A Logically Disaggregated Cache for Replicated Storage Systems

Kiran Hombal

Henry Zhu

Shreesha G. Bhat

Neil Kaushikkar

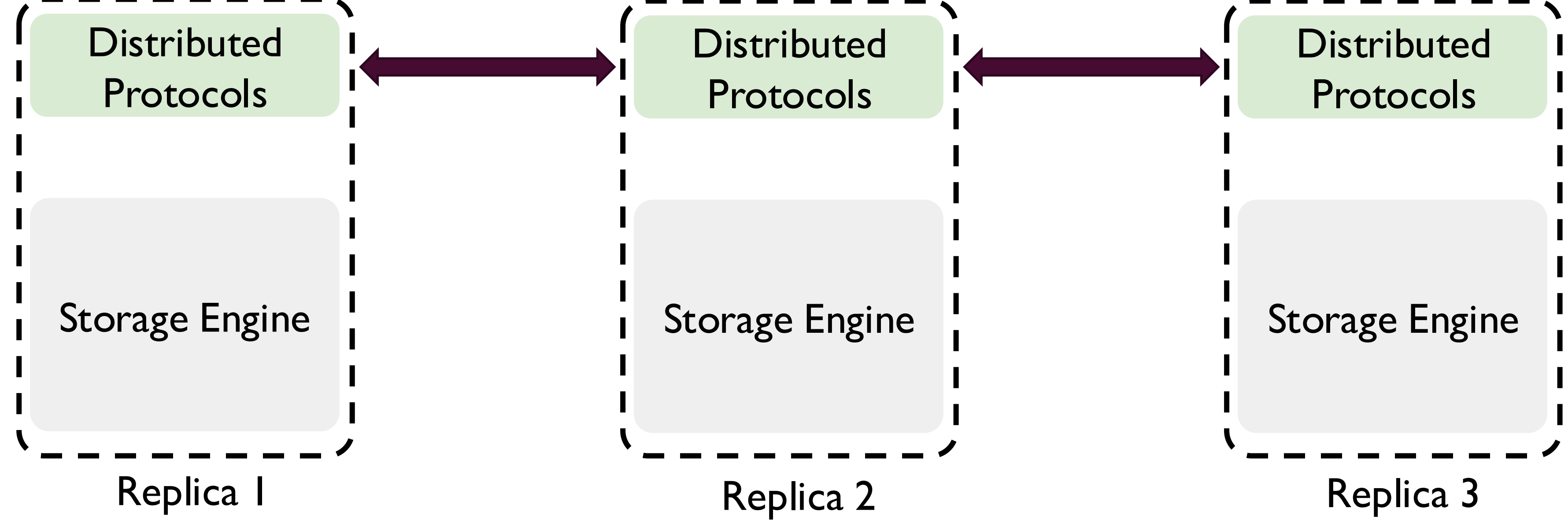
Ram Alagappan

Aishwarya Ganesan

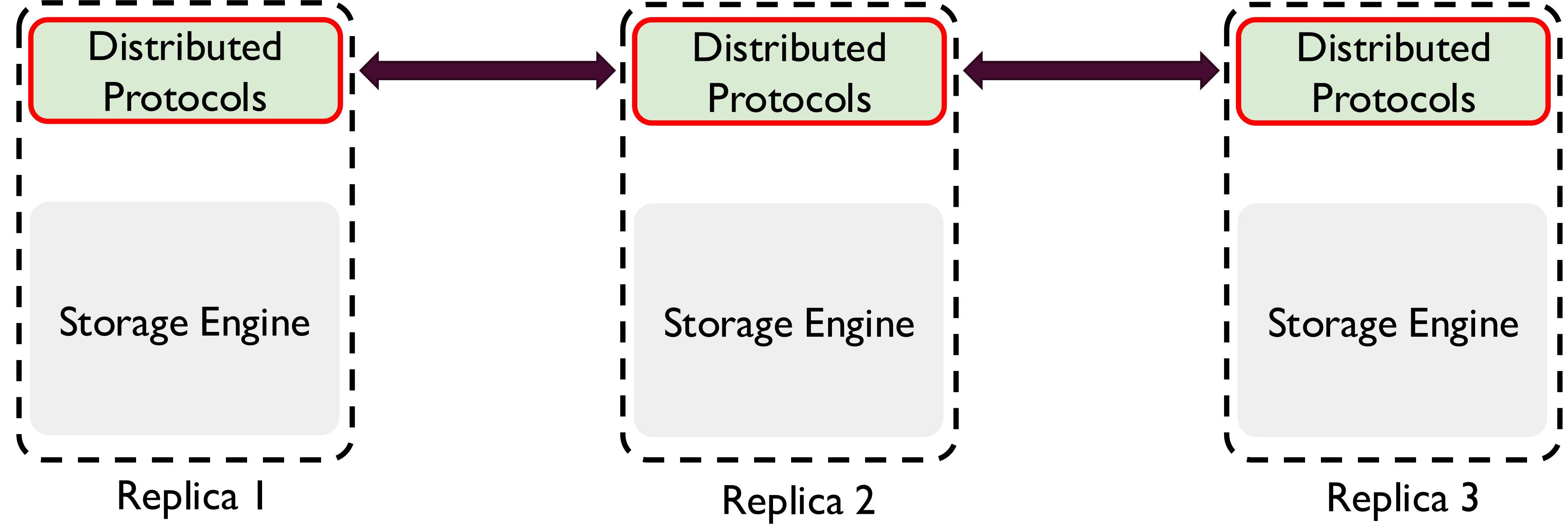
University of Illinois Urbana-Champaign

Replicated Storage Are Important

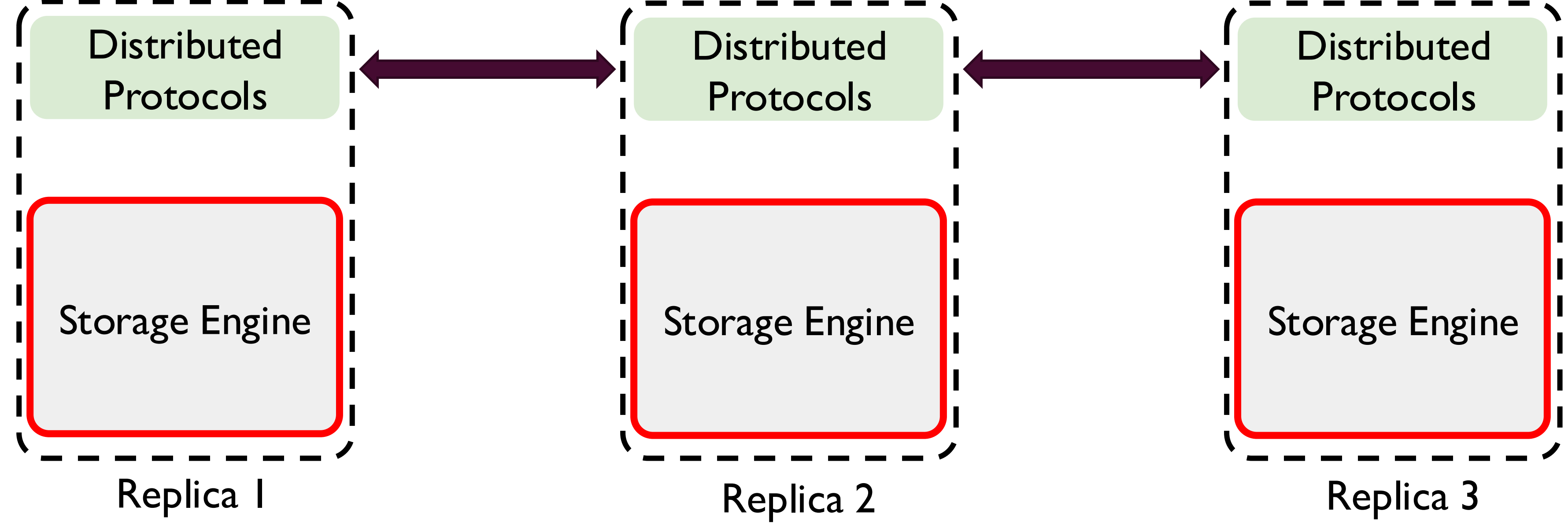
Replicated Storage



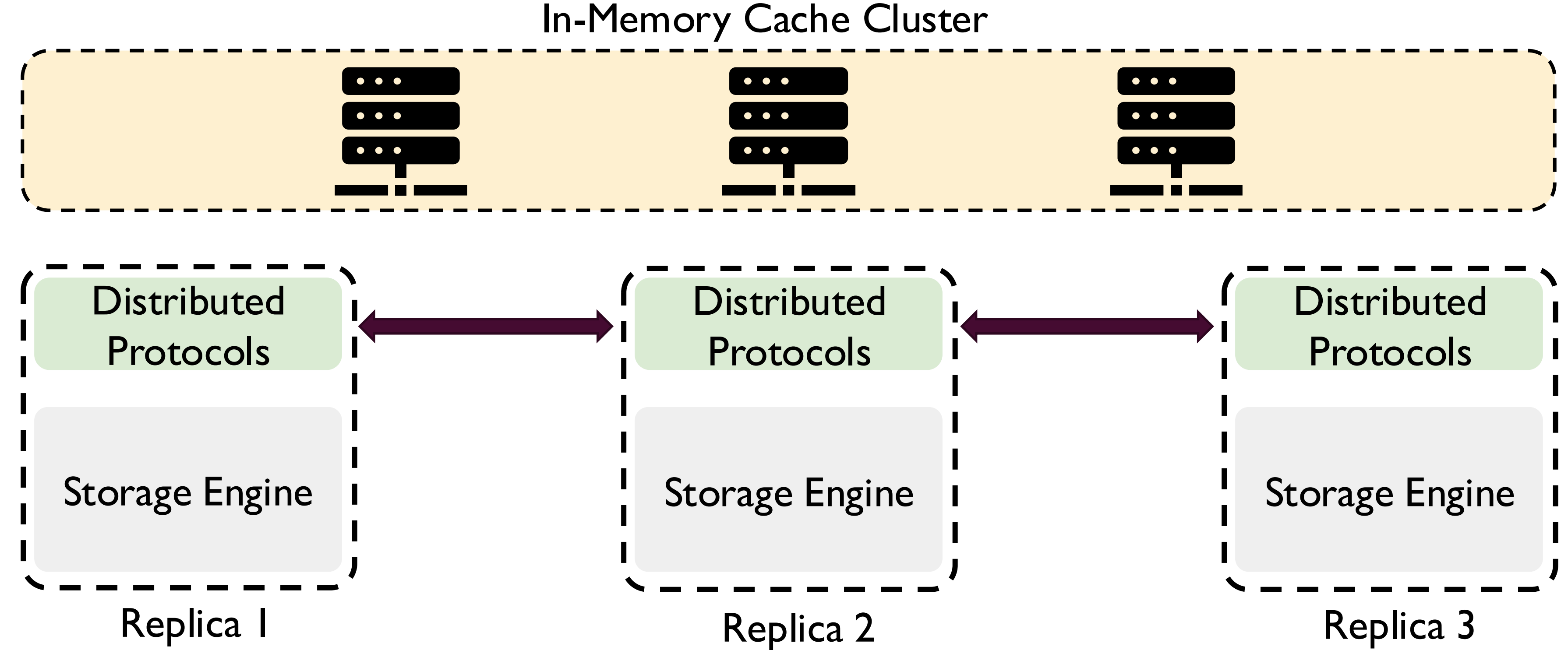
Replicated Storage



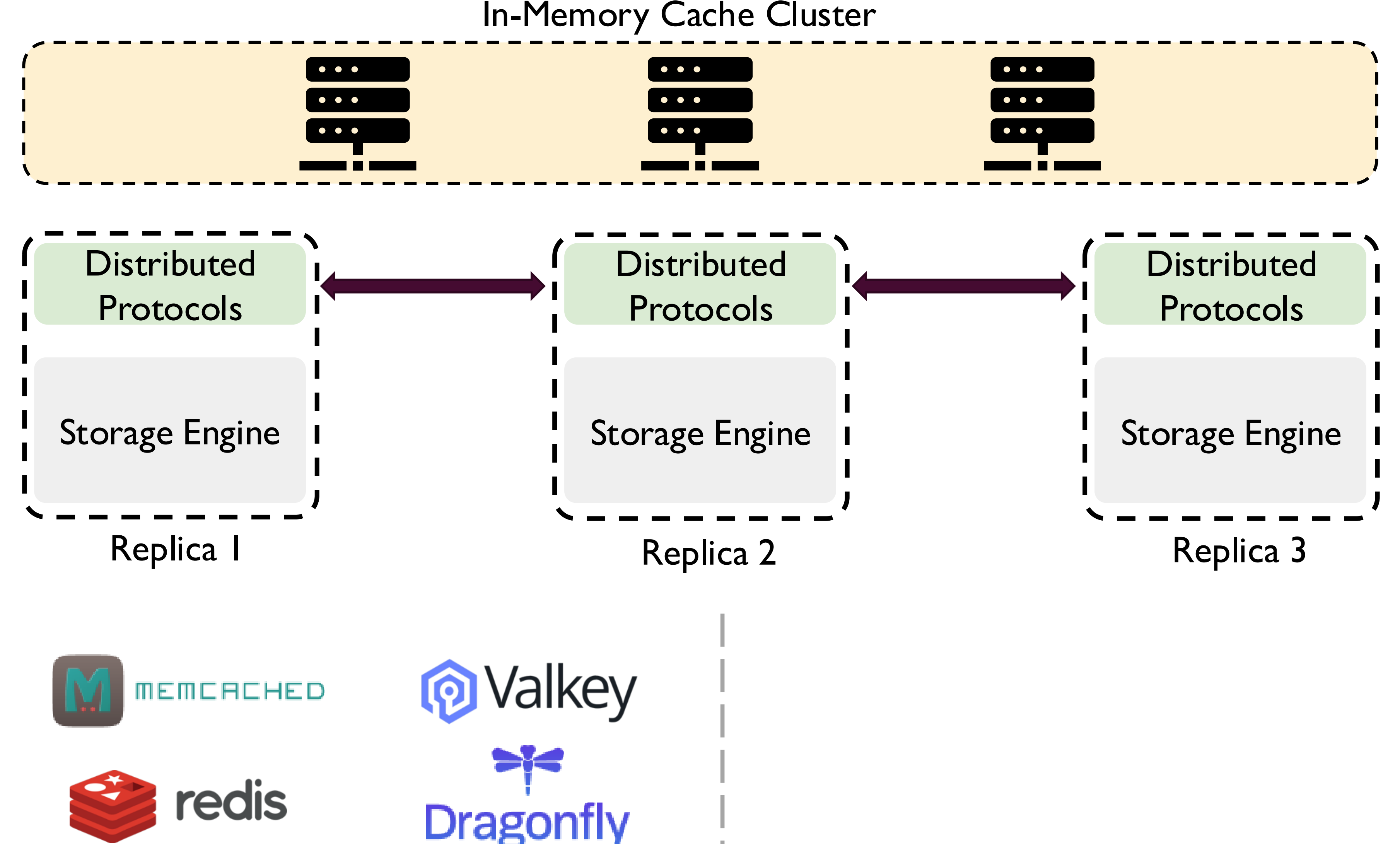
Replicated Storage



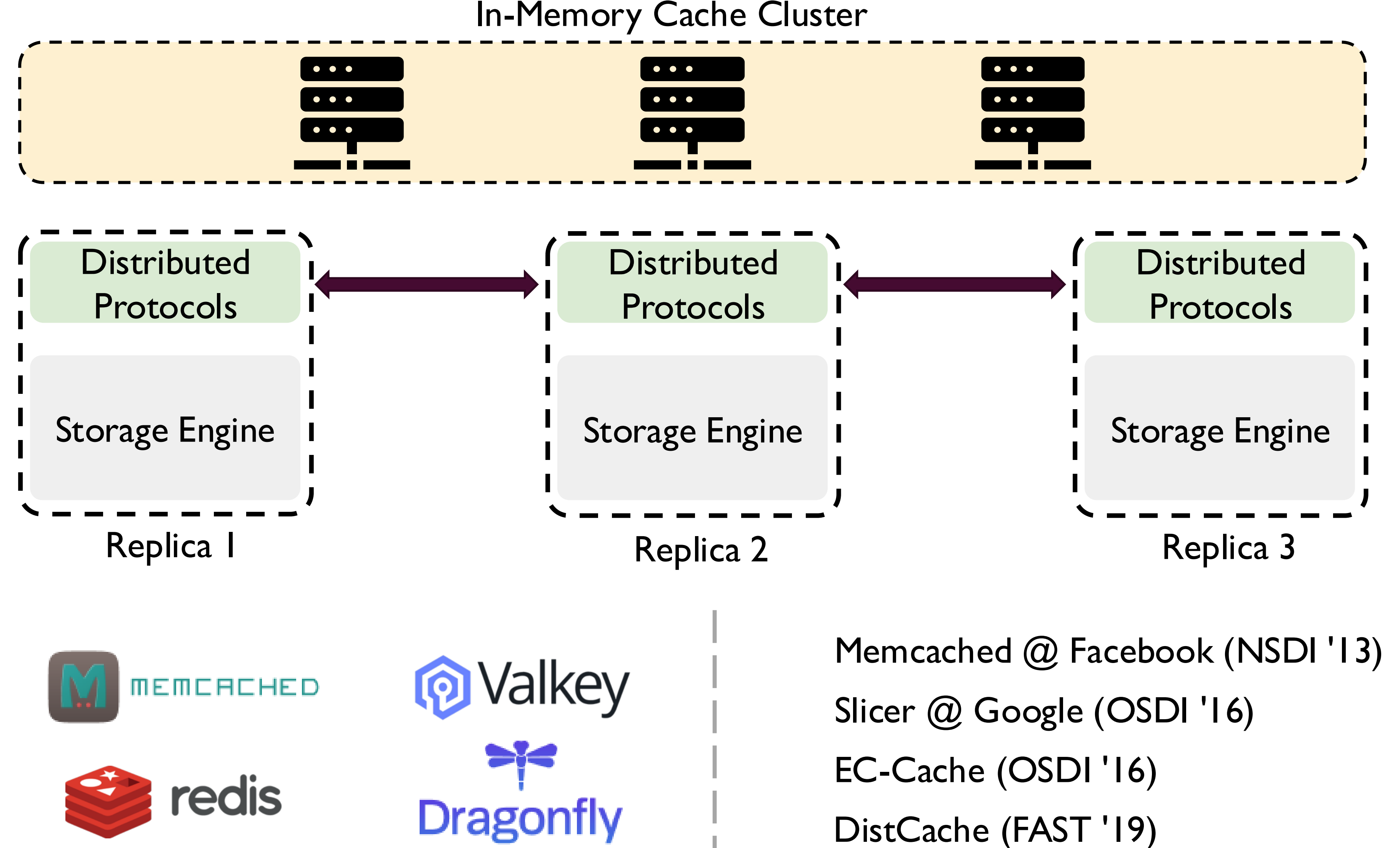
Optimizing Caches In These Storage Systems Is Important



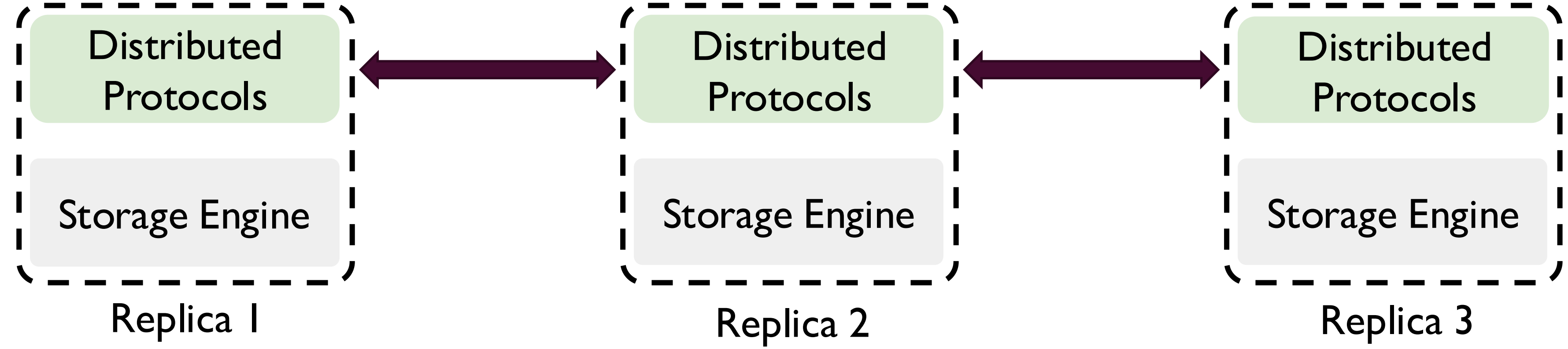
Optimizing Caches In These Storage Systems Is Important



