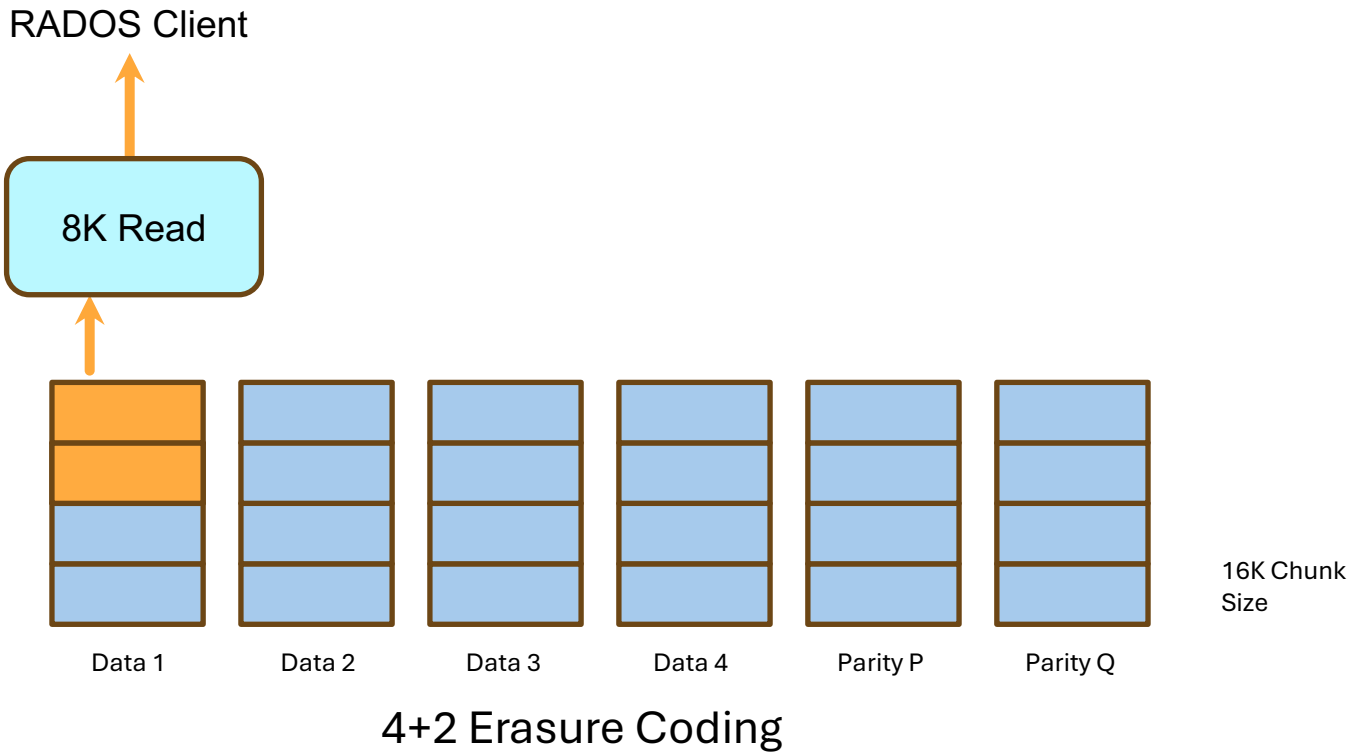
Partial read and Partial write optimizations become ineffective for I/Os larger than the chunk size