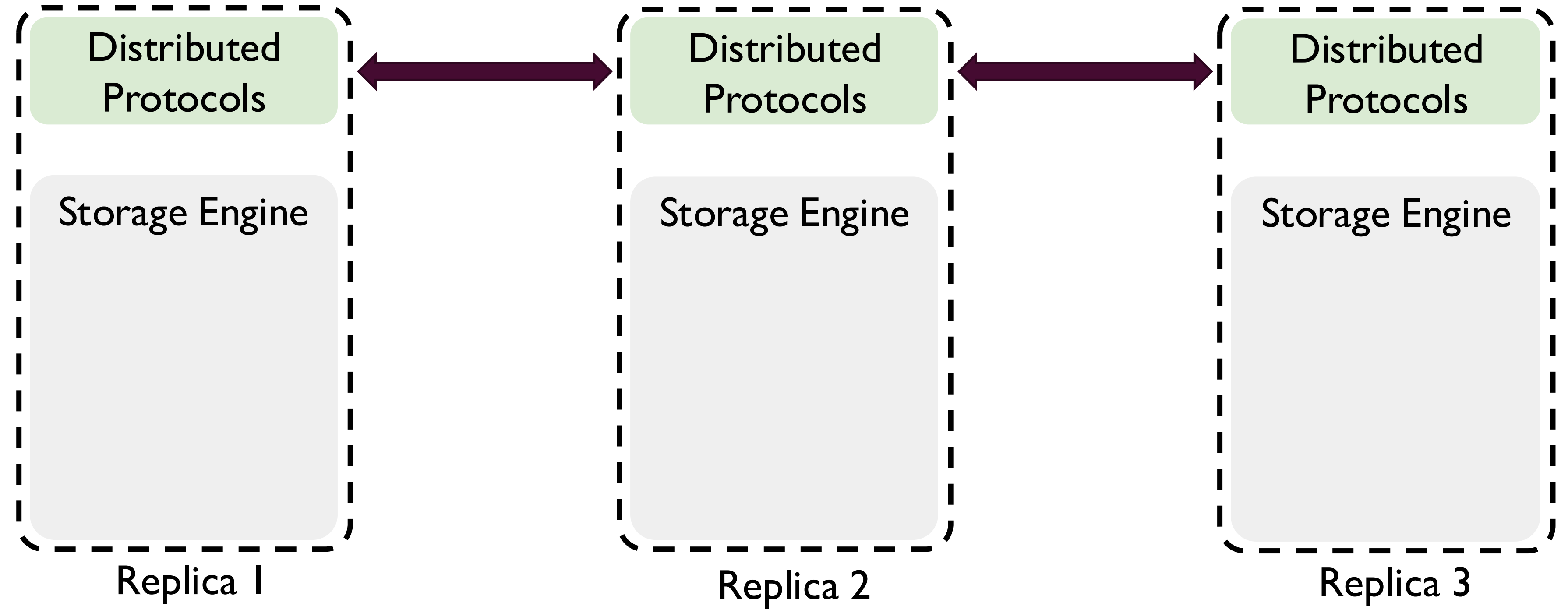
Optimizing Caches In These Storage Systems Is Important



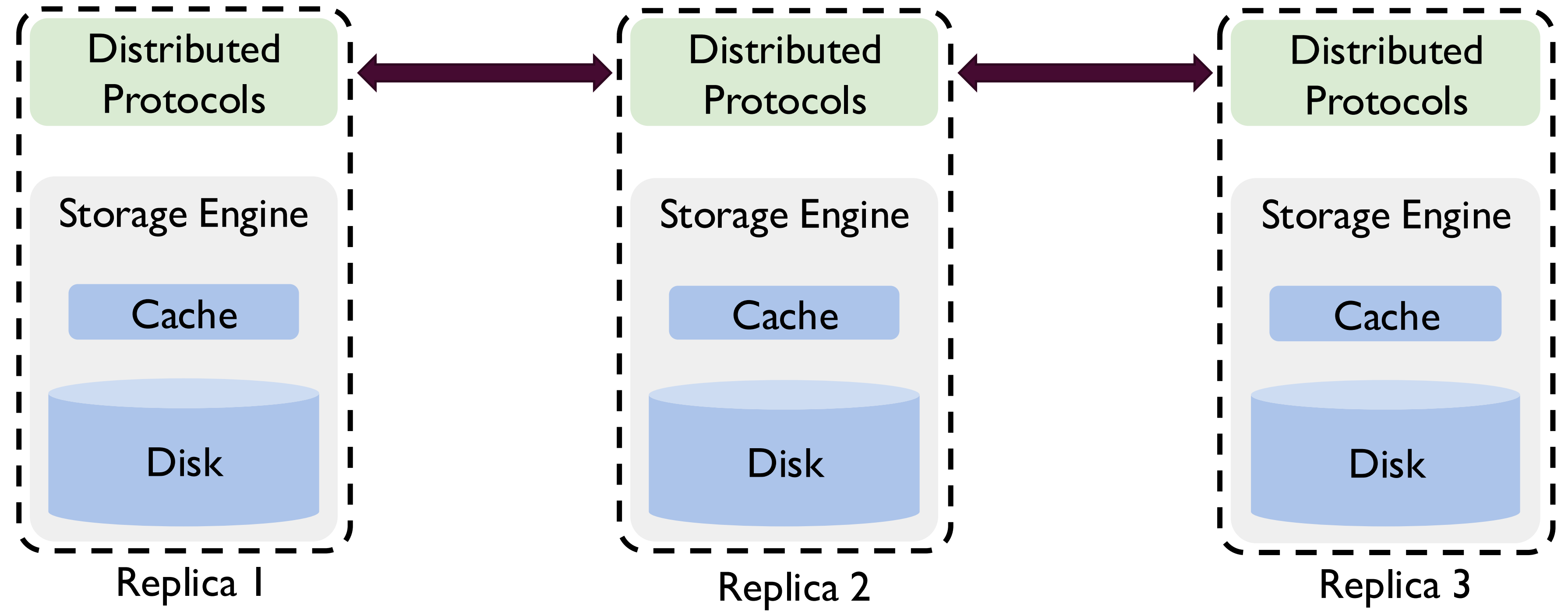
Another Type of Cache That Is Overlooked



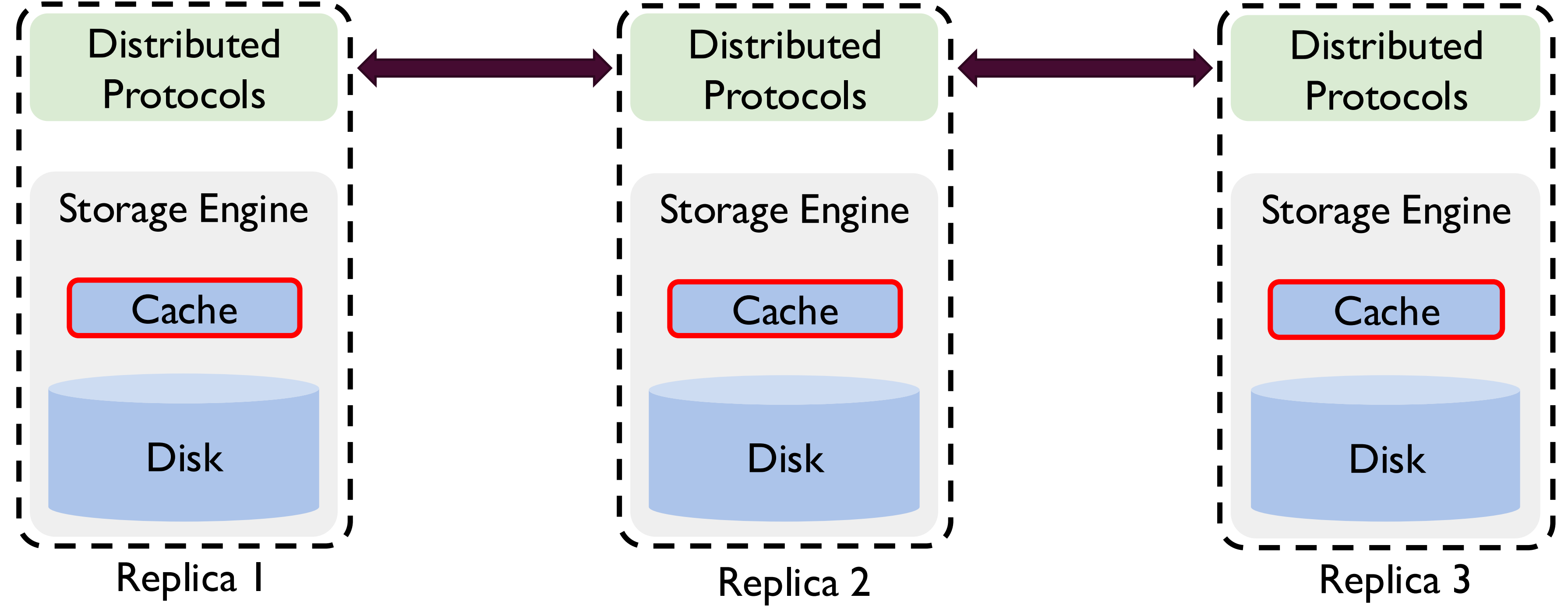
Another Type of Cache That Is Overlooked



Another Type of Cache That Is Overlooked

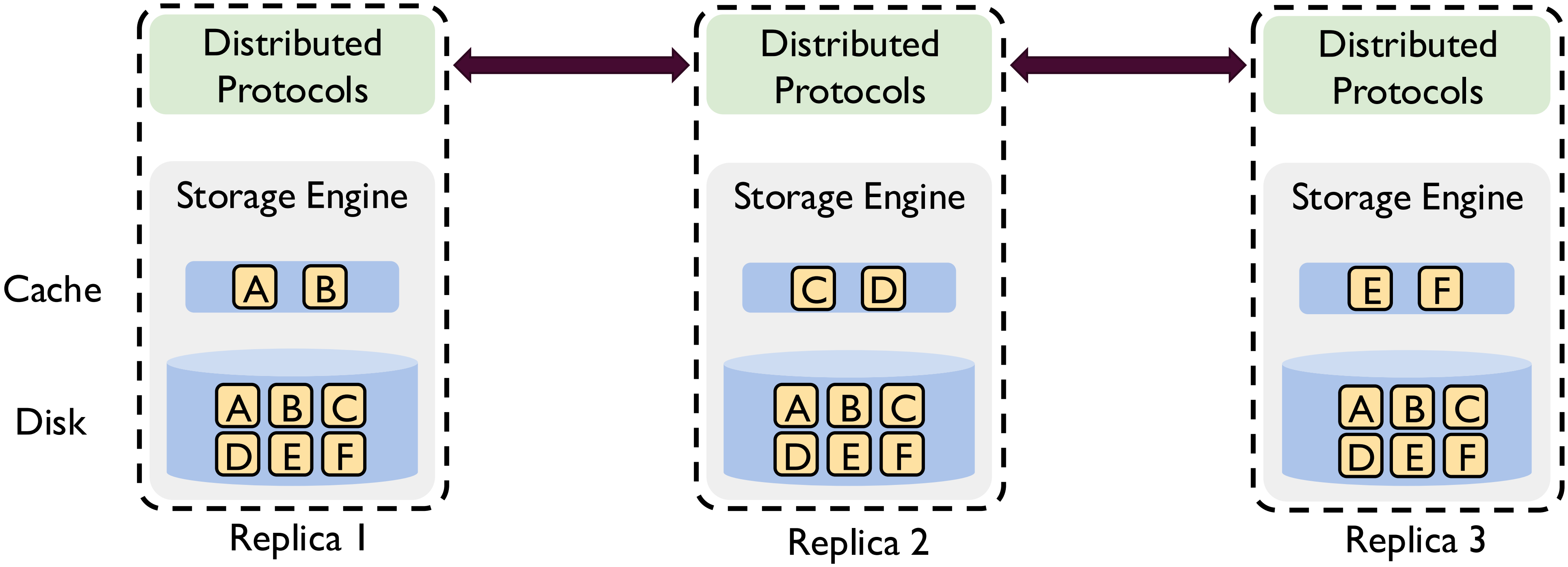


Another Type of Cache That Is Overlooked



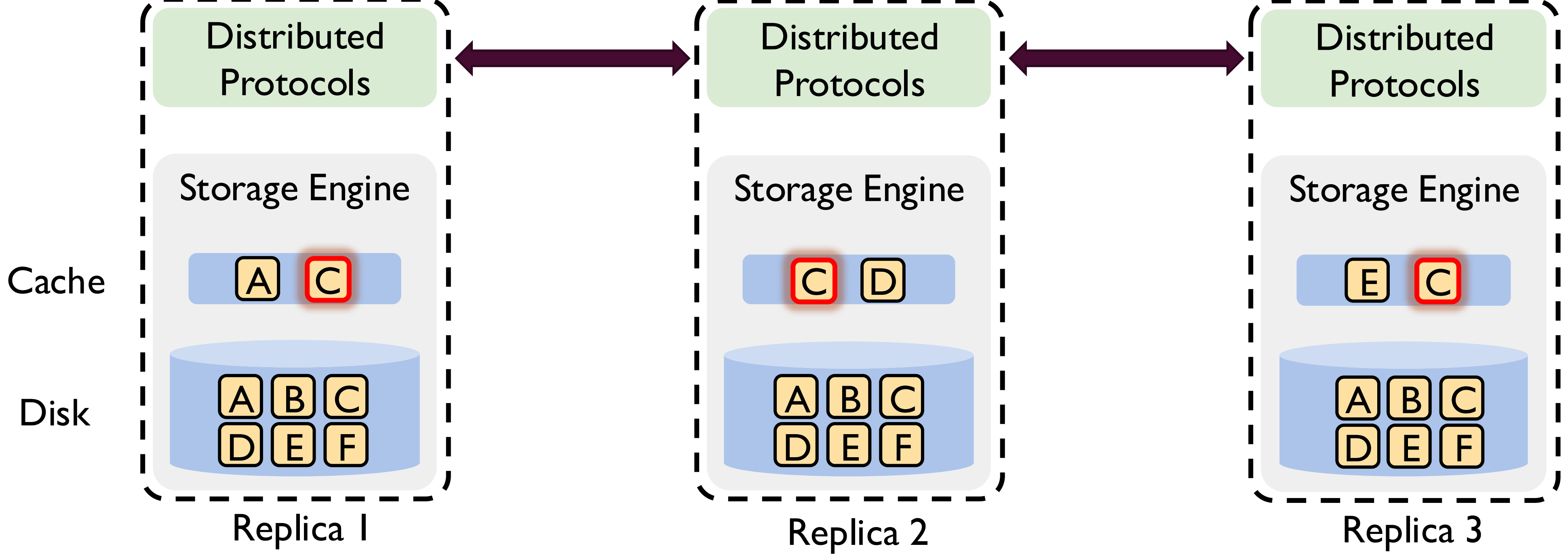
Caches that are internal to each storage engine are overlooked

Are Embedded Caches Effectively Utilized?

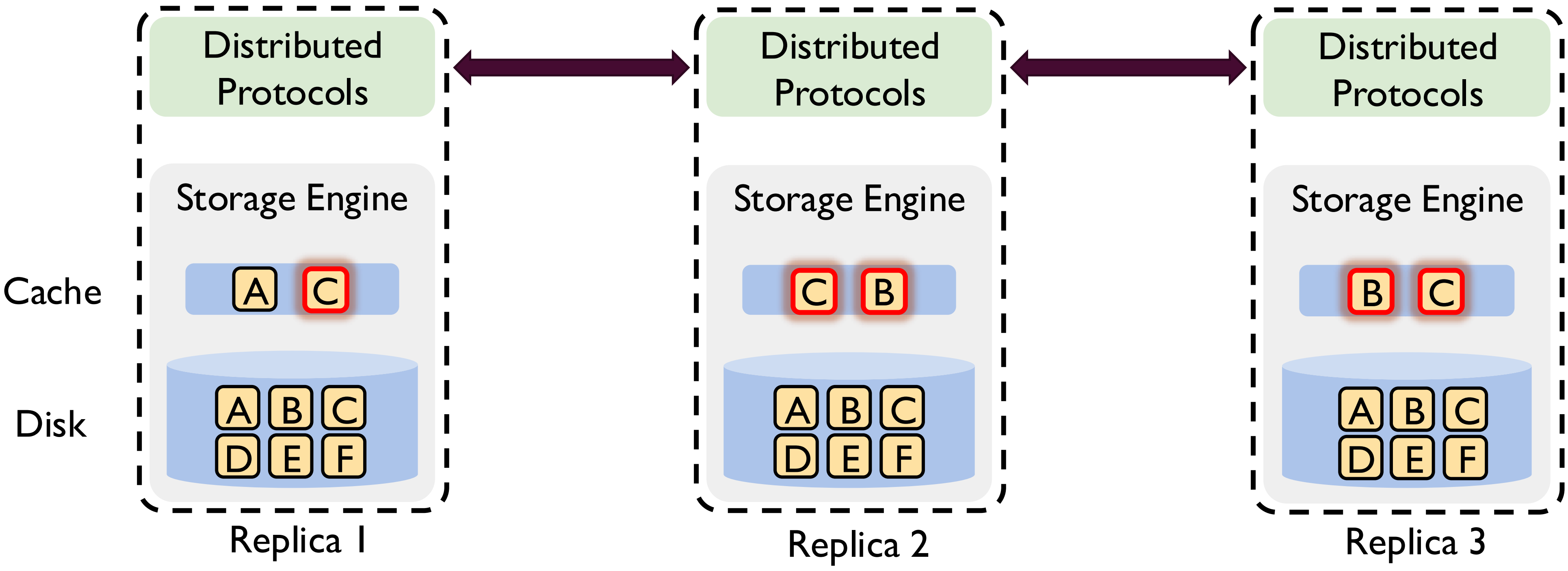


Ideally, the caches together should cover as much of the dataset as possible

Reality : Cache Redundancy

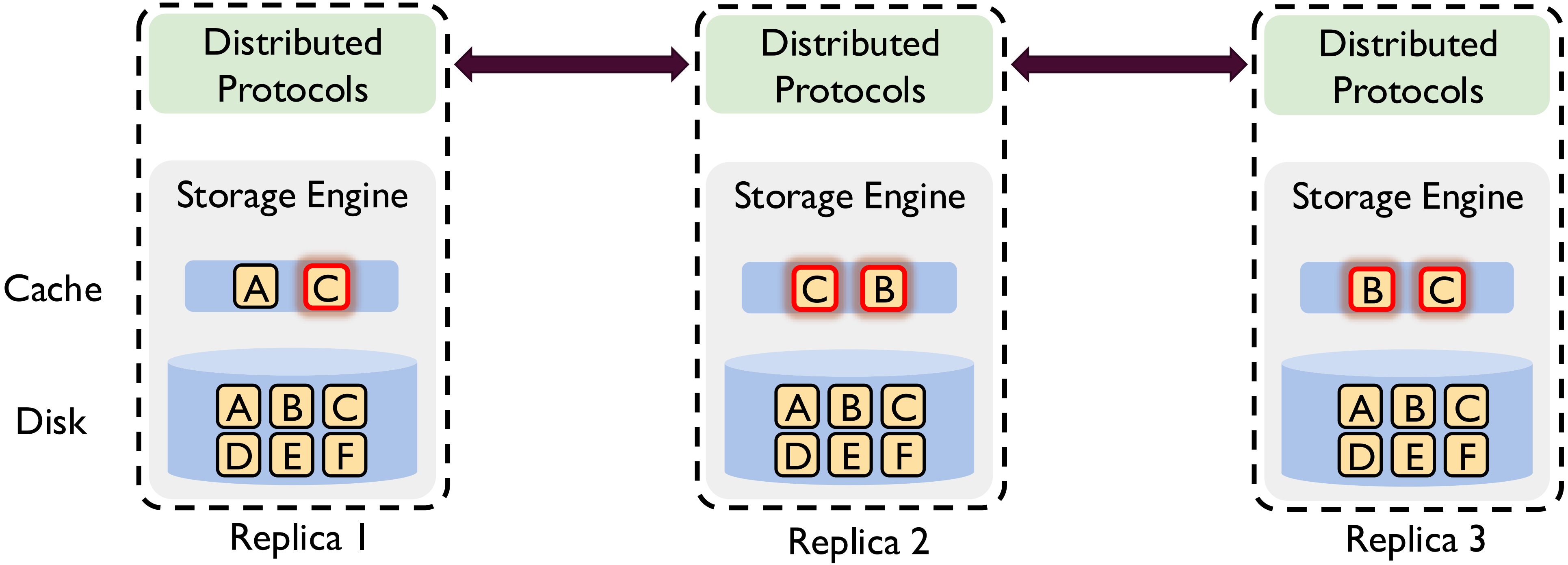


Reality : Cache Redundancy

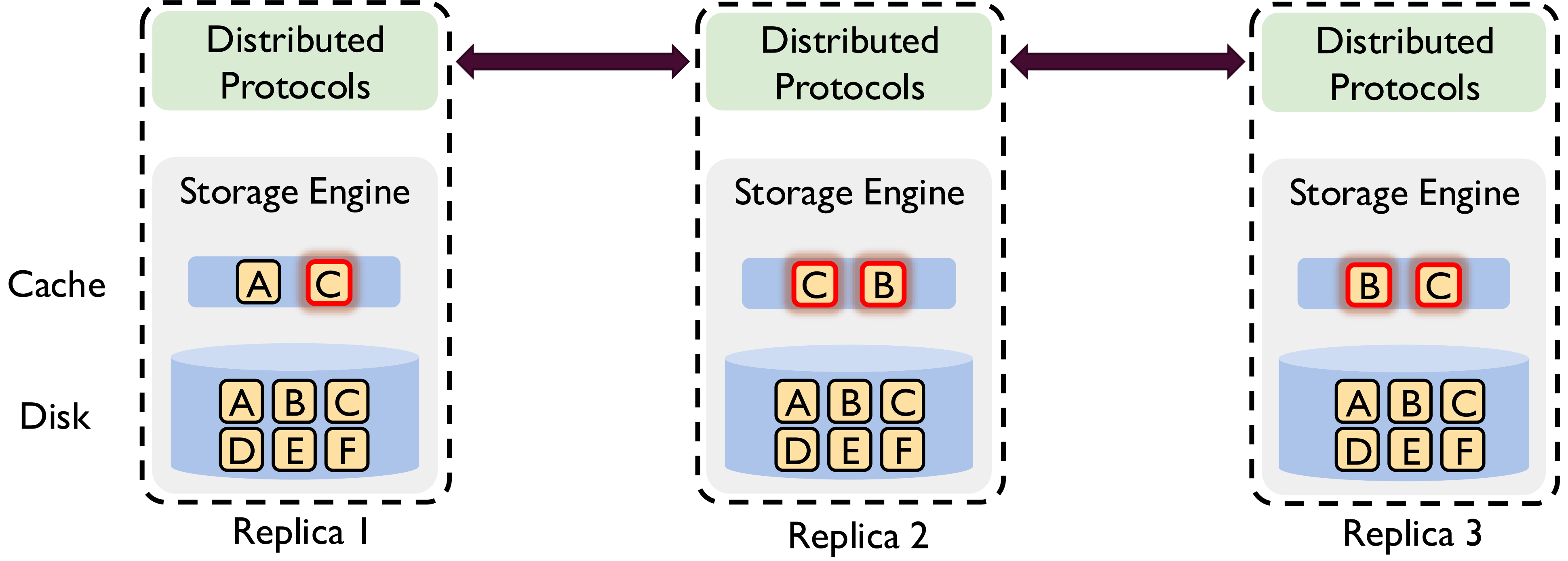


Same objects get cached redundantly across replicas.

Cache Redundancy – Root Cause

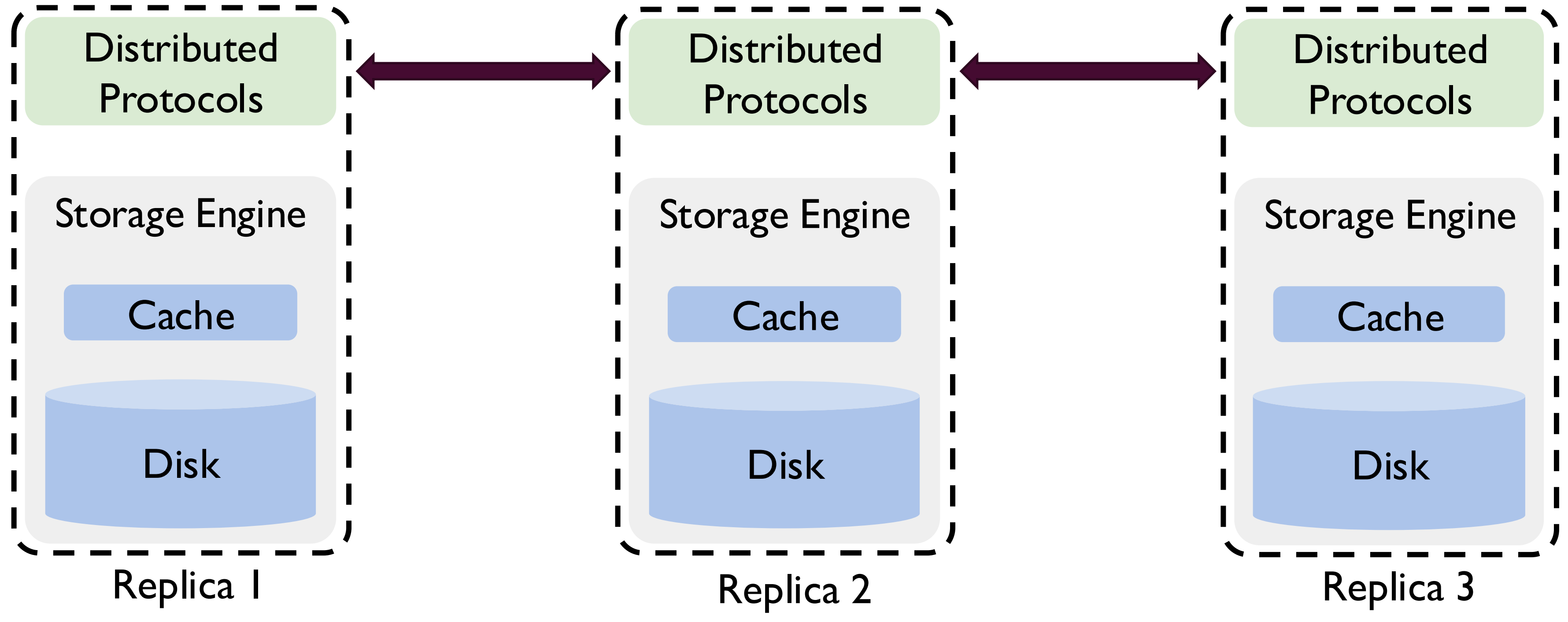


Cache Redundancy – Root Cause

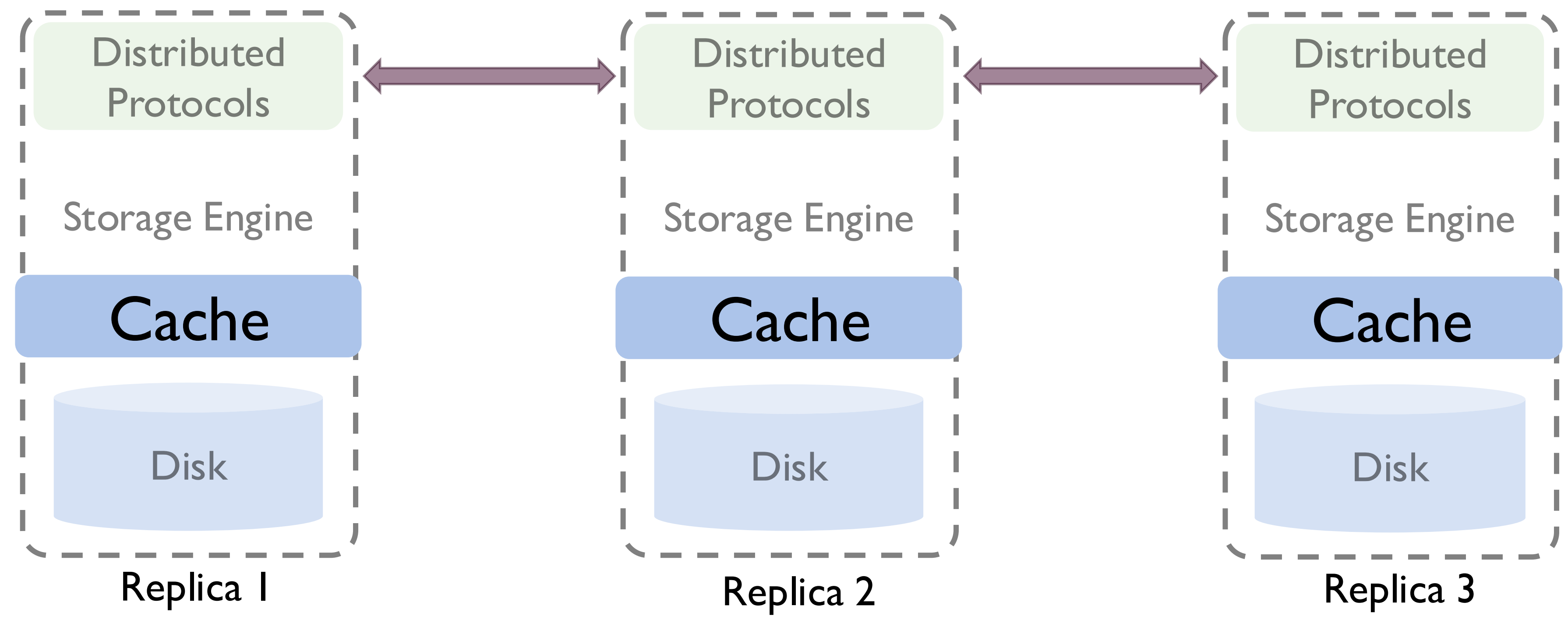


Replicas manage their caches in silos, unaware of each other

Our Solution Is Logically Disaggregated Caches

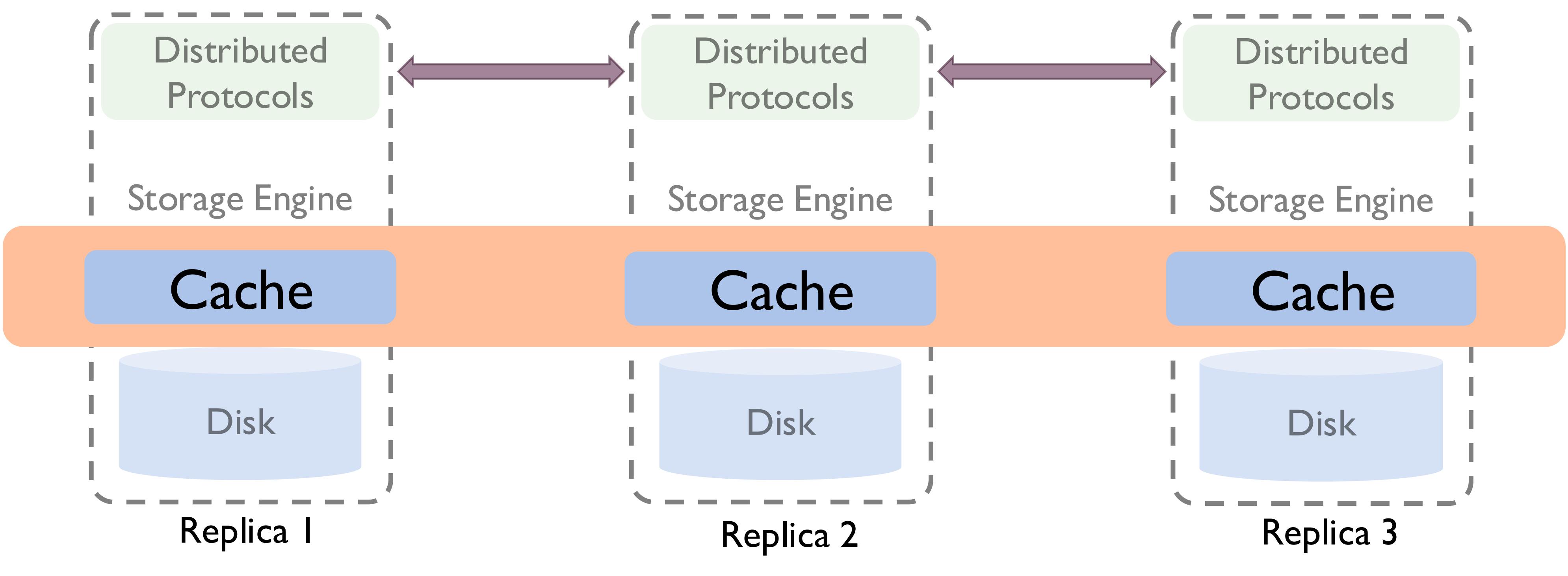


Our Solution Is Logically Disaggregated Caches



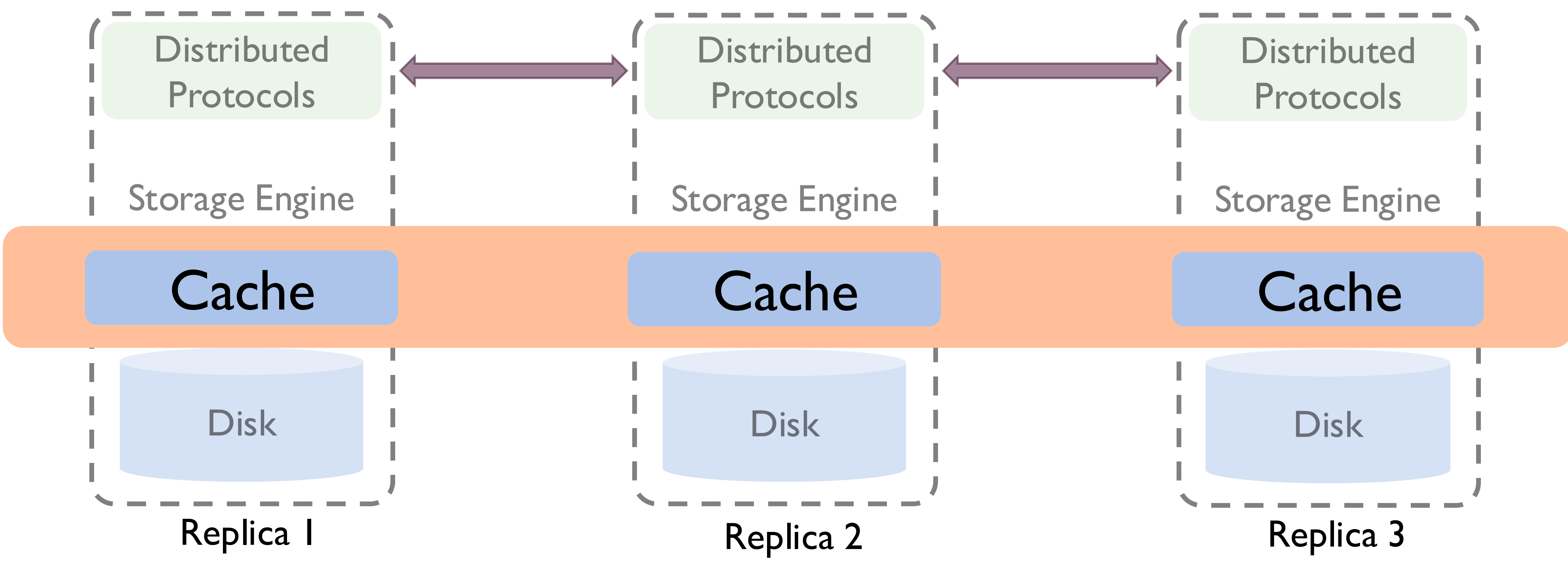
Disaggregate embedded caches from replicas

Our Solution Is Logically Disaggregated Caches



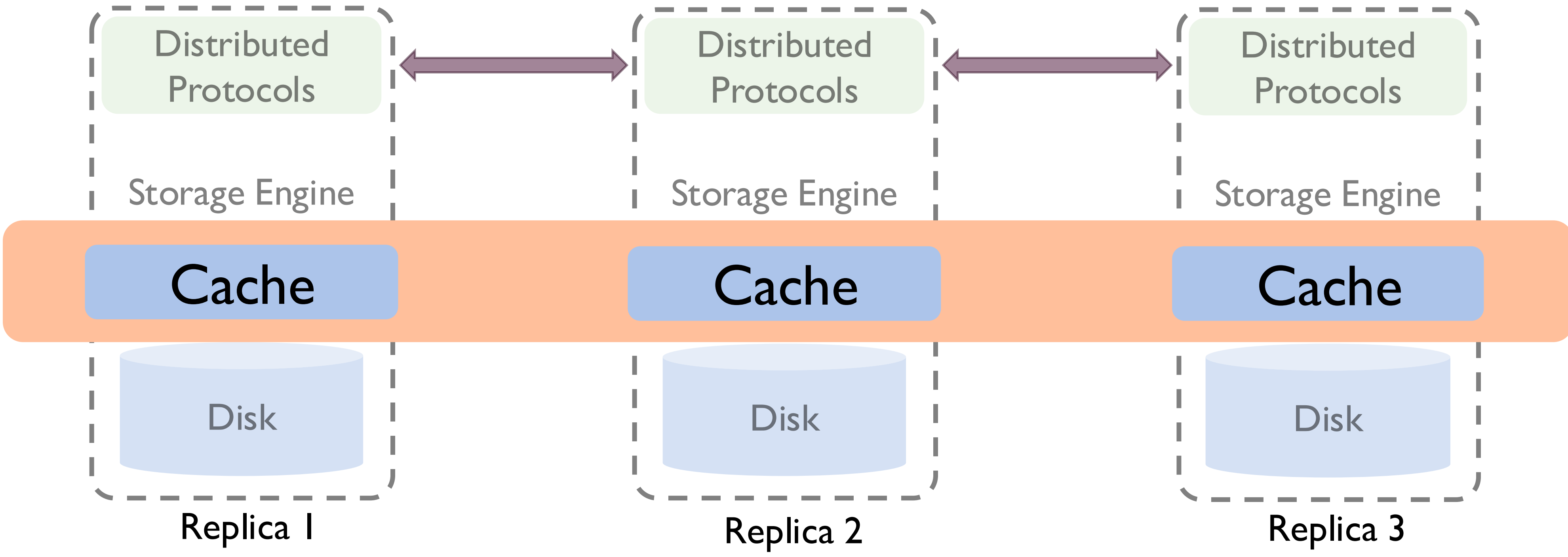
Disaggregate embedded caches from replicas
Manage as a **single, unified, logical** cache

Our Solution Is Logically Disaggregated Caches



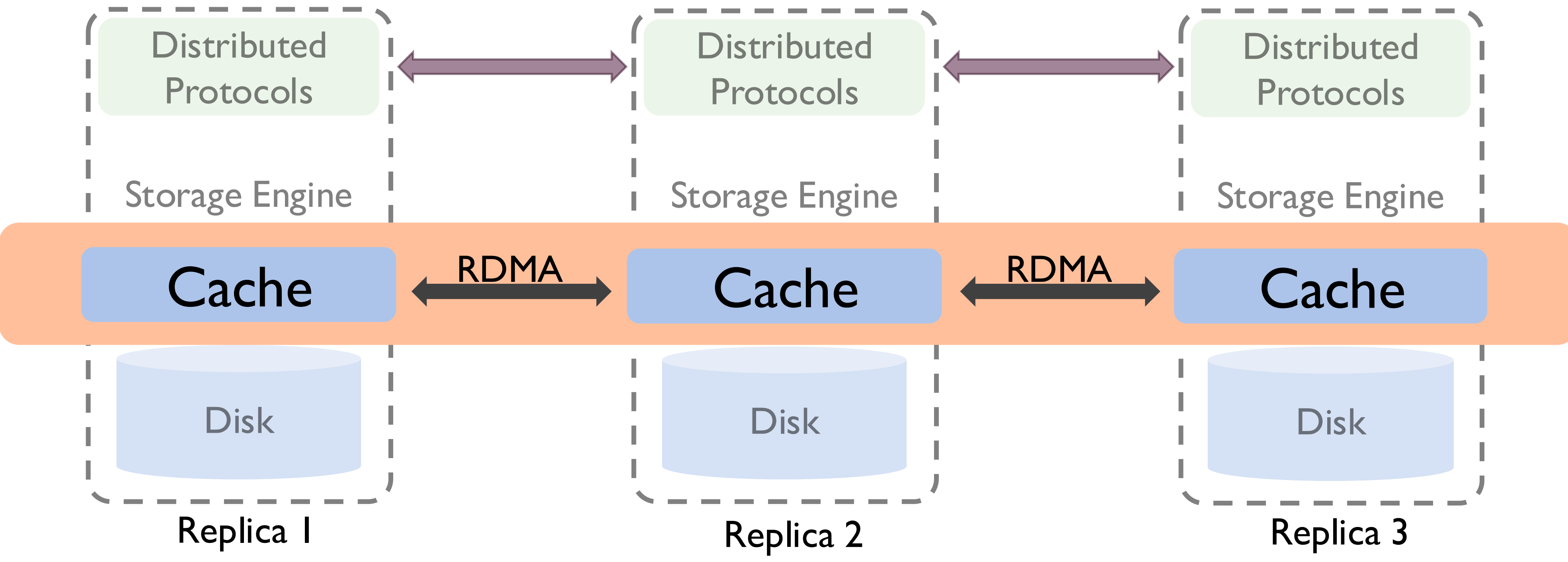
Disaggregate embedded caches from replicas
Manage as a **single, unified, logical** cache
Caches are **aware** of each other — no more silos