4+2 Erasure Coding

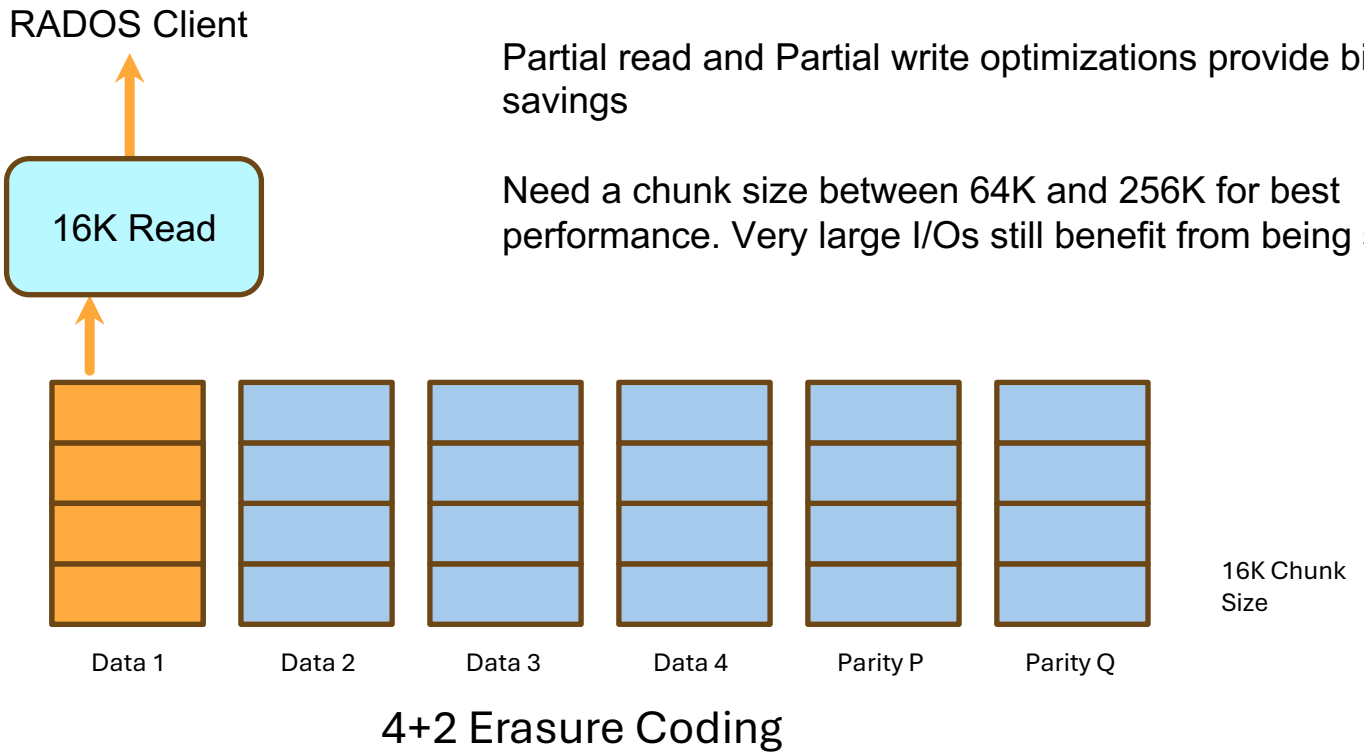
Larger Chunk Size



Larger Chunk Size



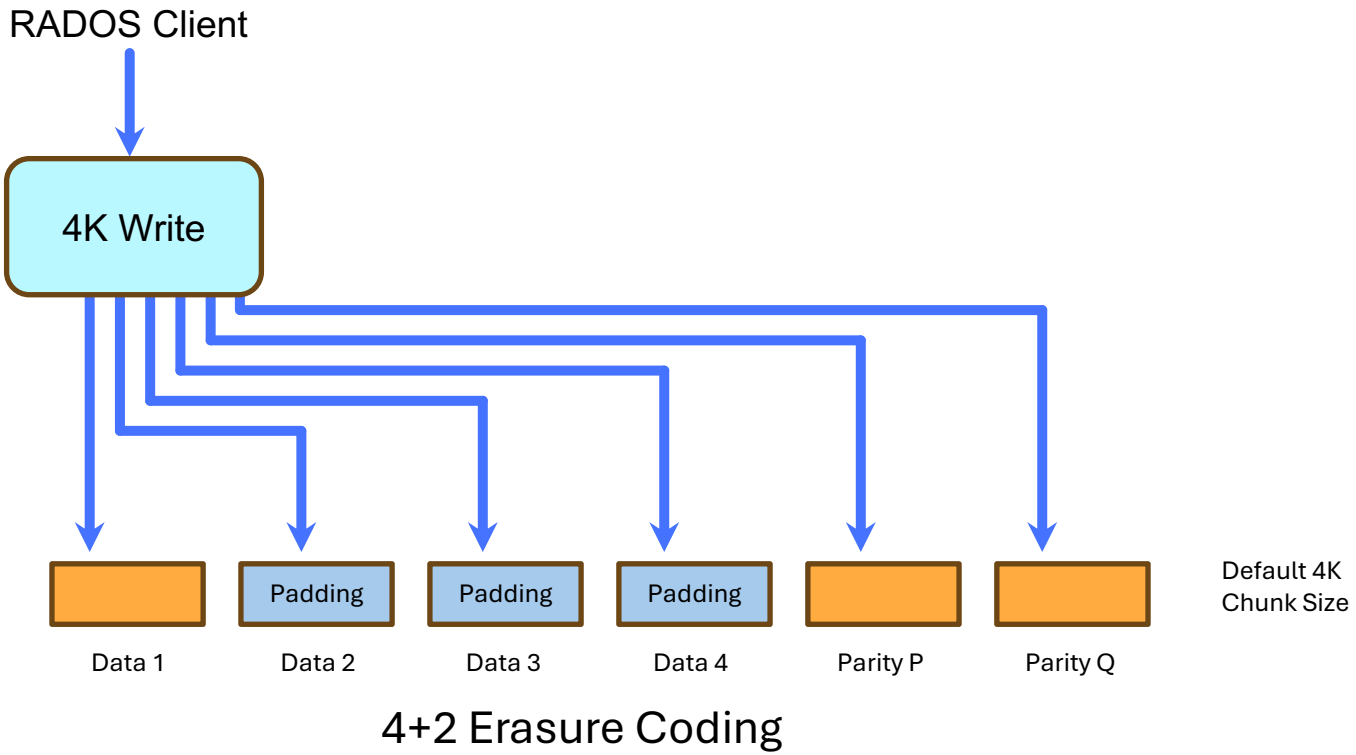
Larger Chunk Size



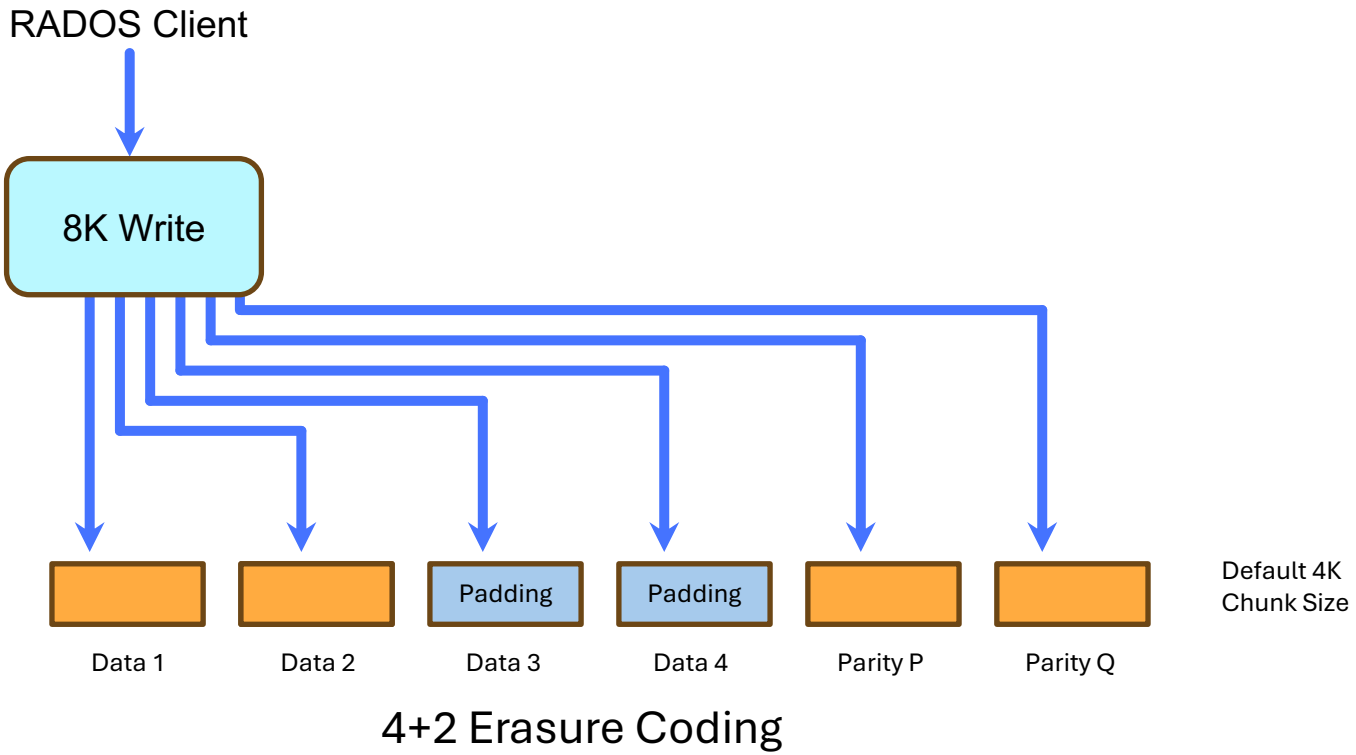
Partial read and Partial write optimizations provide bigger savings

Need a chunk size between 64K and 256K for best performance. Very large I/Os still benefit from being split

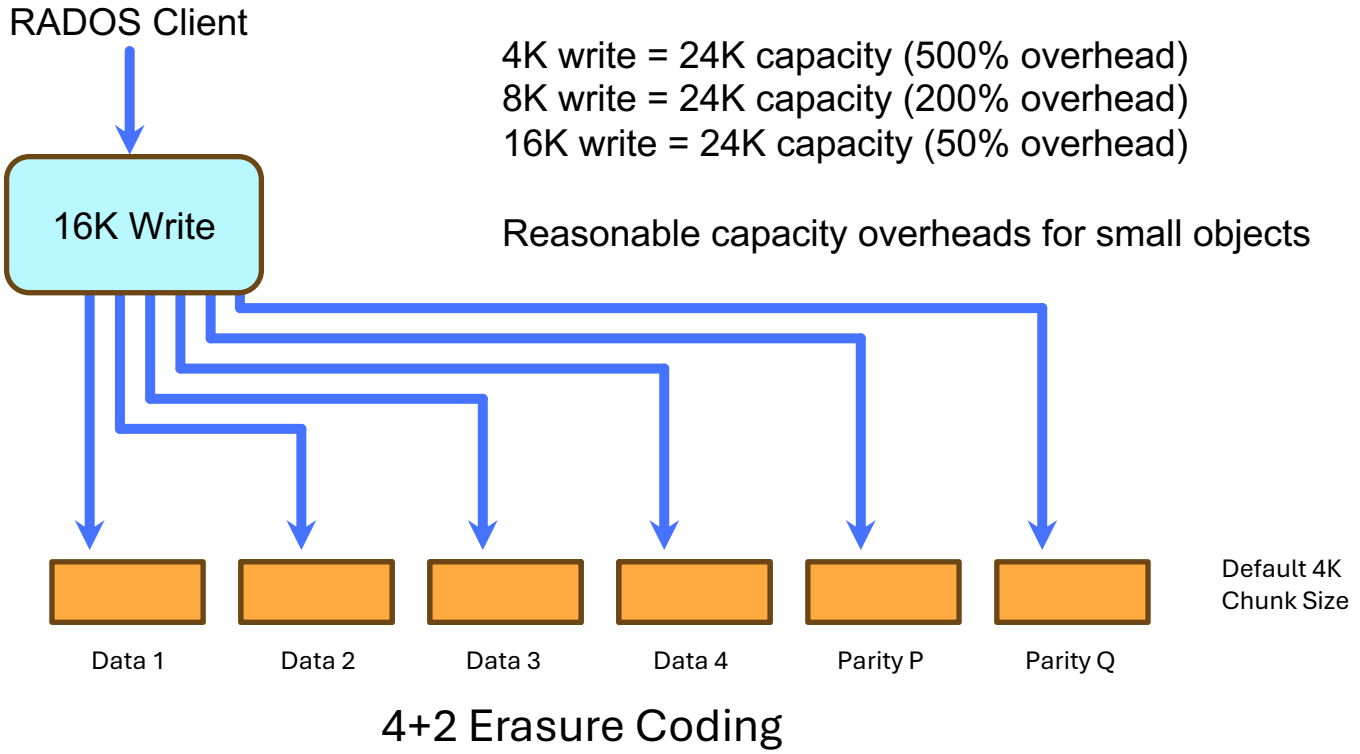
Default 4K Chunk Size and Small Objects



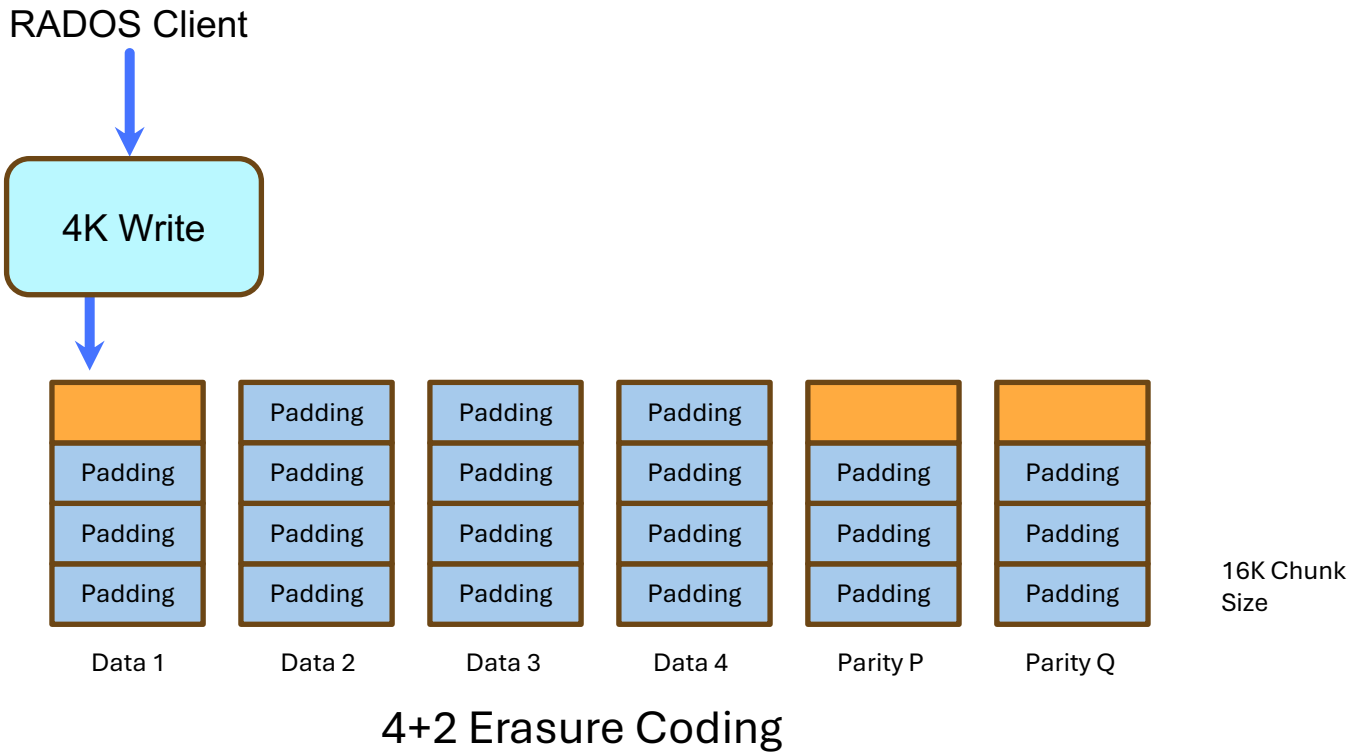
Default 4K Chunk Size and Small Objects