Our Solution Is Logically Disaggregated Caches



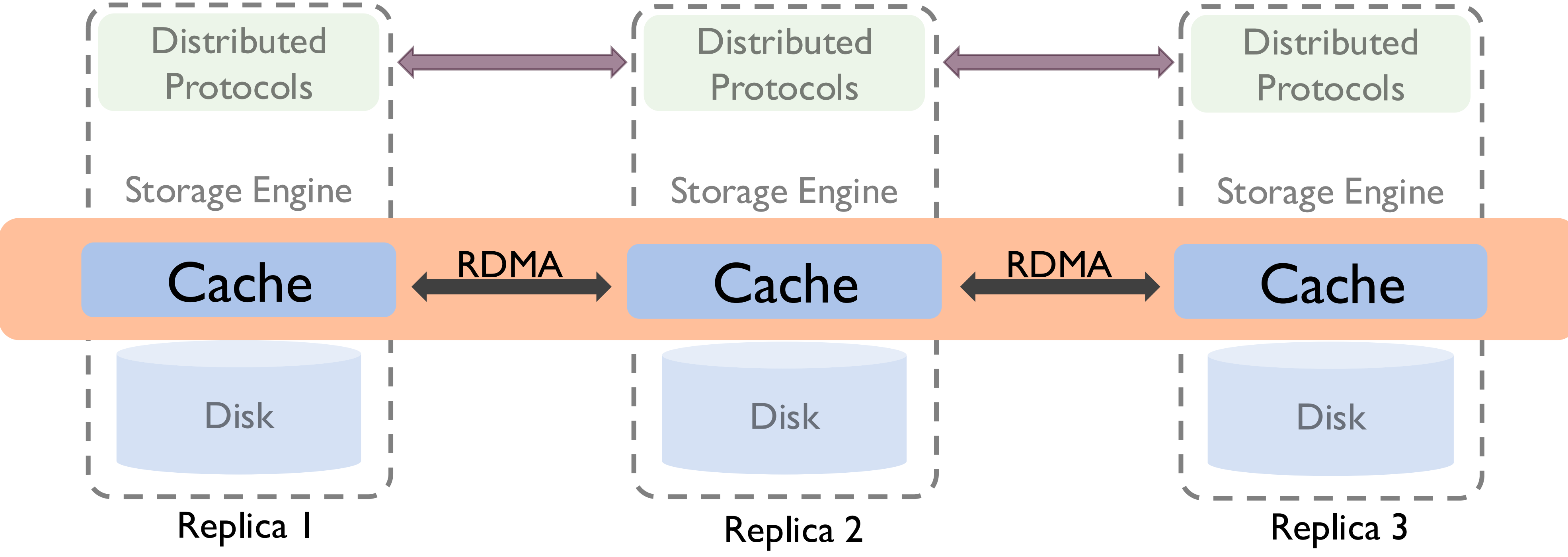
Disaggregate embedded caches from replicas
Manage as a **single, unified, logical** cache
Caches are **aware** of each other — no more silos
Any replica can access **any part** of the logical cache

Our Solution Is Logically Disaggregated Caches



Disaggregate embedded caches from replicas
Manage as a **single, unified, logical** cache
Caches are **aware** of each other — no more silos
Any replica can access **any part** of the logical cache

Our Solution Is Logically Disaggregated Caches

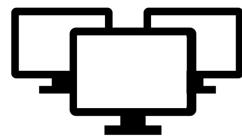


Disaggregate embedded caches from replicas
Manage as a **single, unified, logical** cache
Caches are **aware** of each other — no more silos
Any replica can access **any part** of the logical cache
Improve performance by upto 5.9X

Outline

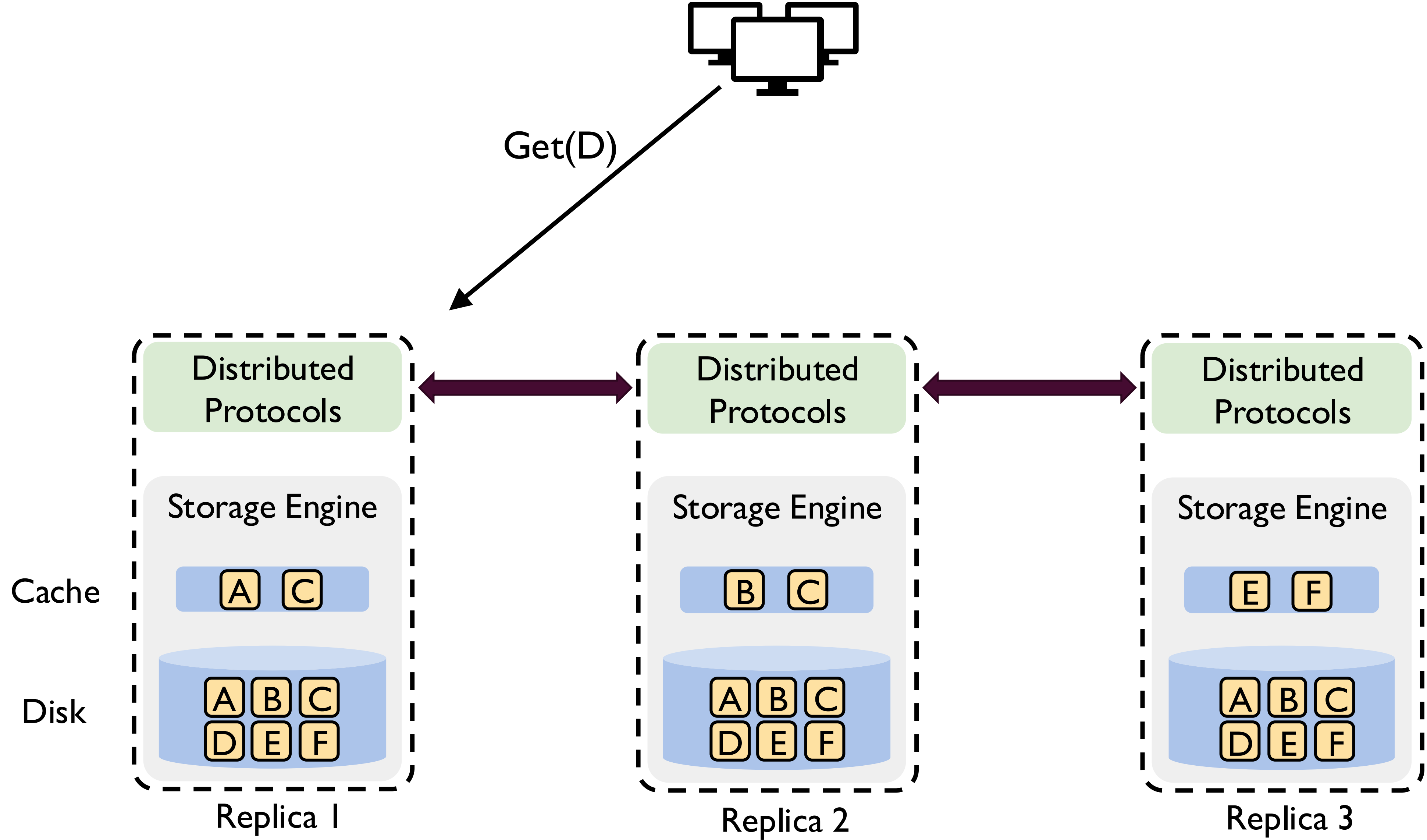
- Introduction
- Study
- Our Idea - Logically Disaggregated Cache
- Implementations
- Evaluation

Causes of Cache Redundancy – READs and WRITEs

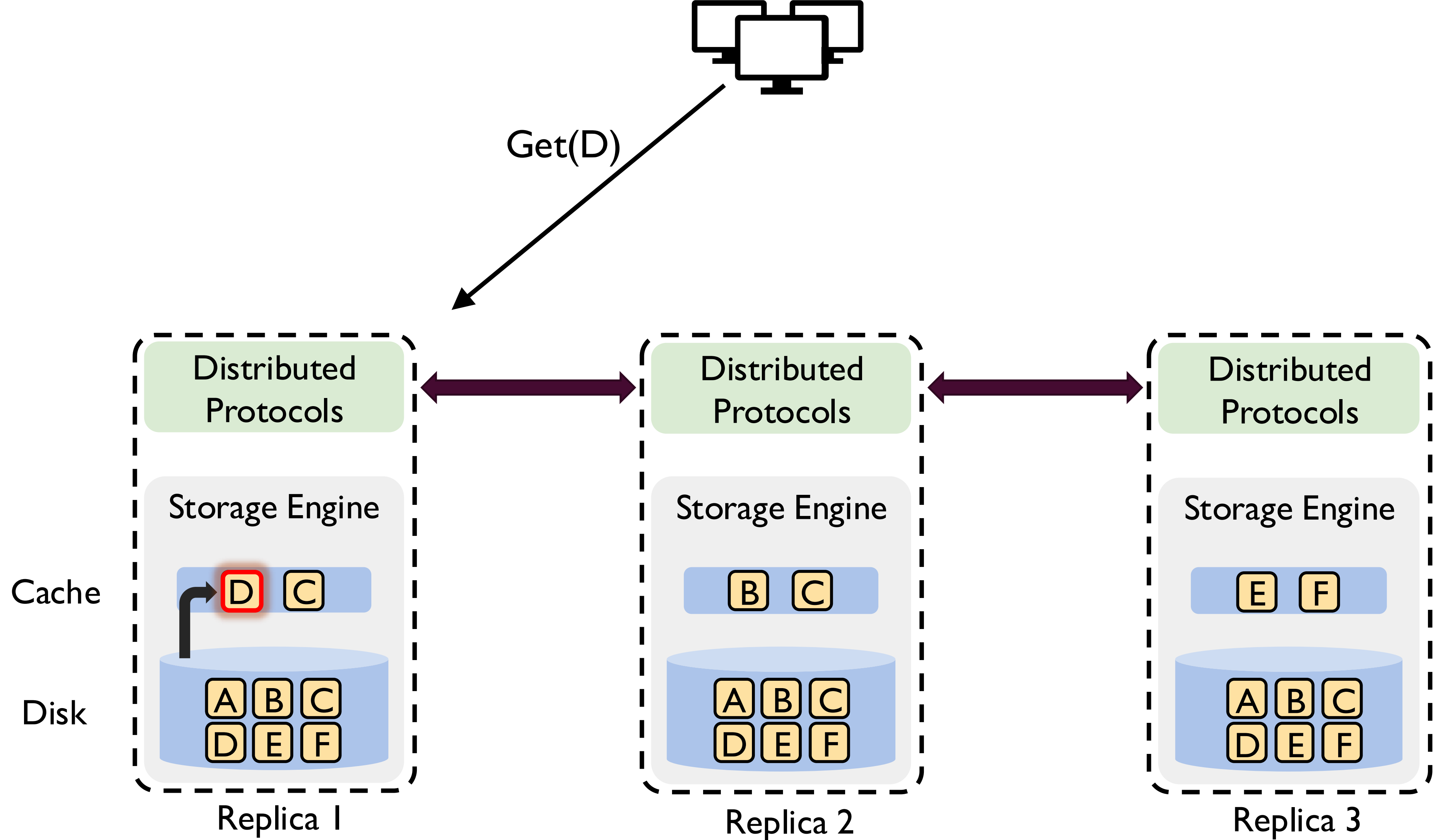


	Replica 1	Replica 2	Replica 3
	Distributed Protocols	Distributed Protocols	Distributed Protocols
	Storage Engine	Storage Engine	Storage Engine
Cache	A C	B C	E F
Disk	A B C D E F	A B C D E F	A B C D E F

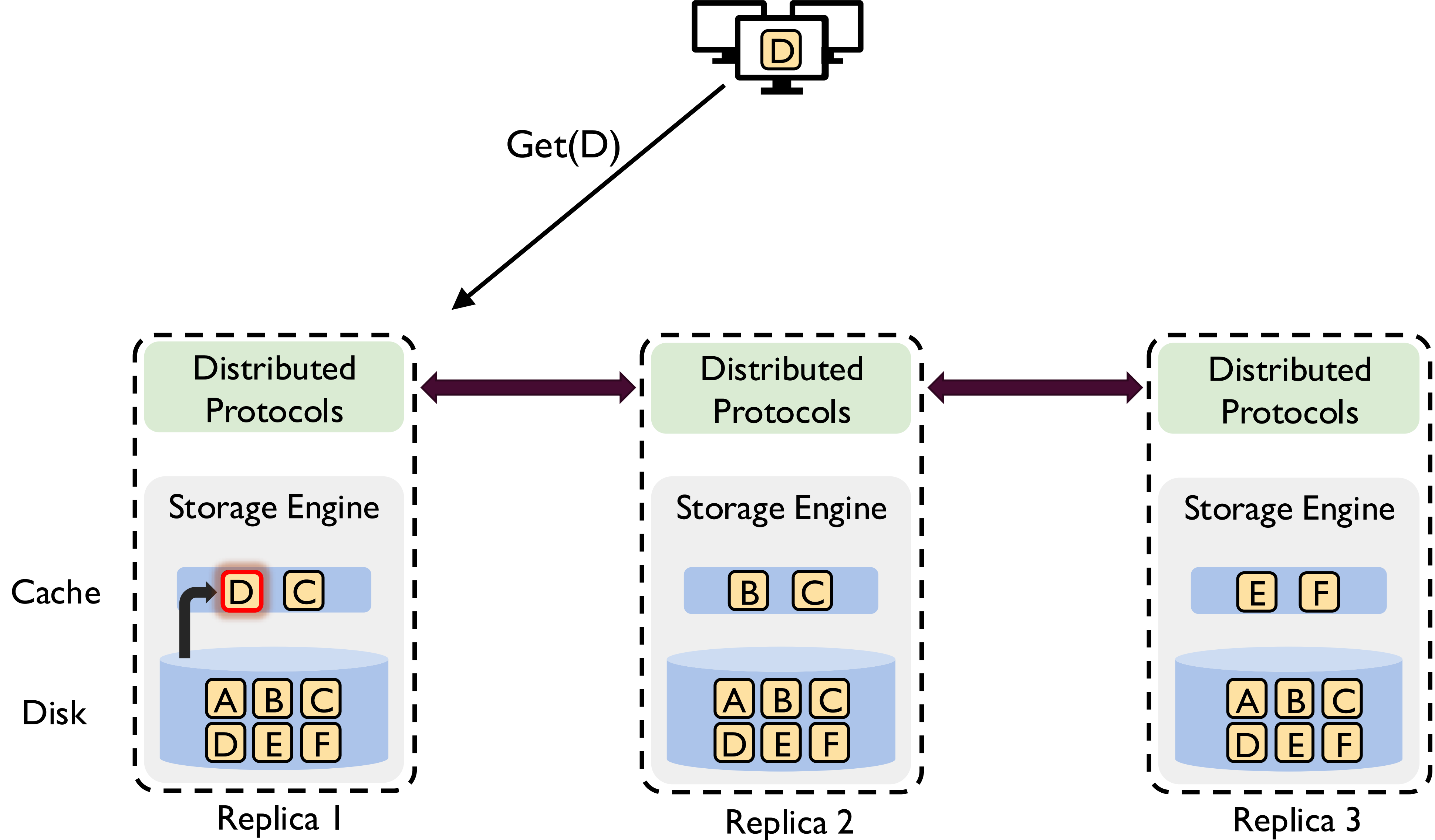
Causes of Cache Redundancy – READs and WRITEs



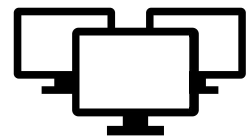
Causes of Cache Redundancy – READs and WRITEs