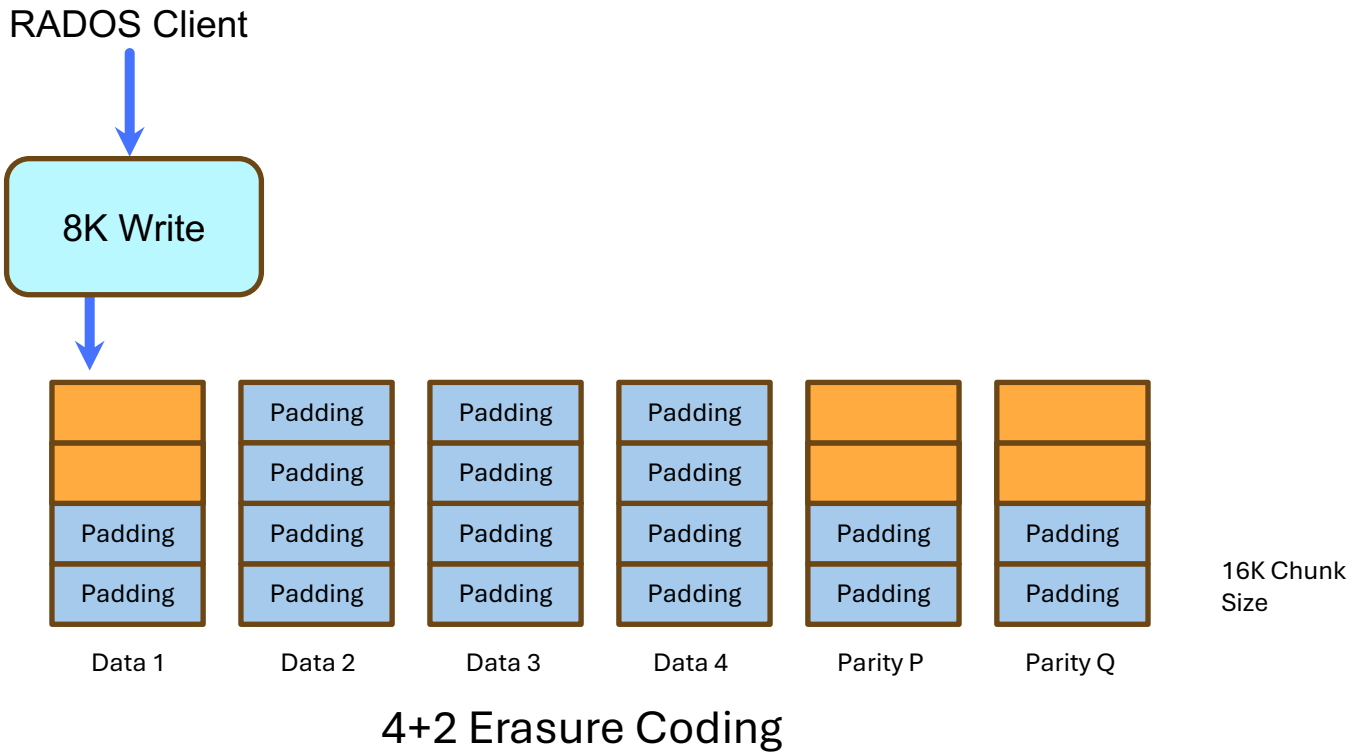
Default 4K Chunk Size and Small Objects



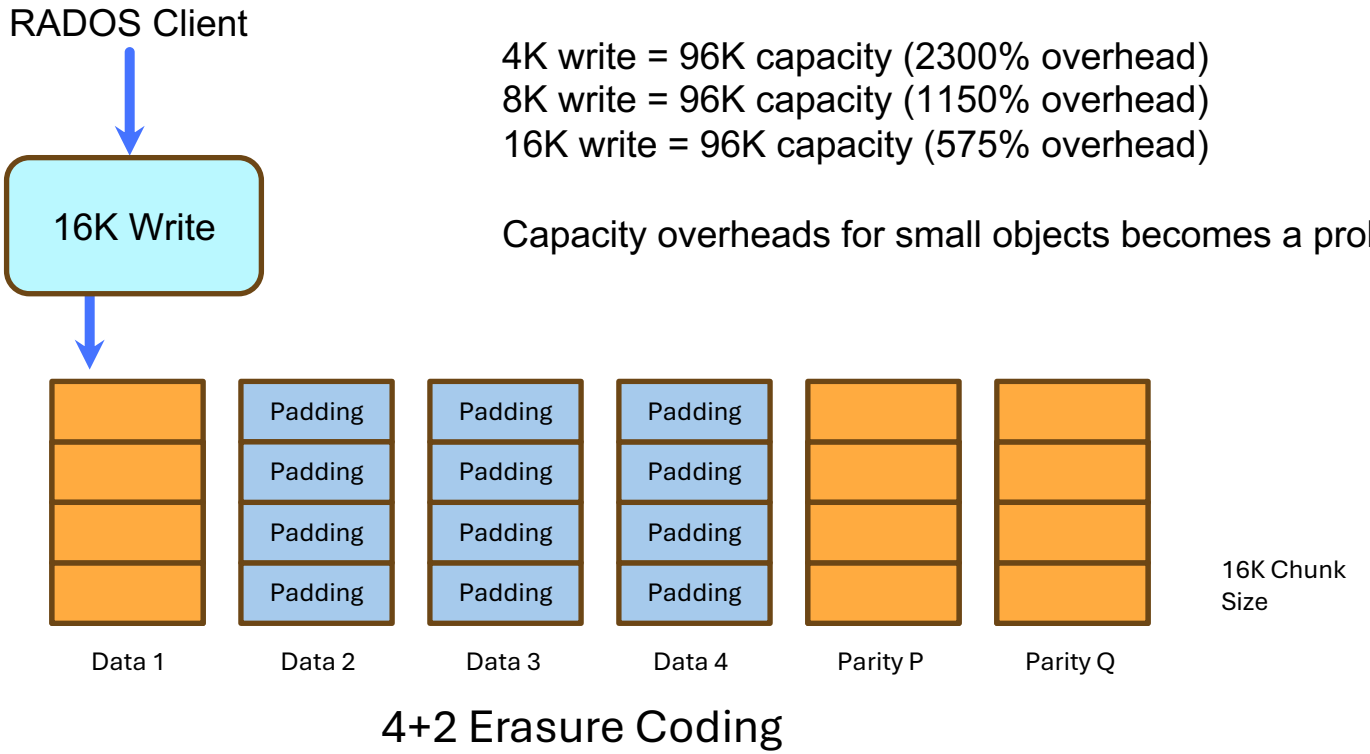