Causes of Cache Redundancy – READs and WRITEs



Causes of Cache Redundancy – READs and WRITEs



Distributed Protocols

Storage Engine

Cache

D C

Disk

A B C
D E F

Replica 1

Distributed Protocols

Storage Engine

B C

A B C
D E F

Replica 2

Distributed Protocols

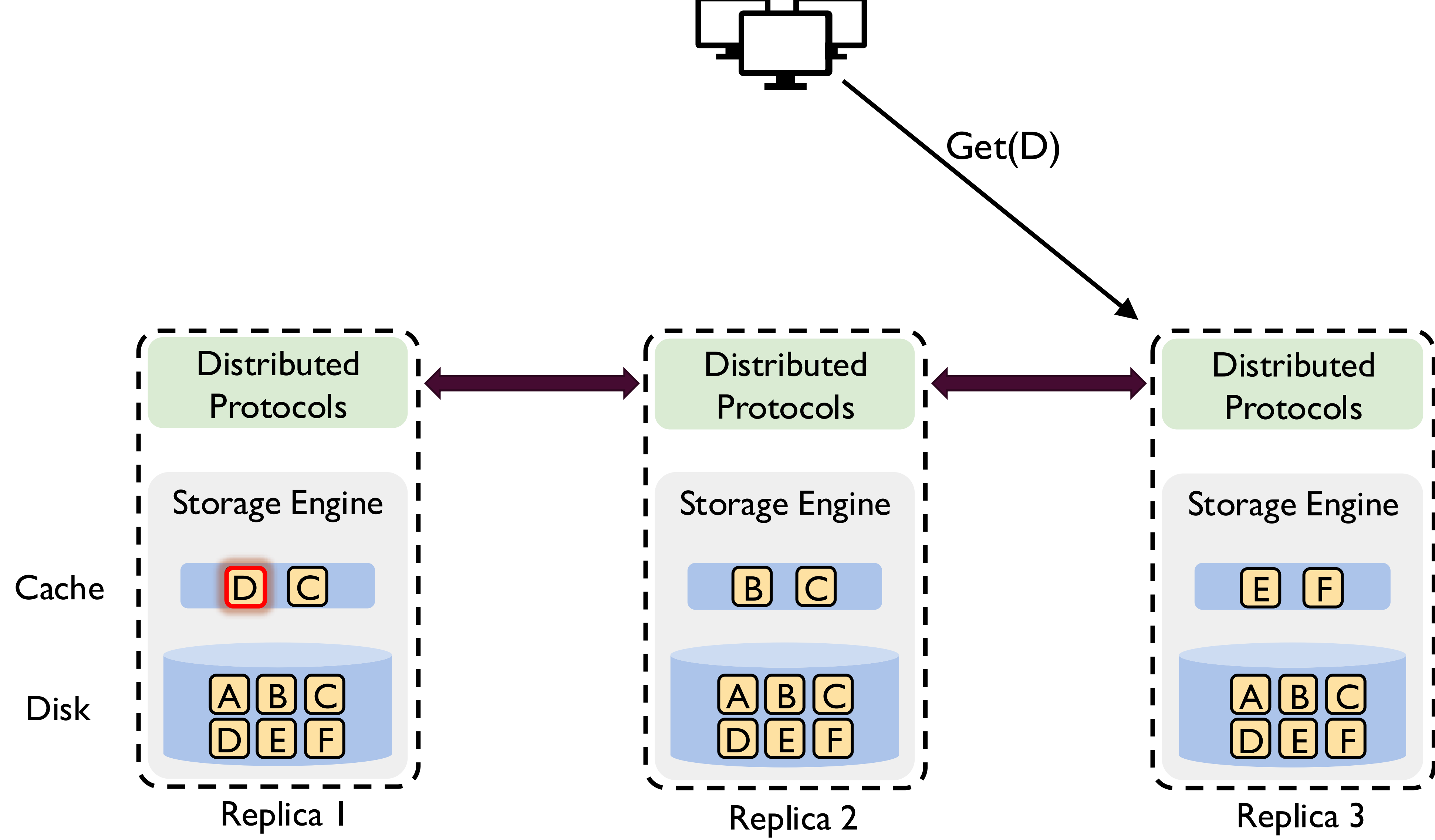
Storage Engine

E F

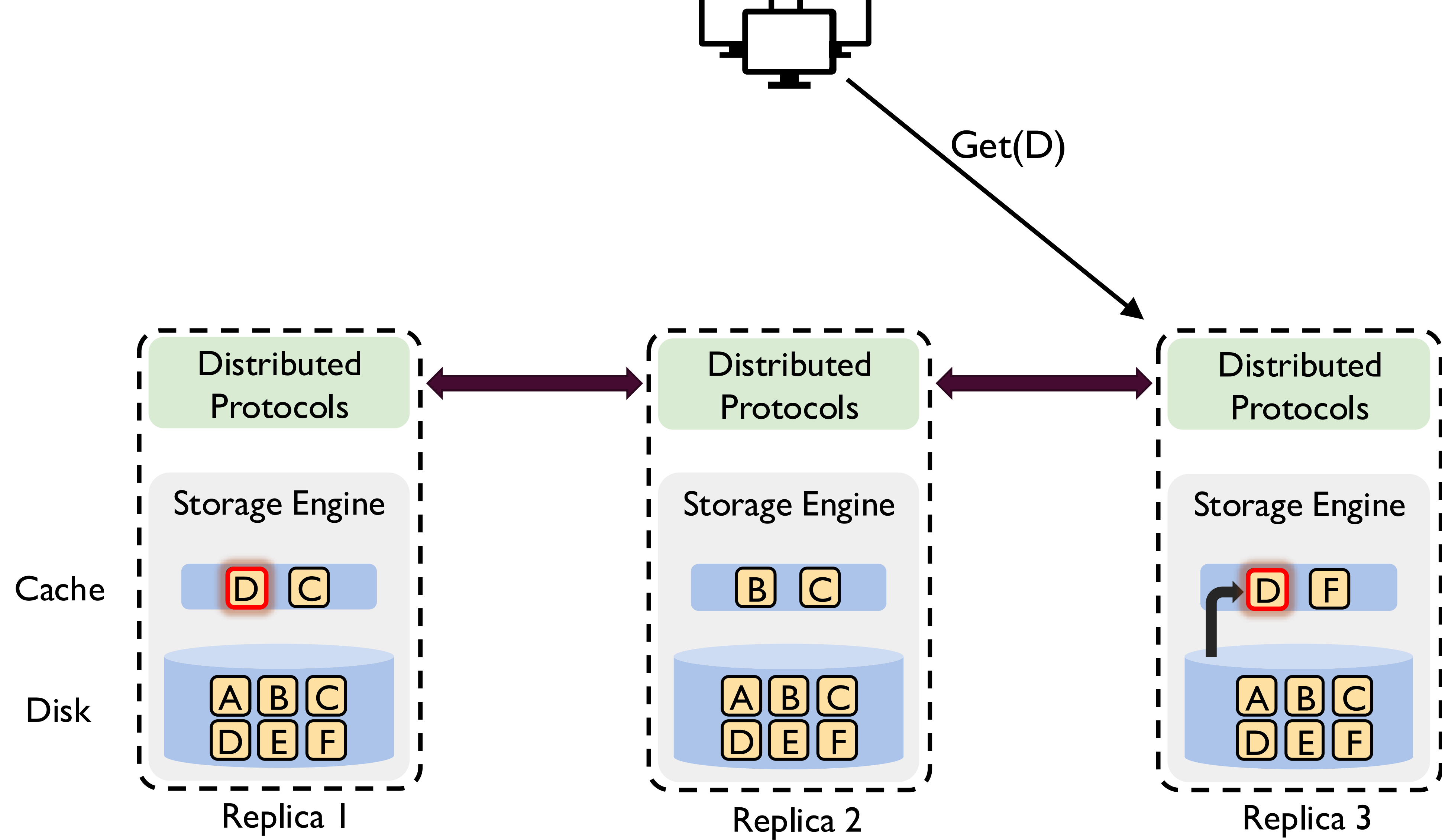
A B C
D E F

Replica 3

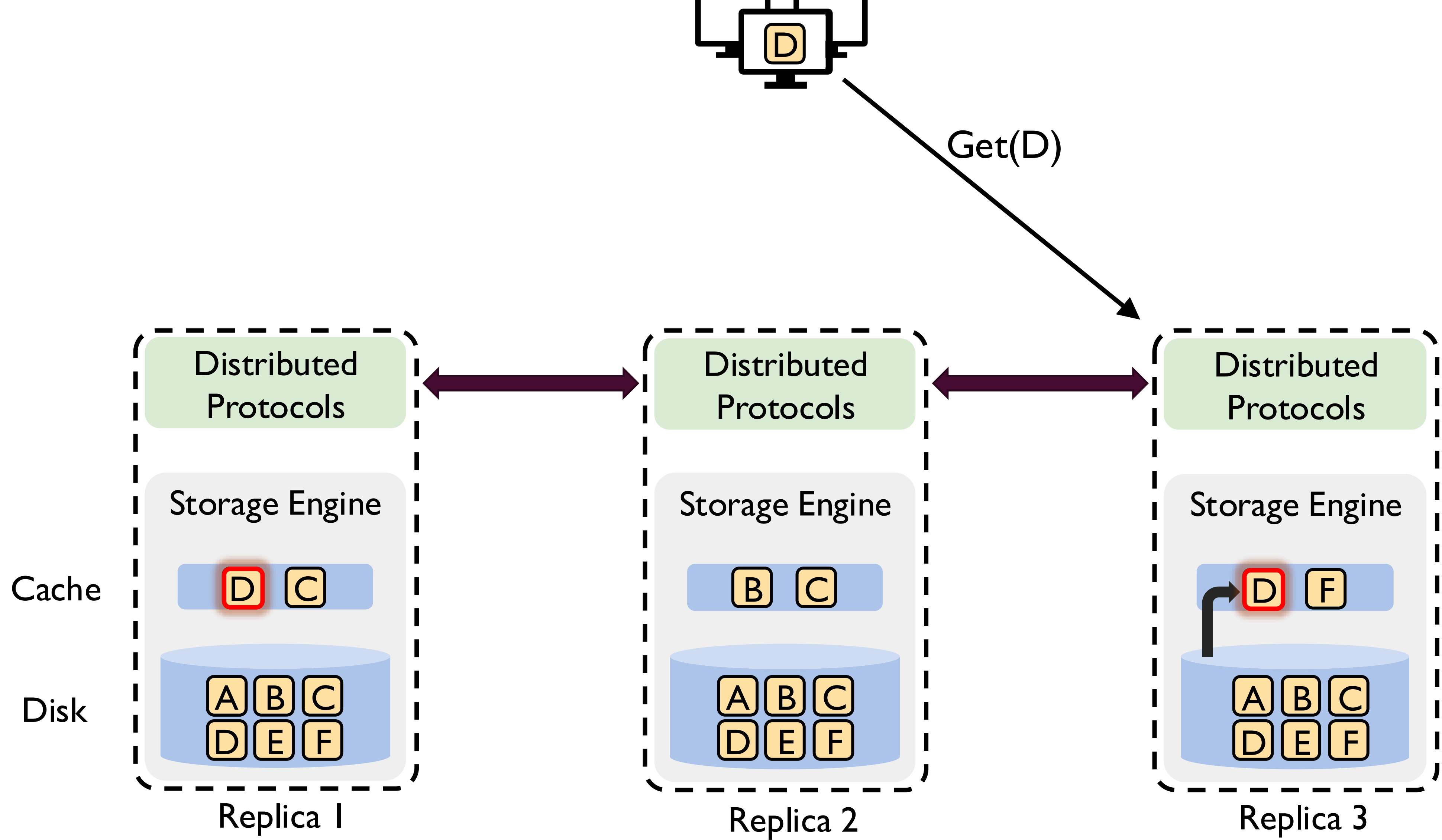
Causes of Cache Redundancy – READs and WRITEs



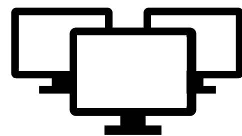
Causes of Cache Redundancy – READs and WRITEs



Causes of Cache Redundancy – READs and WRITEs



Causes of Cache Redundancy – READs and WRITEs



Distributed Protocols

Storage Engine

Cache D C

Disk A B C D E F

Replica 1

Distributed Protocols

Storage Engine

Cache B C

Disk A B C D E F

Replica 2

Distributed Protocols

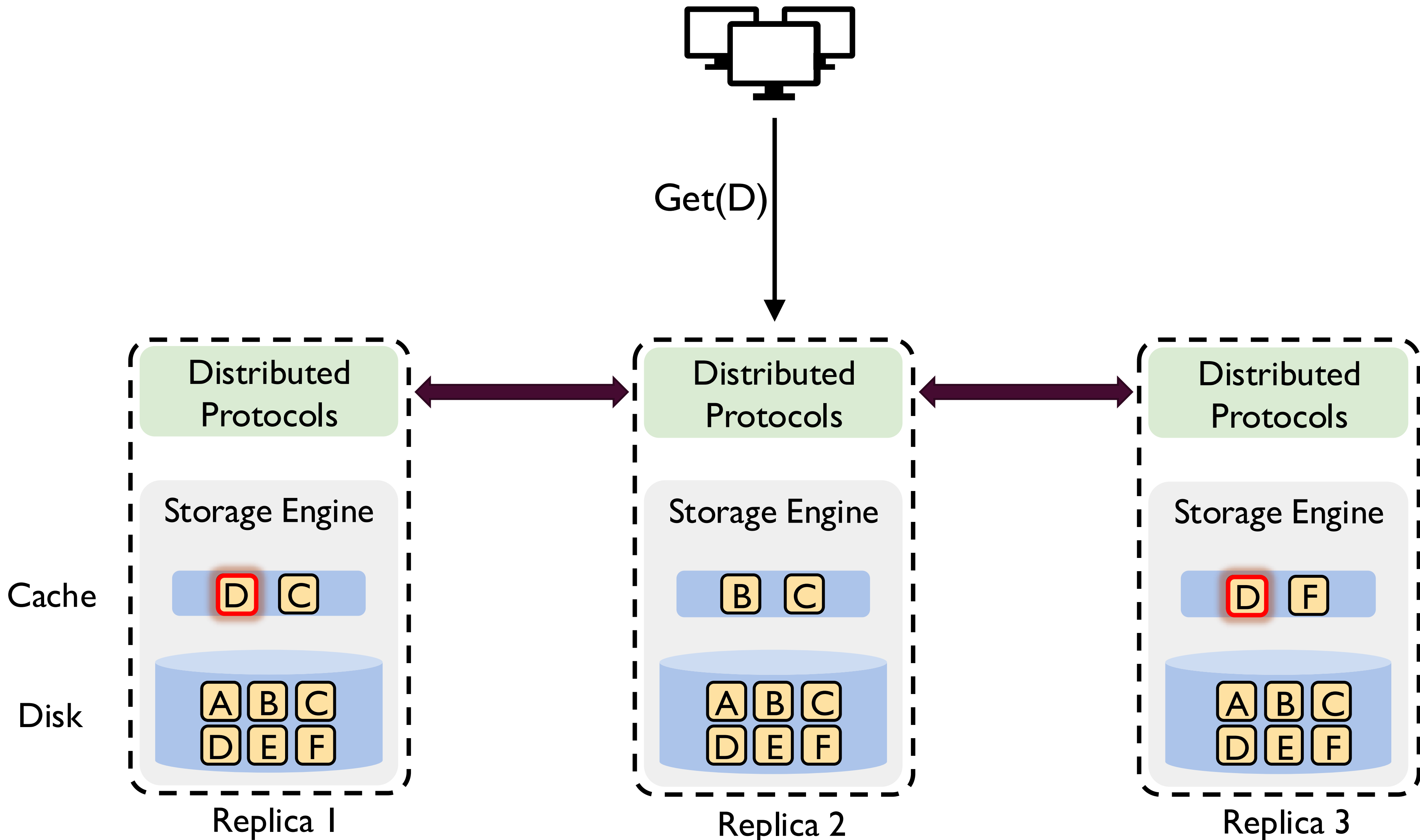
Storage Engine

Cache D F

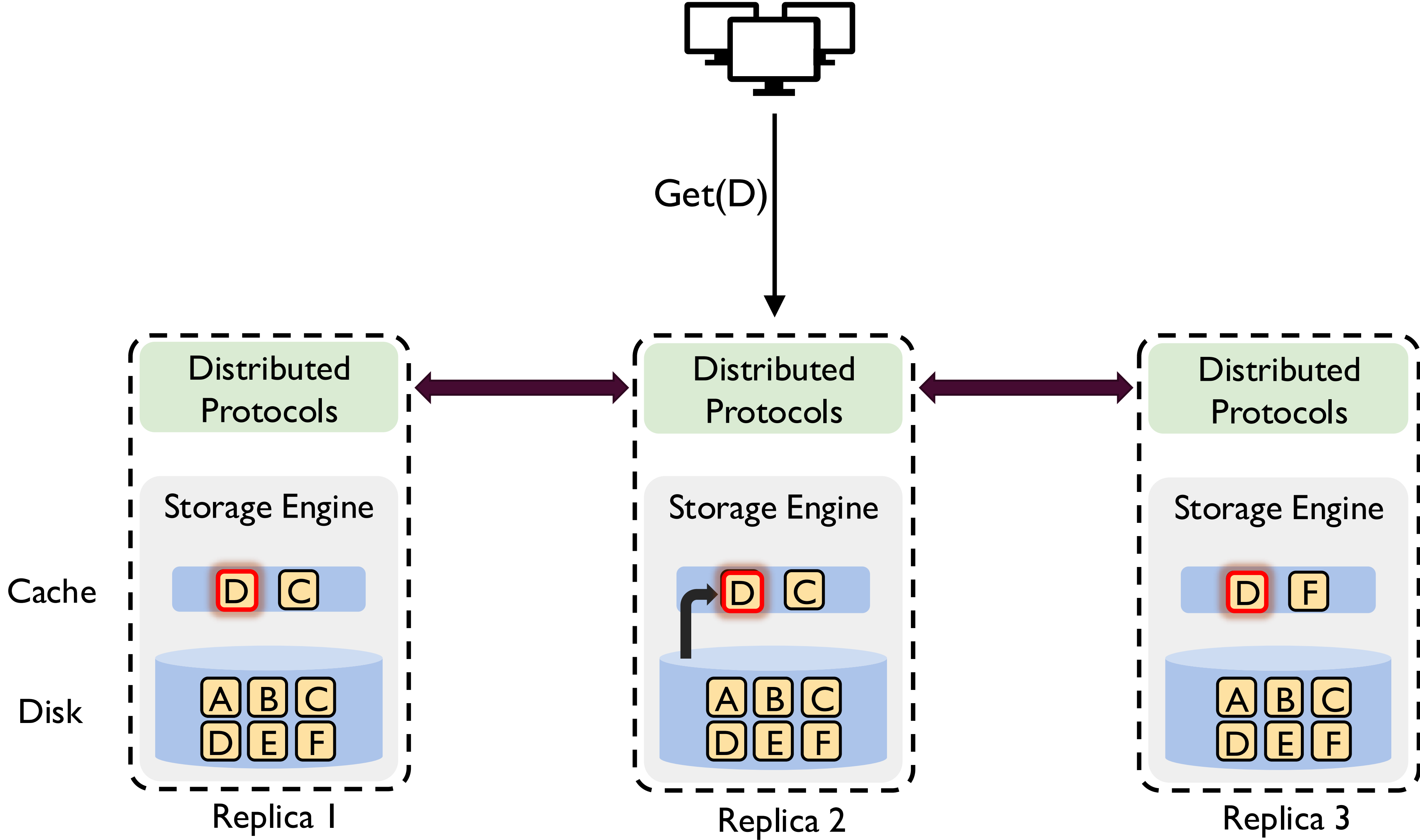
Disk A B C D E F

Replica 3

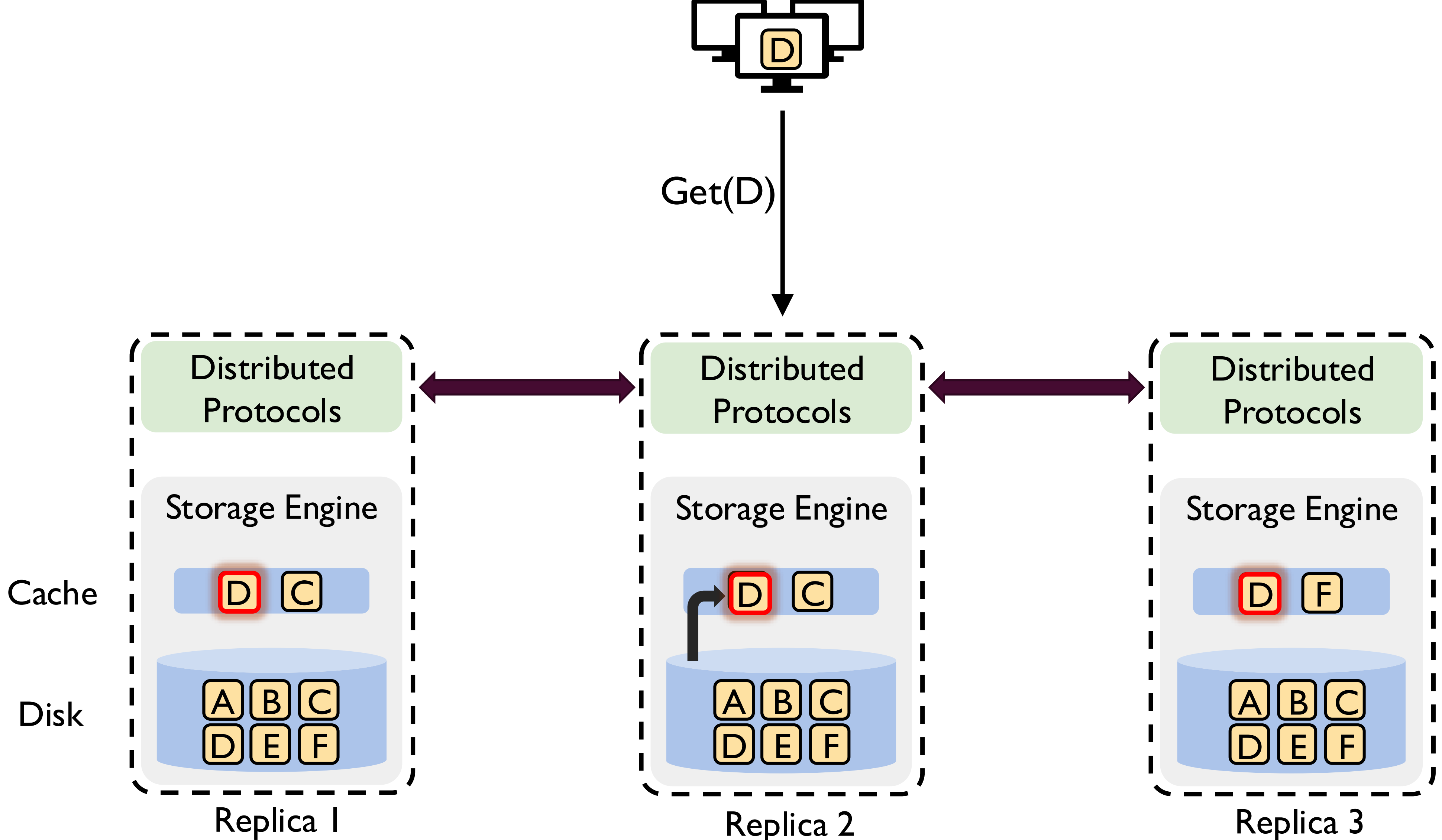
Causes of Cache Redundancy – READs and WRITEs



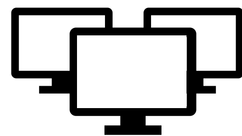
Causes of Cache Redundancy – READs and WRITEs



Causes of Cache Redundancy – READs and WRITEs



Causes of Cache Redundancy – READs and WRITEs



Distributed Protocols | Distributed Protocols | Distributed Protocols

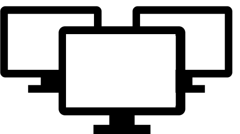
Storage Engine | Storage Engine | Storage Engine

Cache: D C | D C | D F

Disk: A B C D E F | A B C D E F | A B C D E F

Replica 1 | Replica 2 | Replica 3

Causes of Cache Redundancy – READs and WRITEs



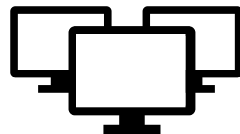
Cache

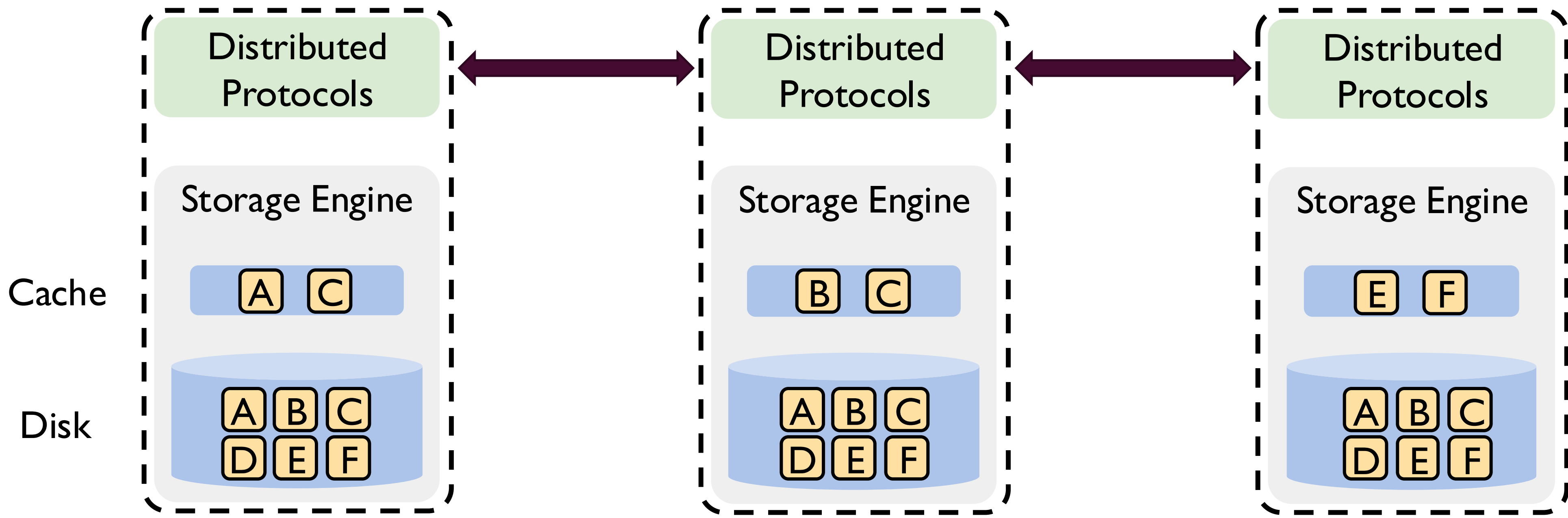
Disk

Replica 1	Replica 2	Replica 3
Distributed Protocols	Distributed Protocols	Distributed Protocols
Storage Engine	Storage Engine	Storage Engine
Cache: D C	Cache: D C	Cache: D F
Disk: A B C D E F	Disk: A B C D E F	Disk: A B C D E F