Larger Chunk Size and Small Objects



Larger Chunk Size and Small Objects



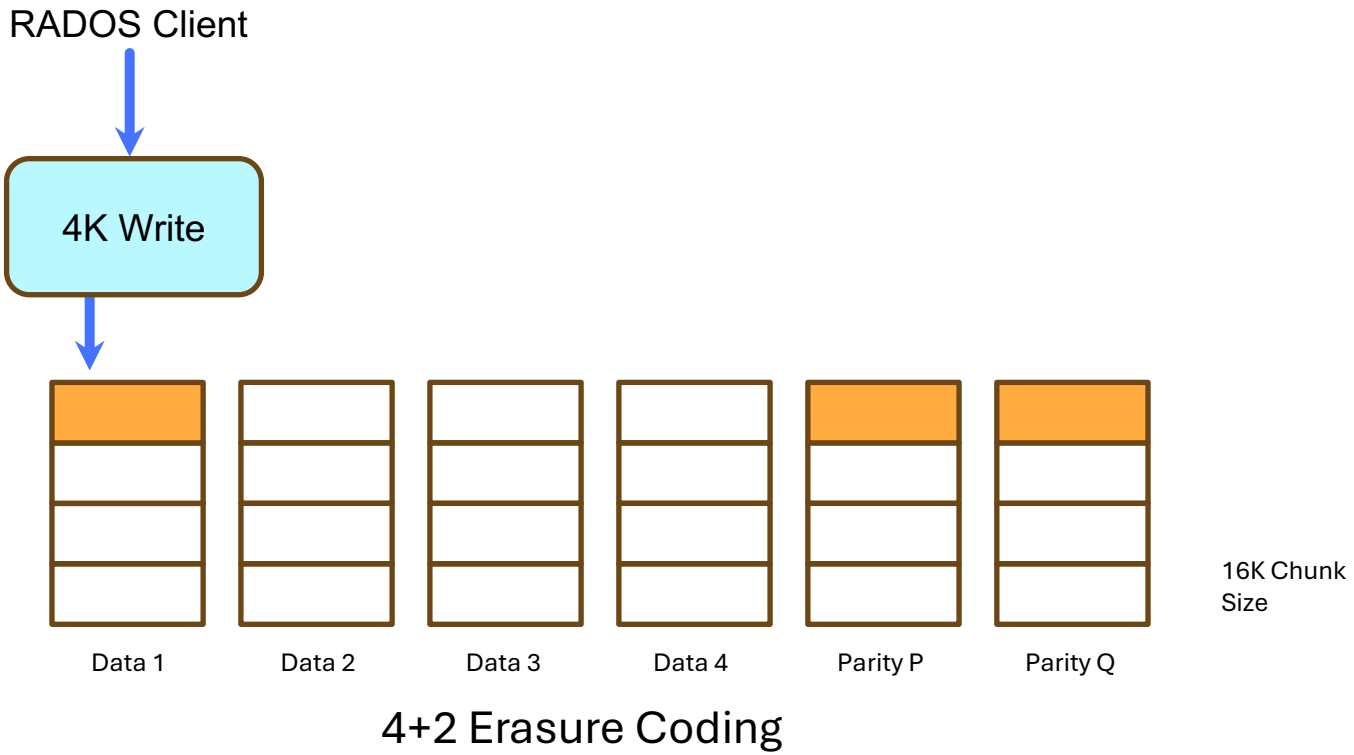
Larger Chunk Size and Small Objects