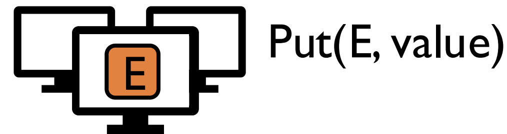
Read-Induced Redundancy

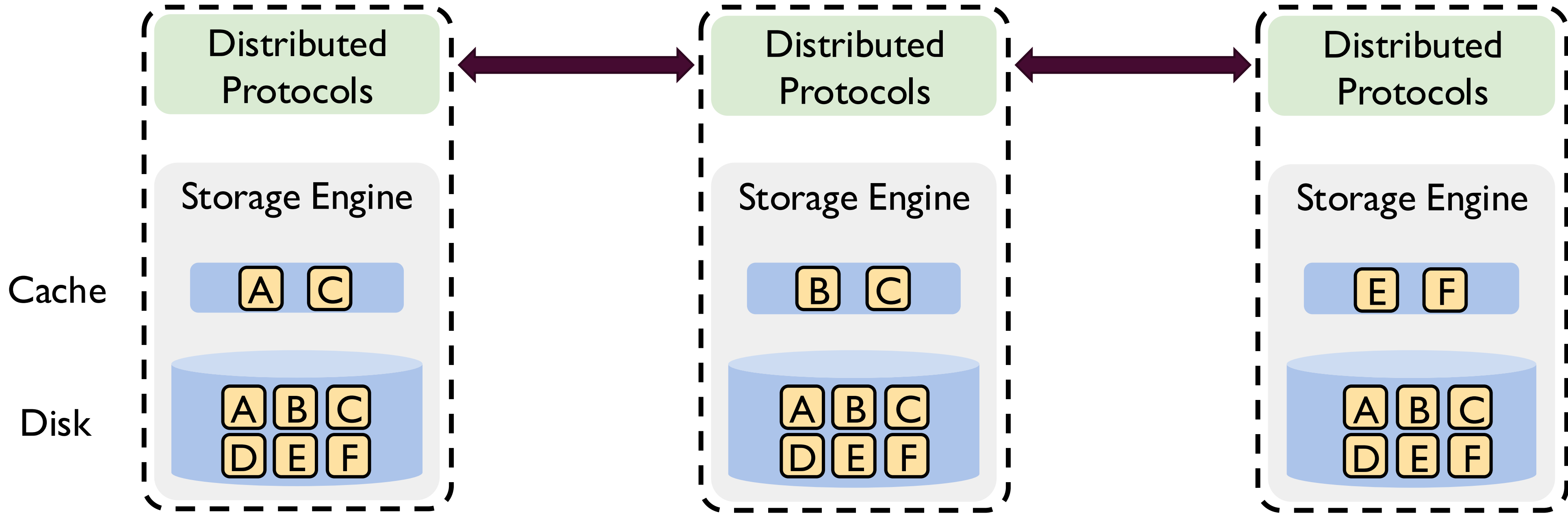
Causes of Cache Redundancy – READs and WRITEs



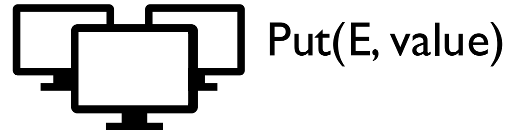


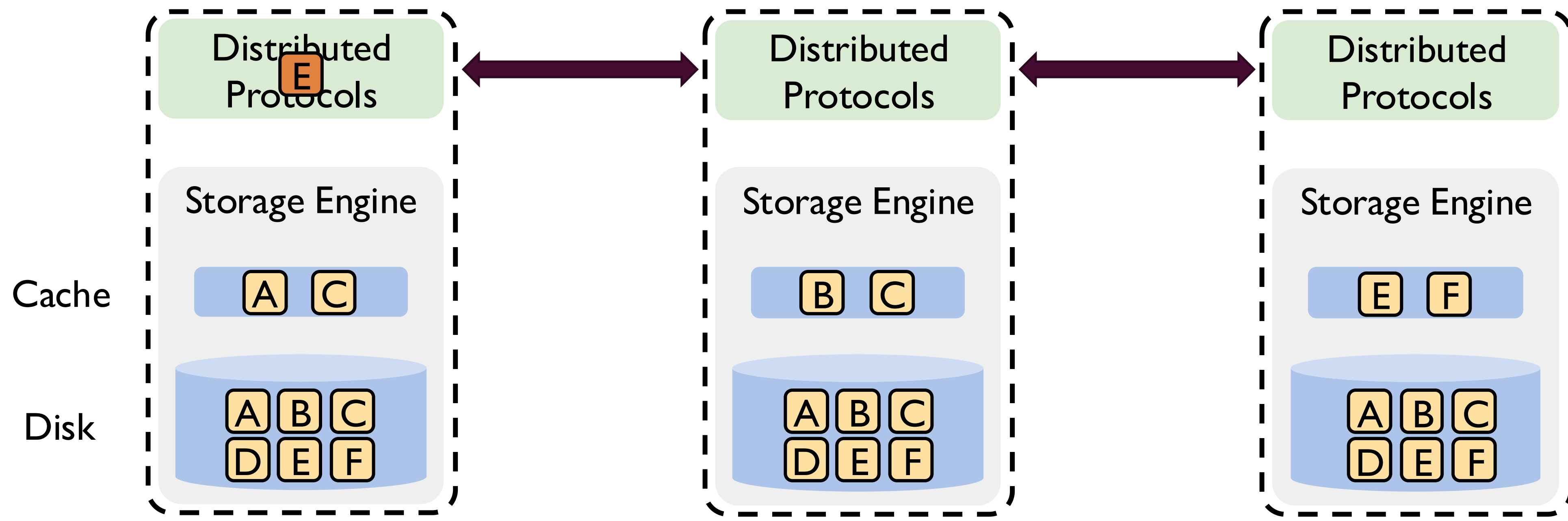
Causes of Cache Redundancy – READs and WRITEs



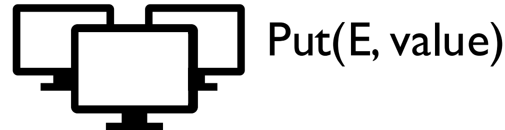


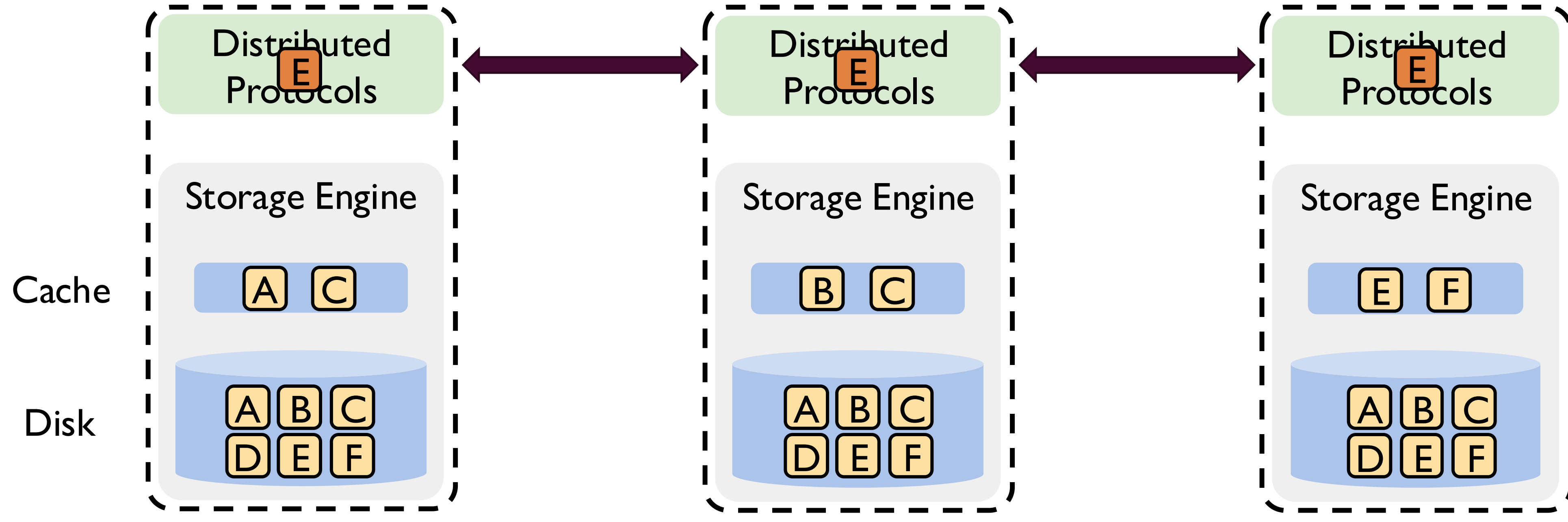
Causes of Cache Redundancy – READs and WRITEs



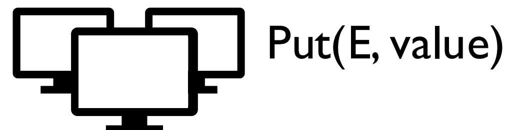


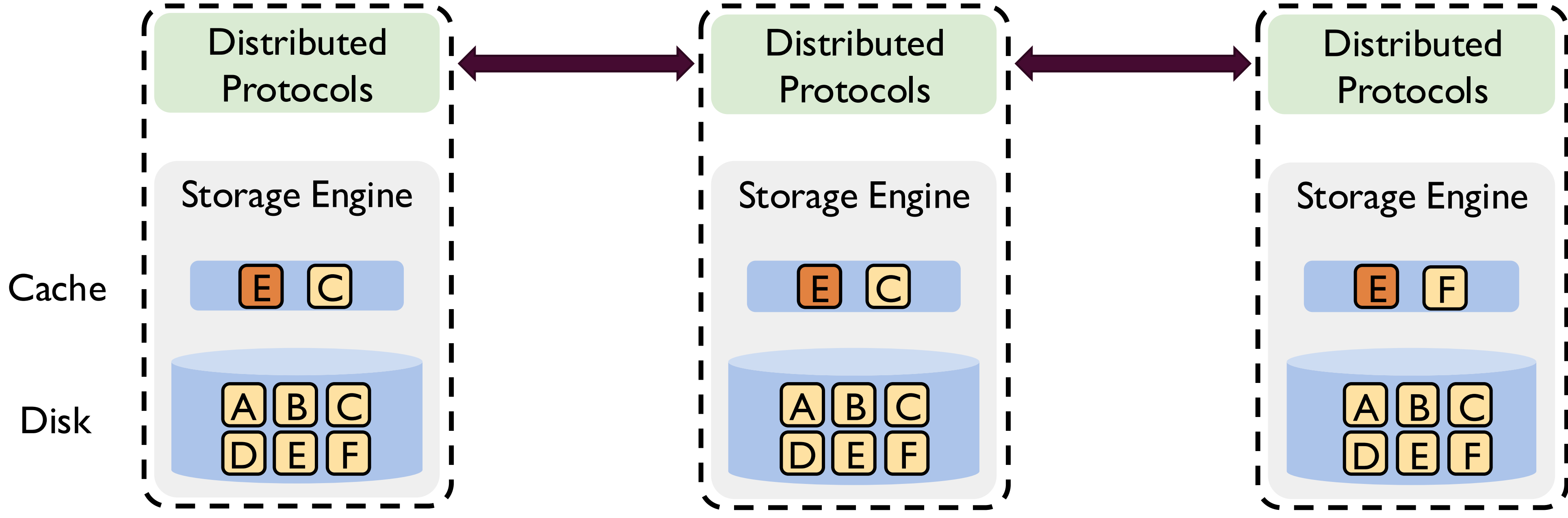
Causes of Cache Redundancy – READs and WRITEs



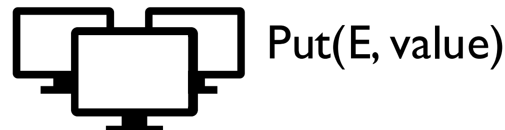


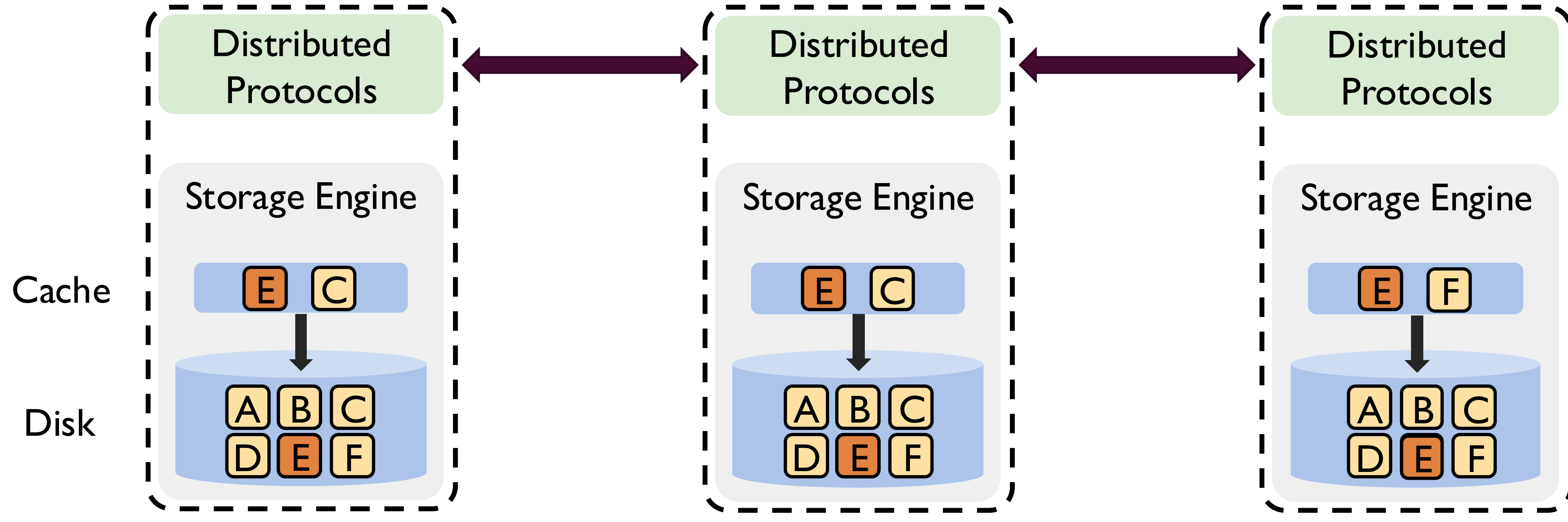
Causes of Cache Redundancy – READs and WRITEs



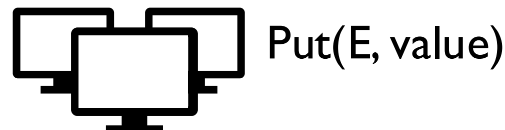


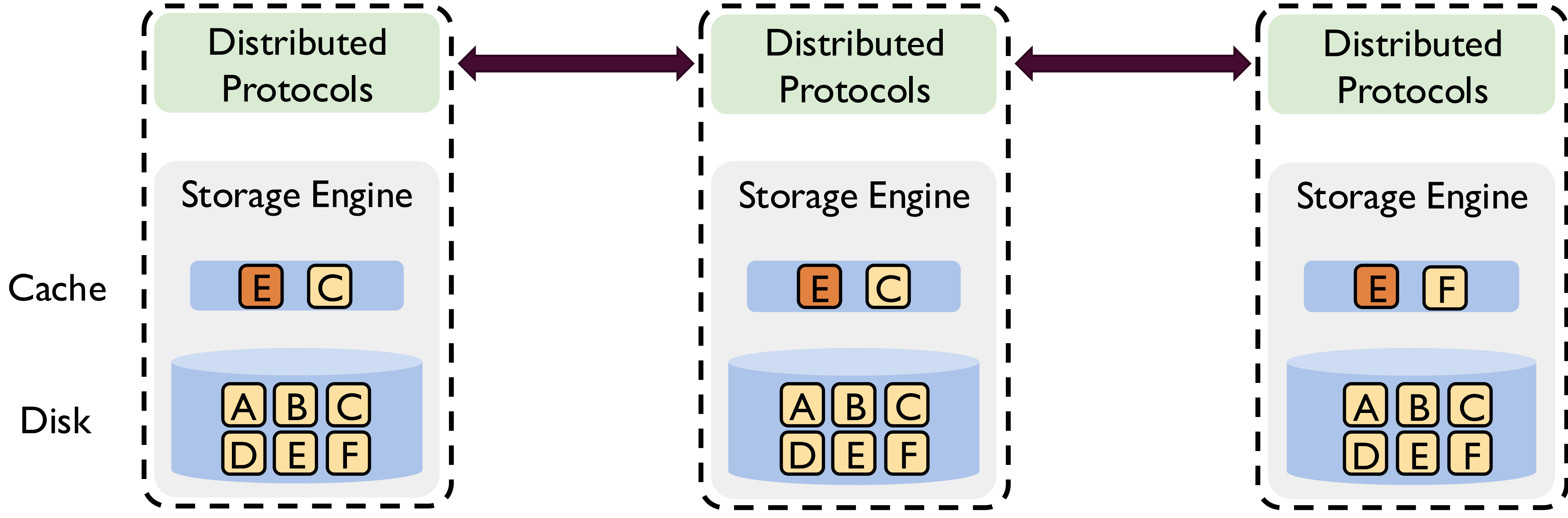
Causes of Cache Redundancy – READs and WRITEs



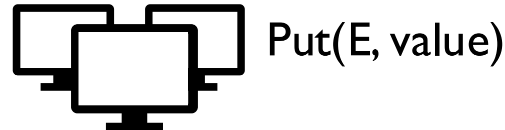


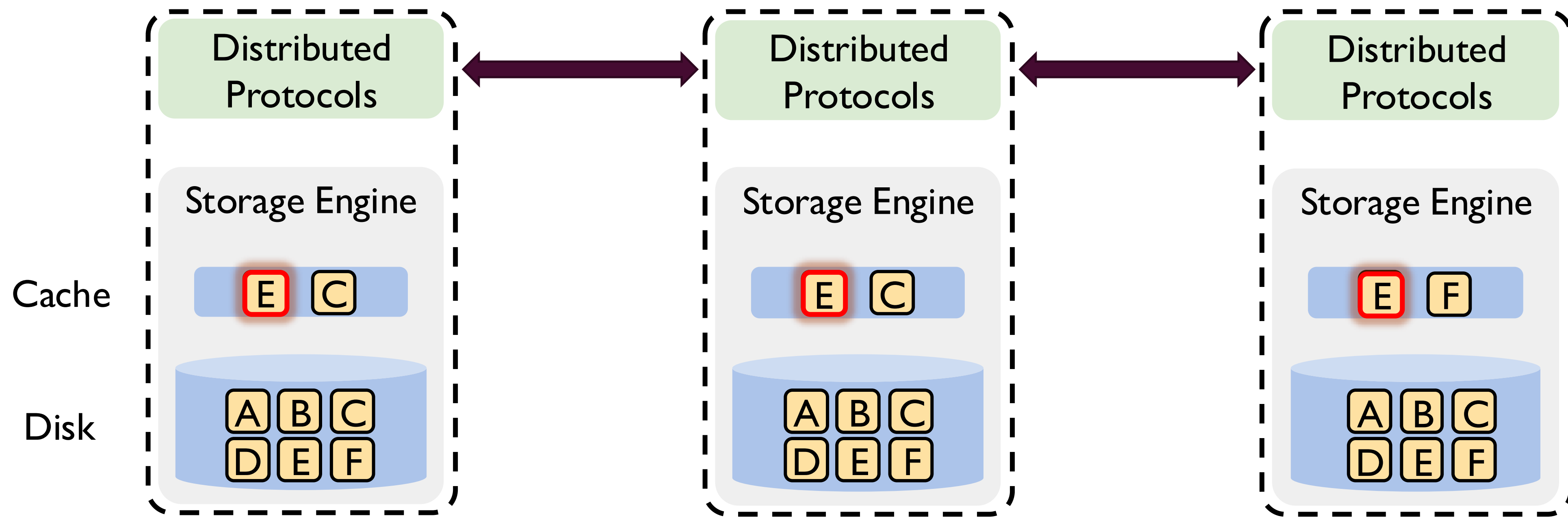
Causes of Cache Redundancy – READs and WRITEs