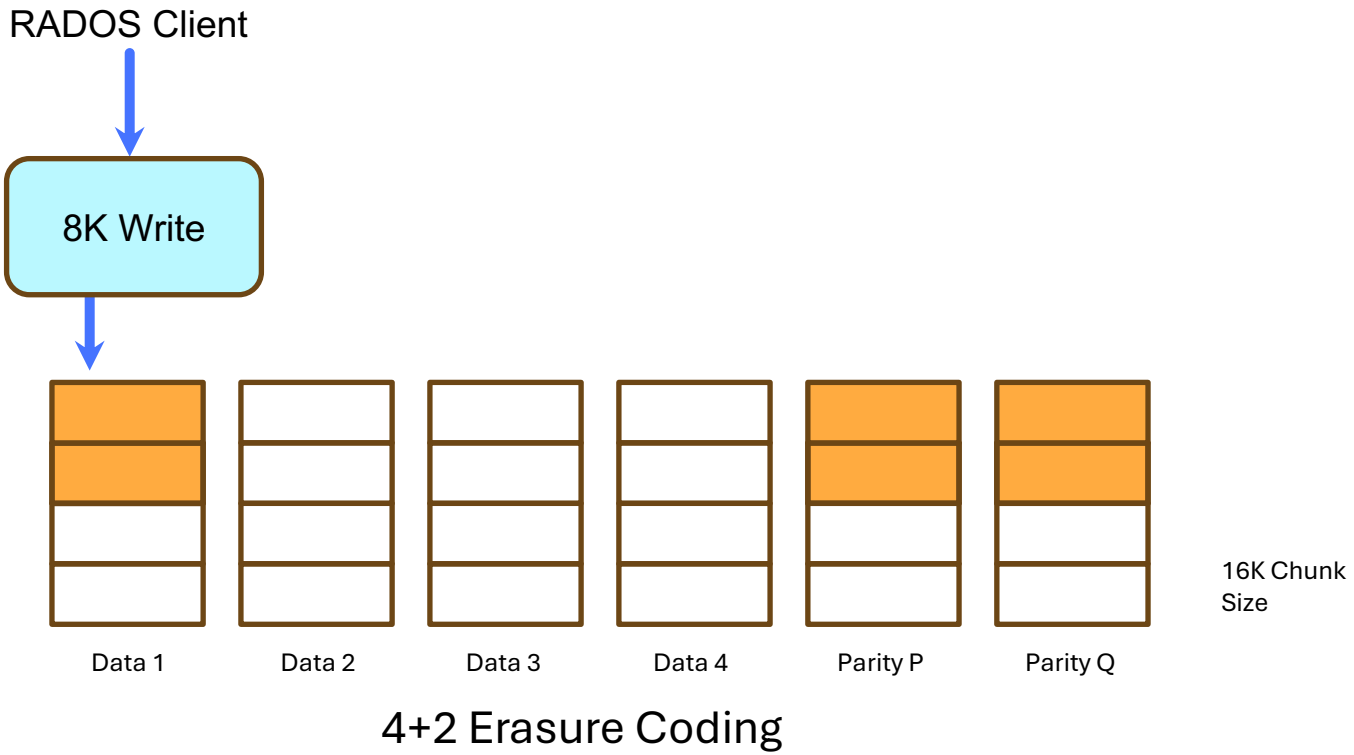
4K write = 96K capacity (2300% overhead)
8K write = 96K capacity (1150% overhead)
16K write = 96K capacity (575% overhead)