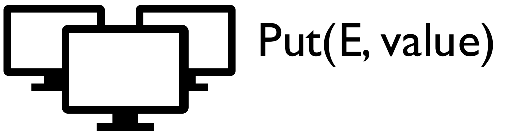


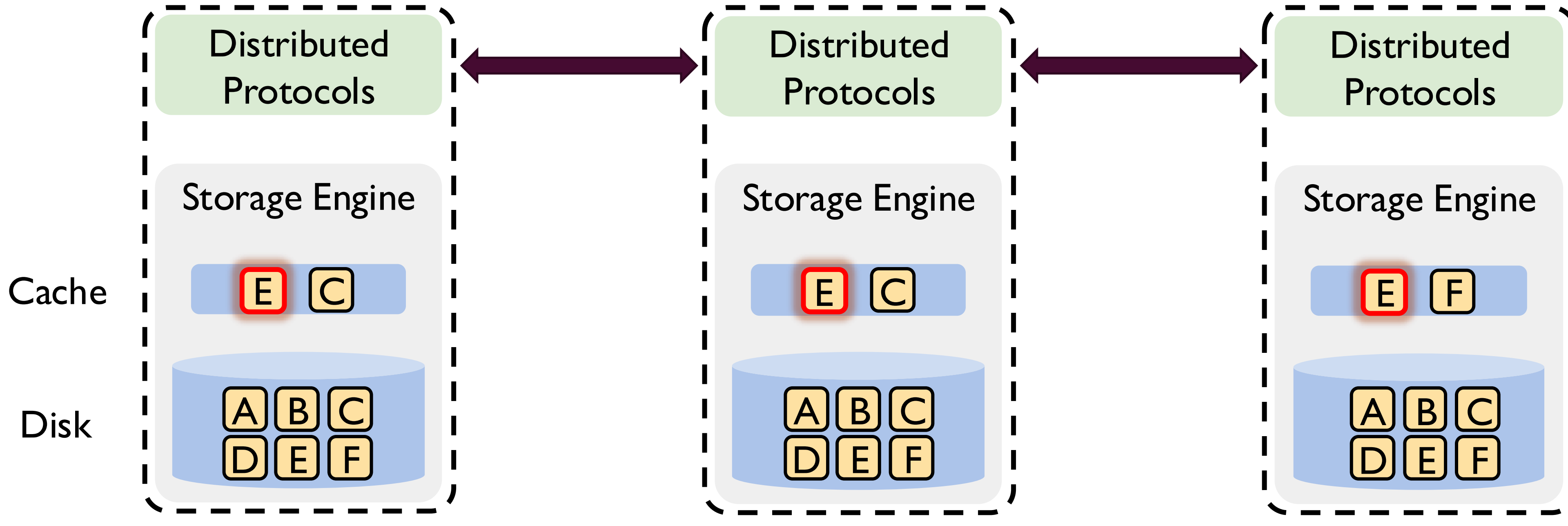
Causes of Cache Redundancy – READs and WRITEs





Causes of Cache Redundancy – READs and WRITEs





Write-Induced Redundancy

Study of Real-World Systems

	Replication Protocol	Local Storage Engine	Caching Granularity
RethinkDB	Raft	Custom B-tree	Page
rqlite	Raft	SQLite B-tree	Page
SPLINTERDB	Raft	B^{ε} - Tree	Page
APACHE HBASE	Read Replicas	LSM	Block
Apache CASSANDRA	Quorums	LSM	Object (row)

Study of Real-World Systems

	Replication Protocol	Local Storage Engine	Caching Granularity
RethinkDB	Raft	Custom B-tree	Page
rqlite	Raft	SQLite B-tree	Page
SPLINTERDB	Raft	B^{ε} - Tree	Page
APACHE HBASE	Read Replicas	LSM	Block
Apache CASSANDRA	Quorums	LSM	Object (row)

Study of Real-World Systems

	Replication Protocol	Local Storage Engine	Caching Granularity
RethinkDB	Raft	Custom B-tree	Page
rqlite	Raft	SQLite B-tree	Page
SPLINTERDB	Raft	B^{ε} - Tree	Page
APACHE HBASE	Read Replicas	LSM	Block
Apache CASSANDRA	Quorums	LSM	Object (row)

Study of Real-World Systems

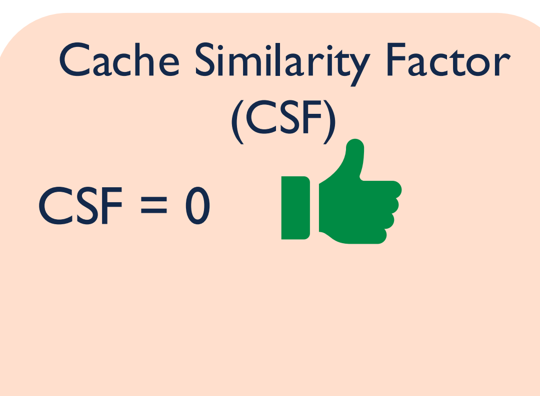
	Replication Protocol	Local Storage Engine	Caching Granularity
RethinkDB	Raft	Custom B-tree	Page
rqlite	Raft	SQLite B-tree	Page
SPLINTERDB	Raft	B^{ε} - Tree	Page
APACHE HBASE	Read Replicas	LSM	Block
Apache CASSANDRA	Quorums	LSM	Object (row)

Study of Real-World Systems

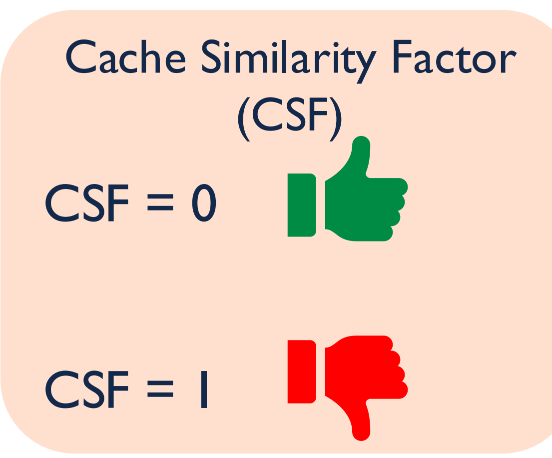
Study of Real-World Systems

Cache Similarity Factor
(CSF)

Study of Real-World Systems



Study of Real-World Systems

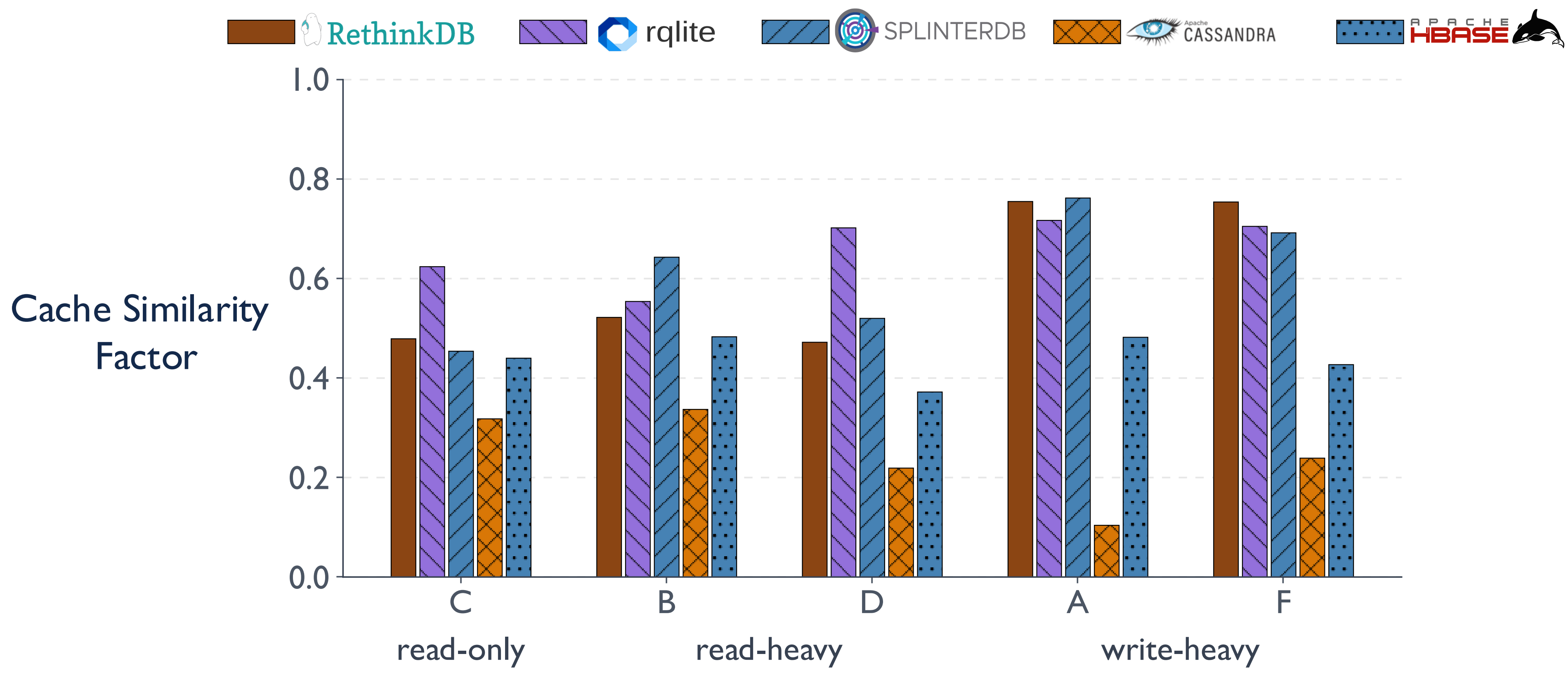


Study of Real-World Systems

Study of Real-World Systems

In RethinkDB, read- and write-induced redundancy cause high similarity across replica caches under varying read-write mixes

Study of Real-World Systems

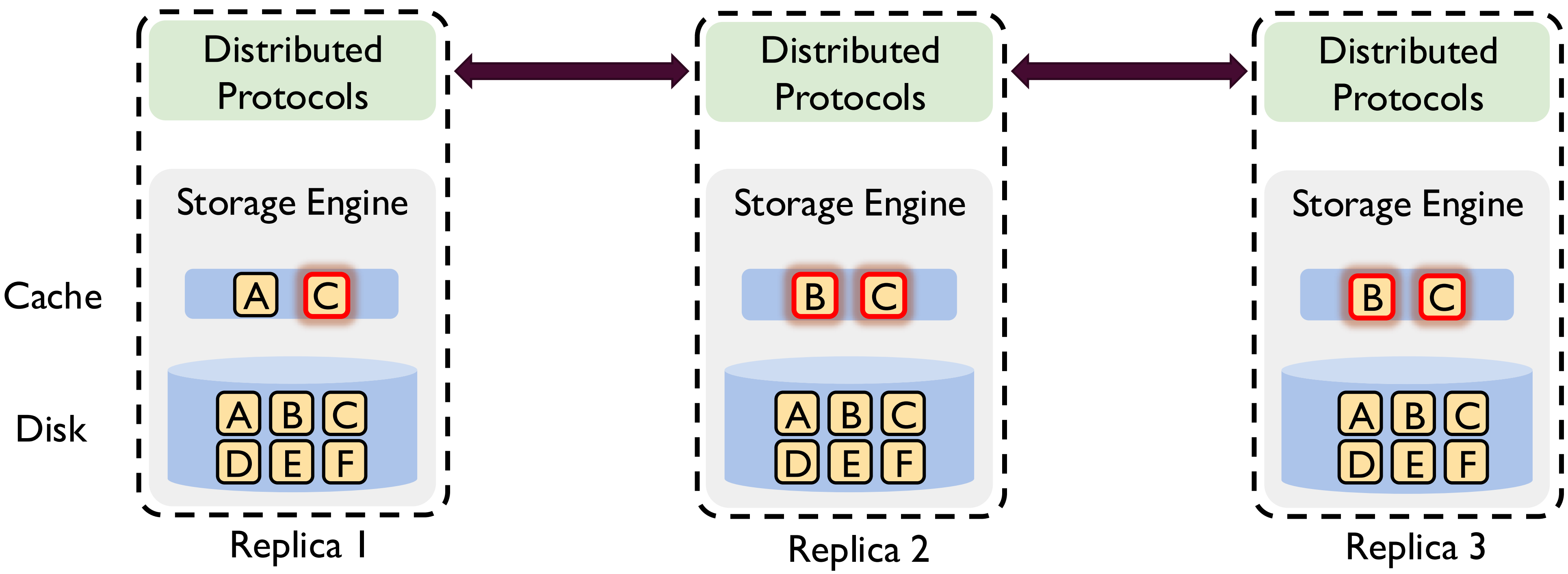


Similarly, all other systems also experience high cache redundancy across workloads

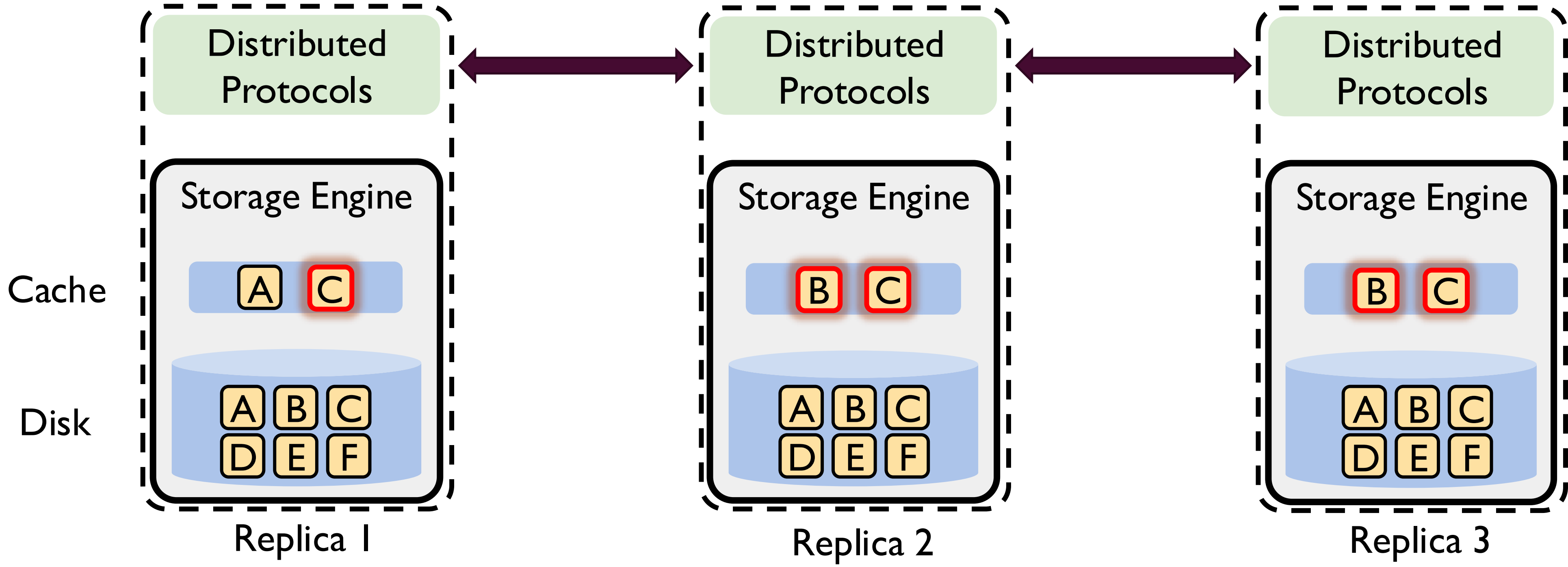
Outline

- Introduction
- Study
- Our Idea - Logically Disaggregated Cache
- Implementations
- Evaluation