Capacity overheads for small objects becomes a problem

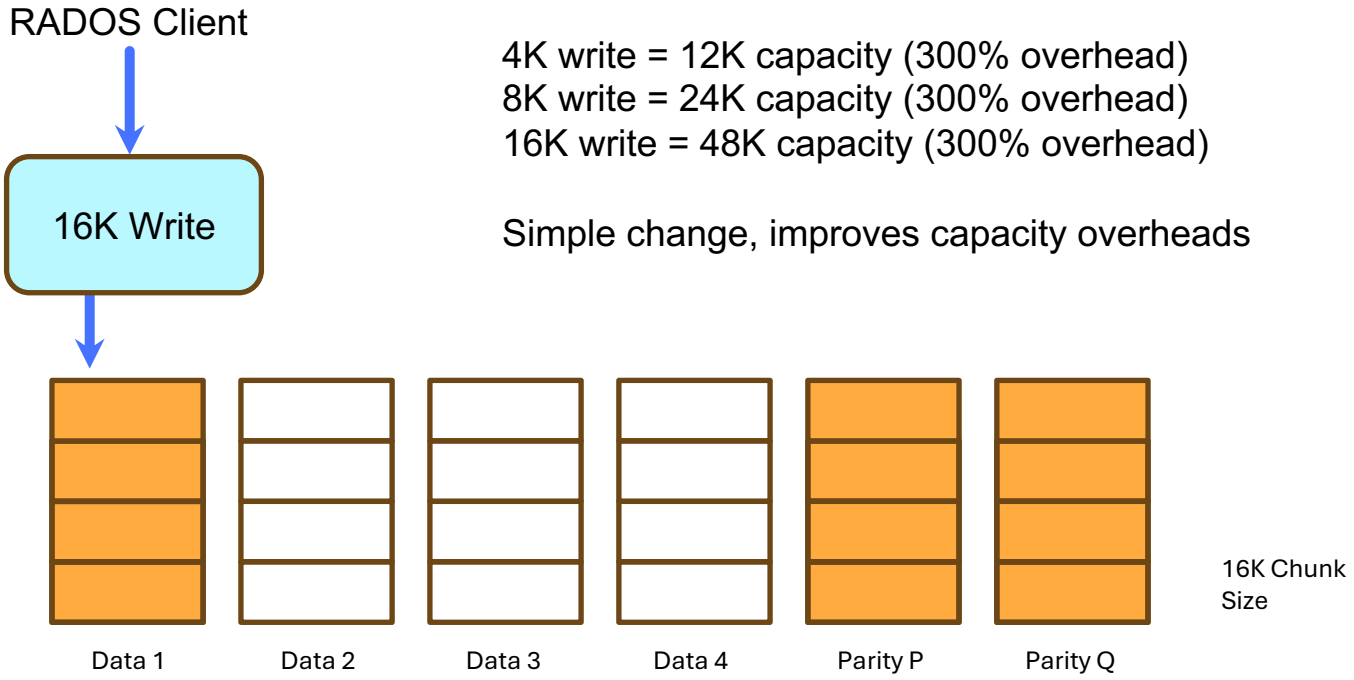
Larger Chunk Size and No Padding



Larger Chunk Size and No Padding



Larger Chunk Size and No Padding



4K write = 12K capacity (300% overhead)
8K write = 24K capacity (300% overhead)
16K write = 48K capacity (300% overhead)

Simple change, improves capacity overheads

4+2 Erasure Coding

Variable Chunk Size

Idea: Automatically select a chunk size based on the object size

- Small sized objects use a 4K chunk size
- Large sized objects (e.g. >=1MB) use a 256K chunk size
- Convert the chunk size by reading and re-writing the object when it expands above 1MB or is truncated to less than 1MB
- Use the existing object size hint to select the chunk size to avoid creating a small object and then almost immediately having to convert it as data is written to a new object


Any Questions?