Cache Redundancy – Root Cause

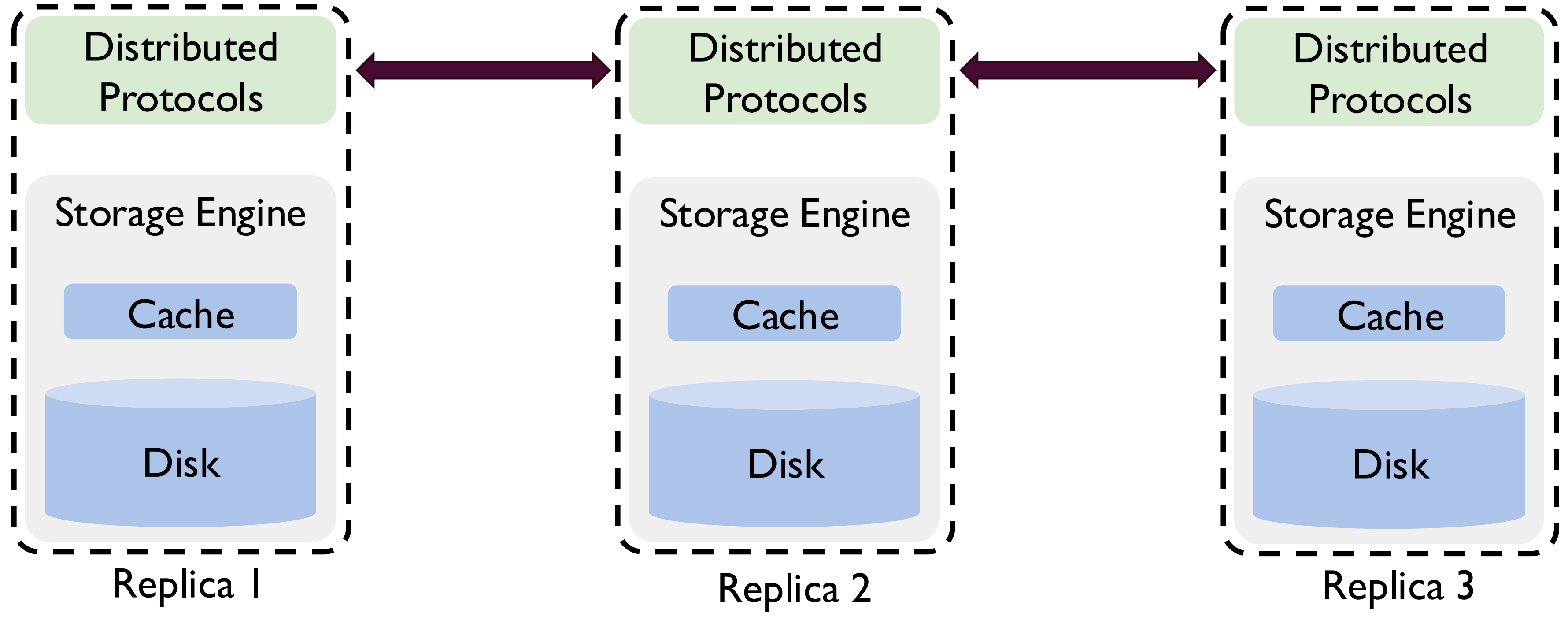


Cache Redundancy – Root Cause

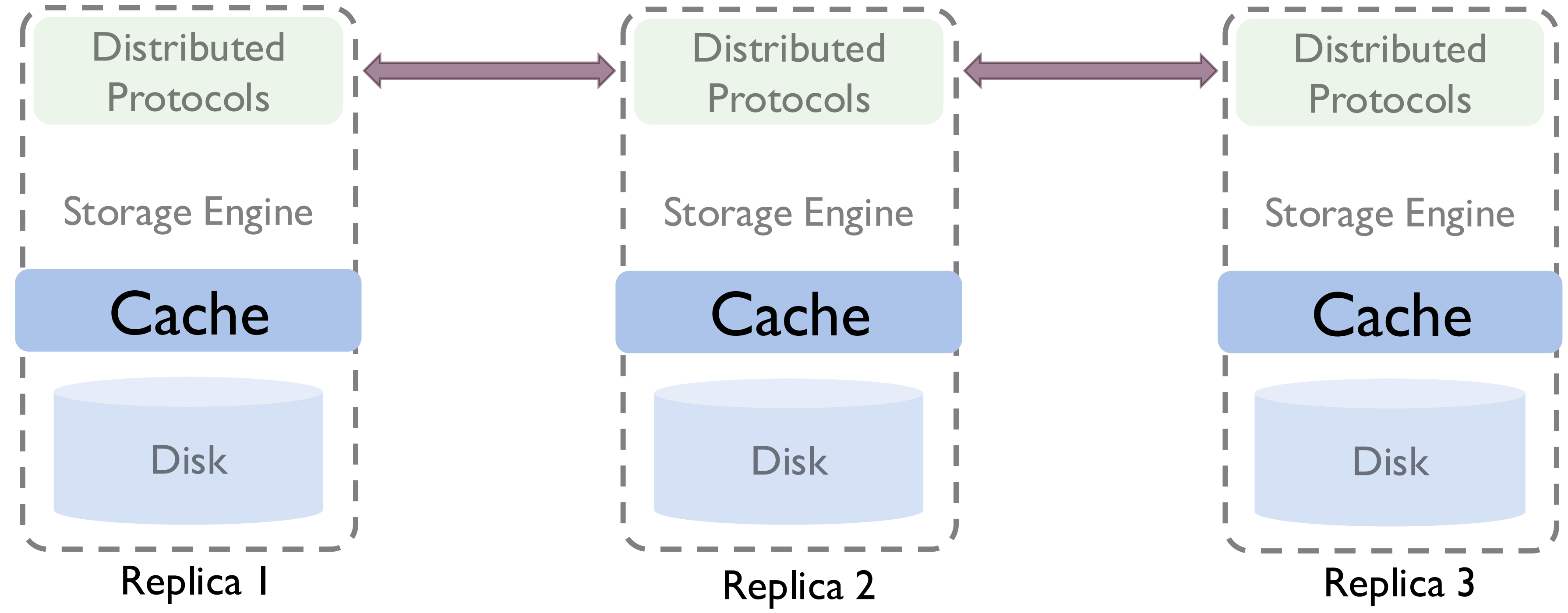


Replicas manage their caches in silos, unaware of each other

Our Solution Is Logically Disaggregated Caches

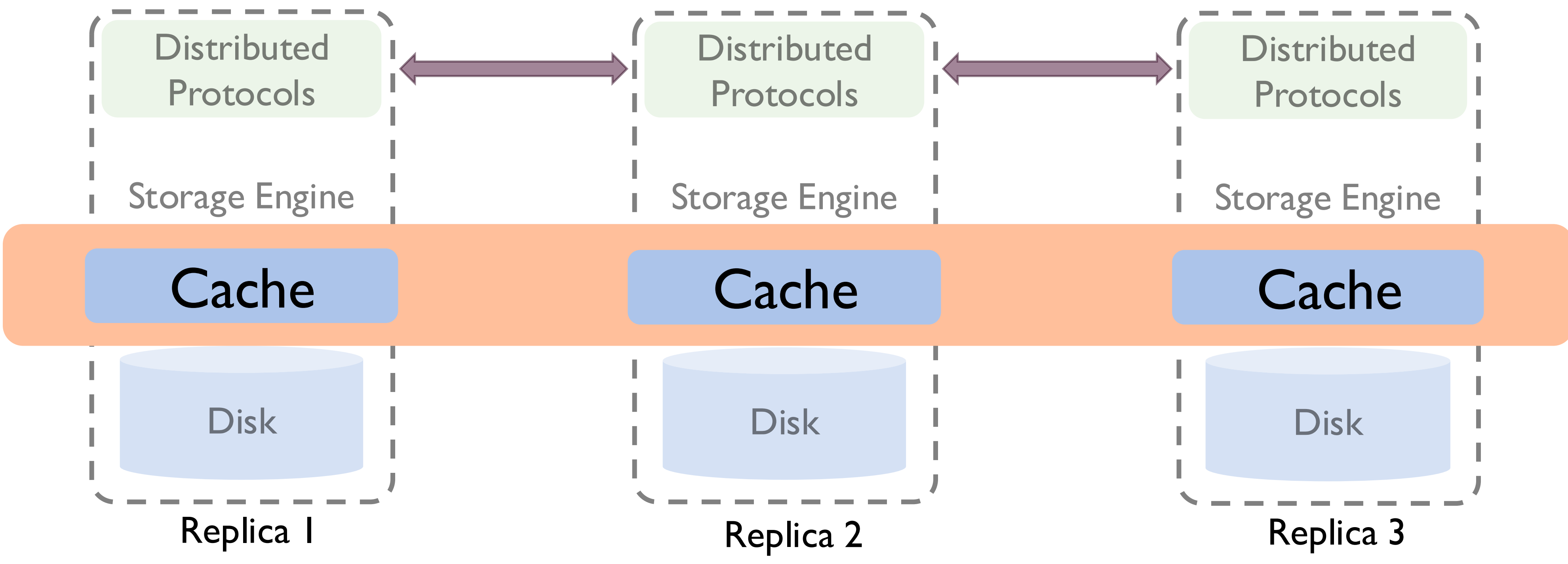


Our Solution Is Logically Disaggregated Caches



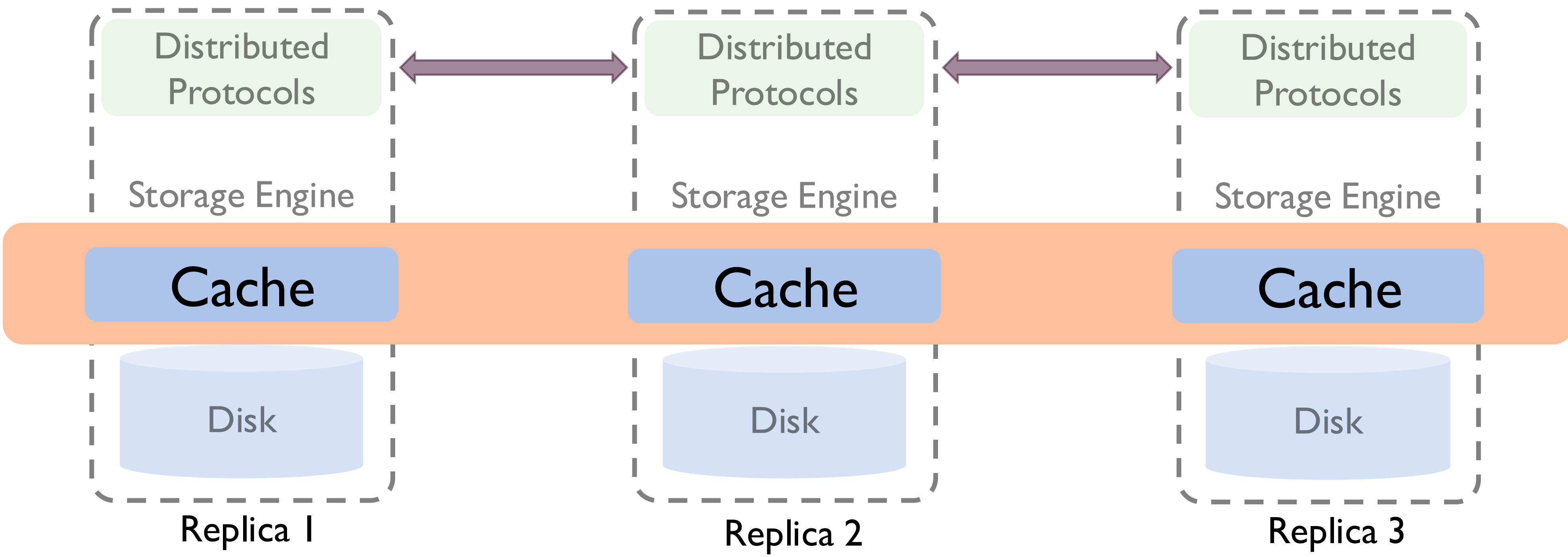
Disaggregate embedded caches from replicas

Our Solution Is Logically Disaggregated Caches



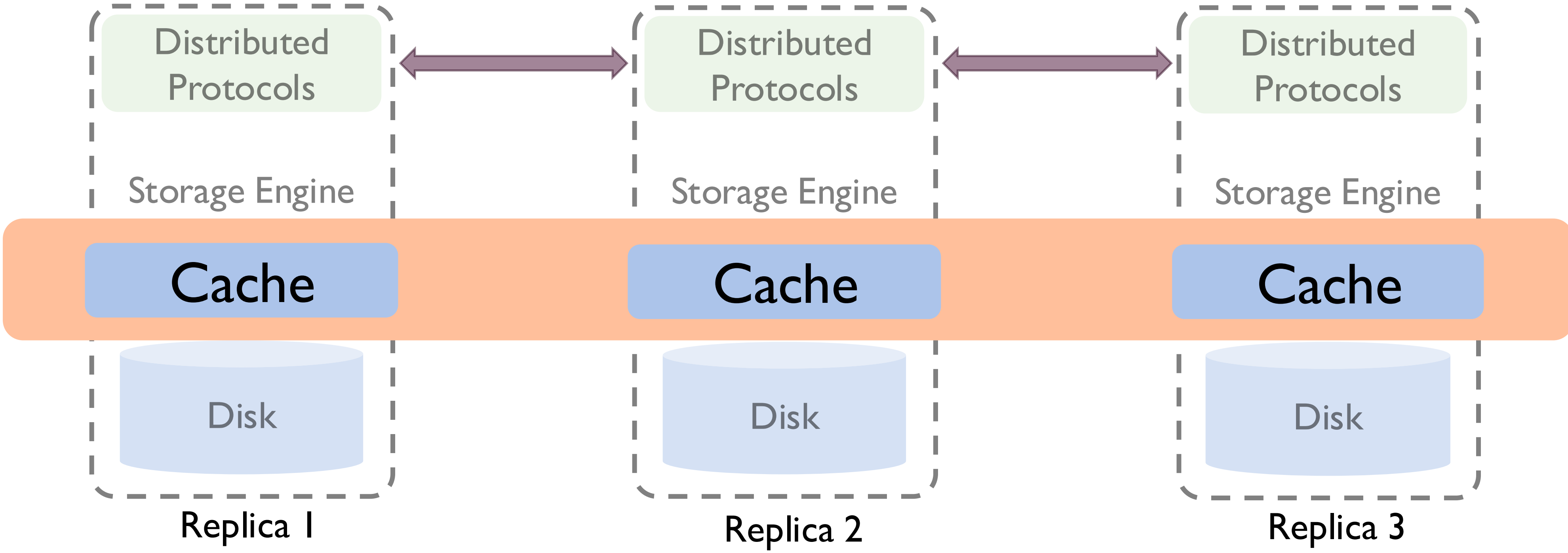
Disaggregate embedded caches from replicas
Manage as a **single, unified, logical** cache

Our Solution Is Logically Disaggregated Caches



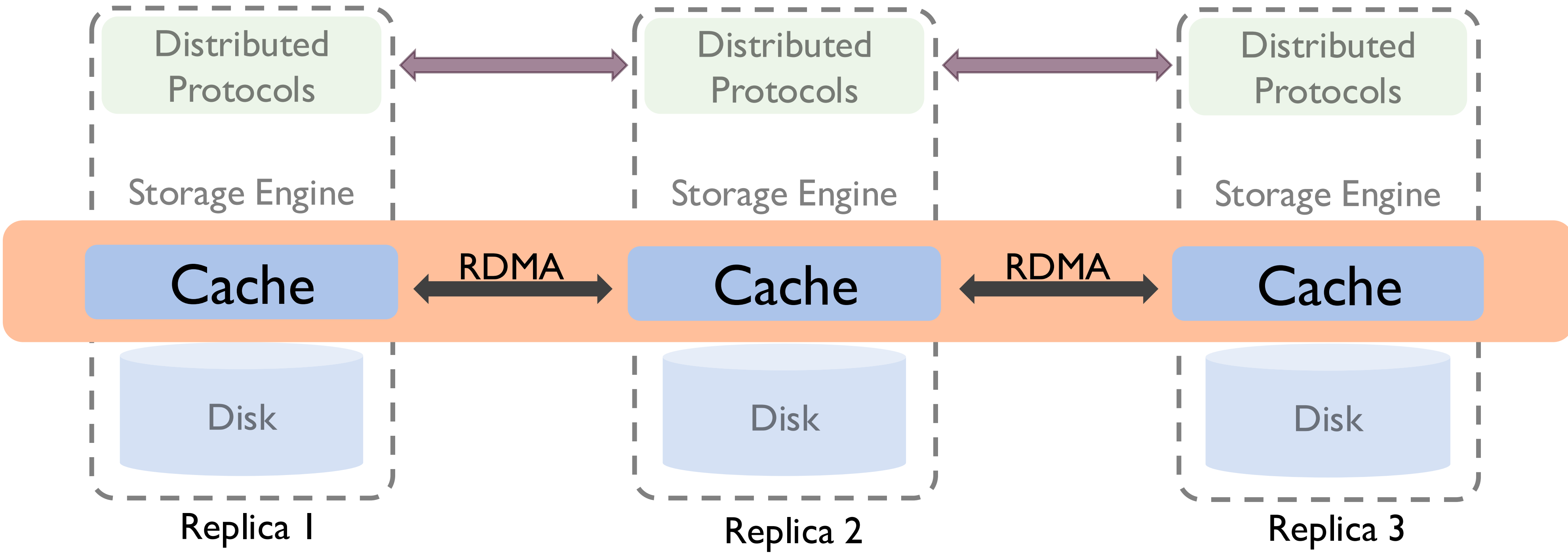
Disaggregate embedded caches from replicas
Manage as a **single, unified, logical** cache
Caches are **aware** of each other — no more silos

Our Solution Is Logically Disaggregated Caches



Disaggregate embedded caches from replicas
Manage as a **single, unified, logical** cache
Caches are **aware** of each other — no more silos
Any replica can access **any part** of the logical cache

Our Solution Is Logically Disaggregated Caches



Disaggregate embedded caches from replicas
Manage as a **single, unified, logical** cache
Caches are **aware** of each other — no more silos
Any replica can access **any part** of the logical cache

LDC Realizes Its Architecture With Three Salient Features

LDC Realizes Its Architecture With Three Salient Features

Remote Cache Access

avoids read-induced redundancy

LDC Realizes Its Architecture With Three Salient Features

Remote Cache Access

avoids read-induced redundancy

Selective Quick Demotion

avoids write-induced redundancy

LDC Realizes Its Architecture With Three Salient Features

Remote Cache Access

avoids read-induced redundancy

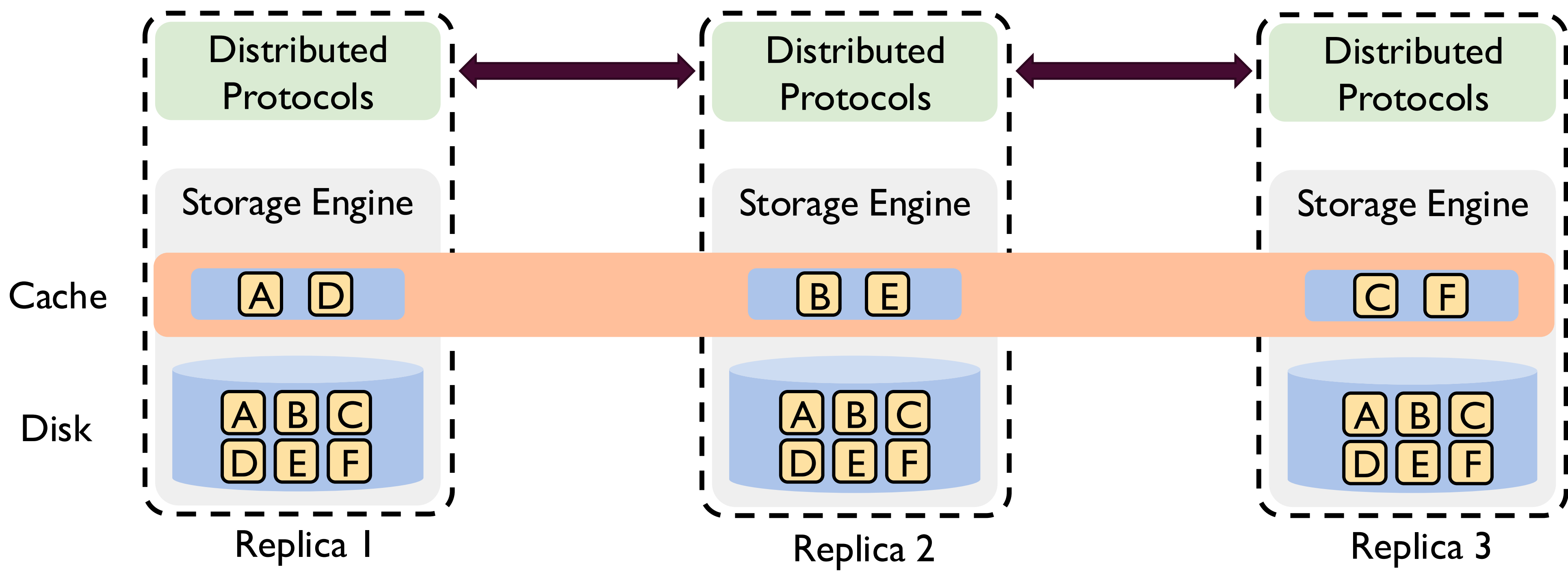
Selective Quick Demotion

avoids write-induced redundancy

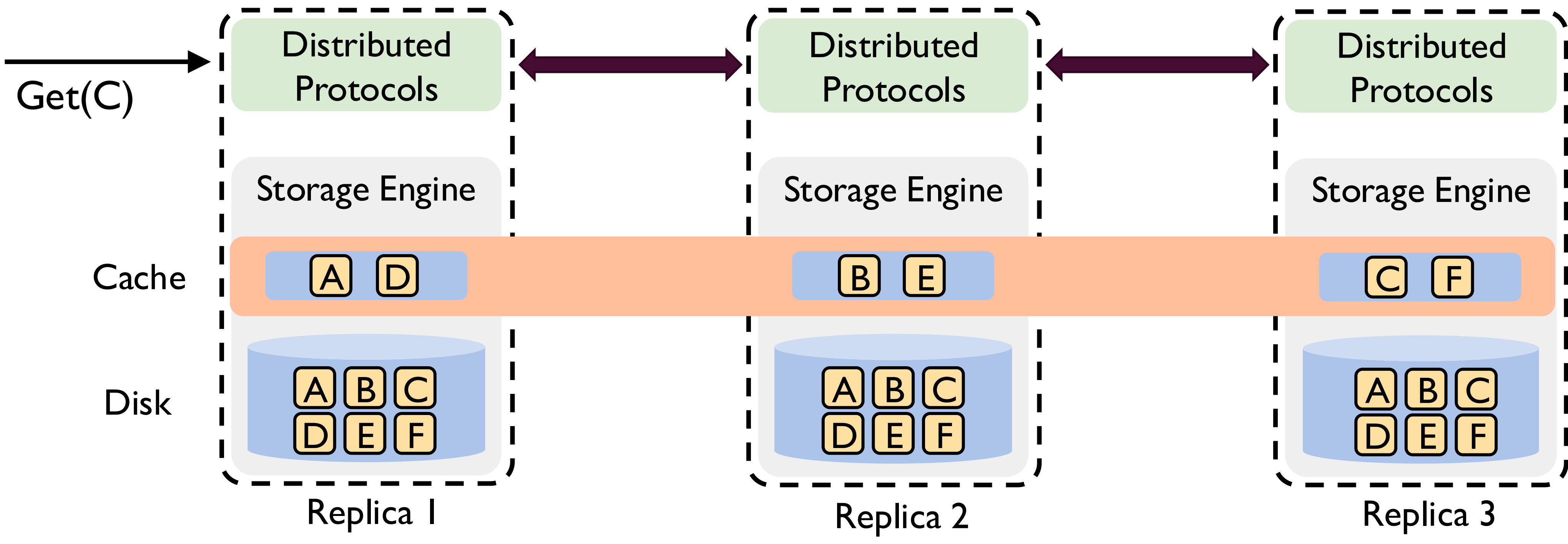
Cost-Benefit Analyzer

admits into cache only when beneficial

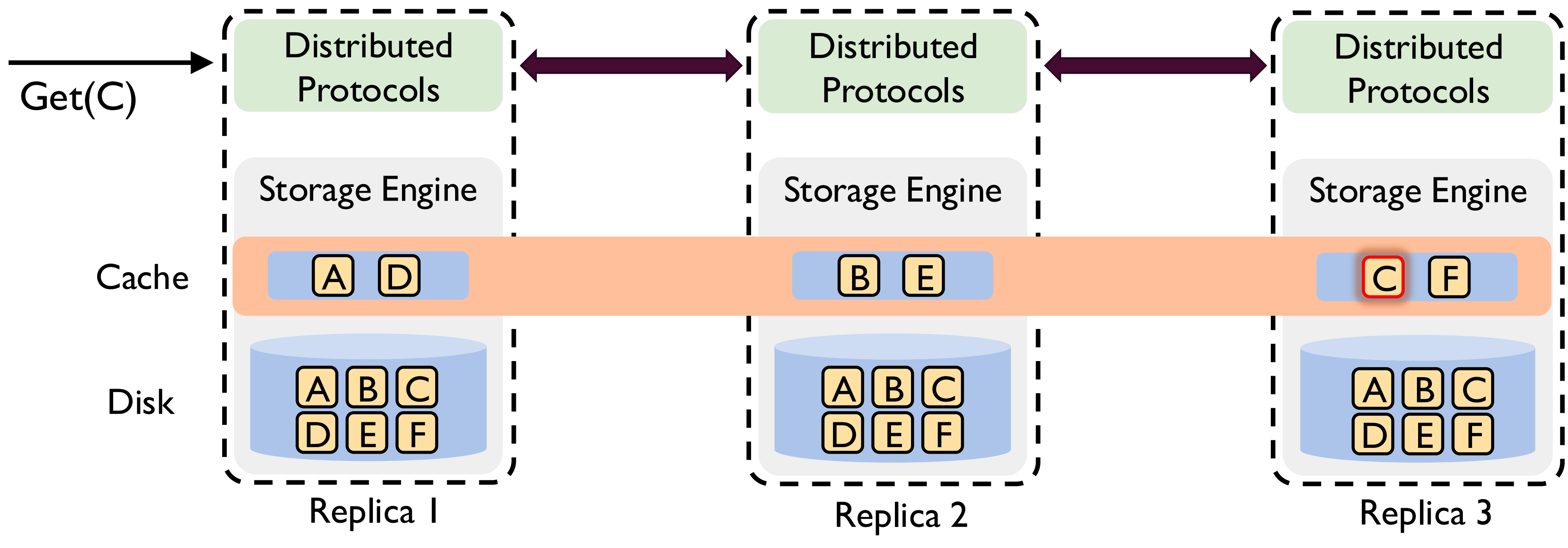
Avoiding Read-Induced Redundancy via Remote Cache Access



Avoiding Read-Induced Redundancy via Remote Cache Access

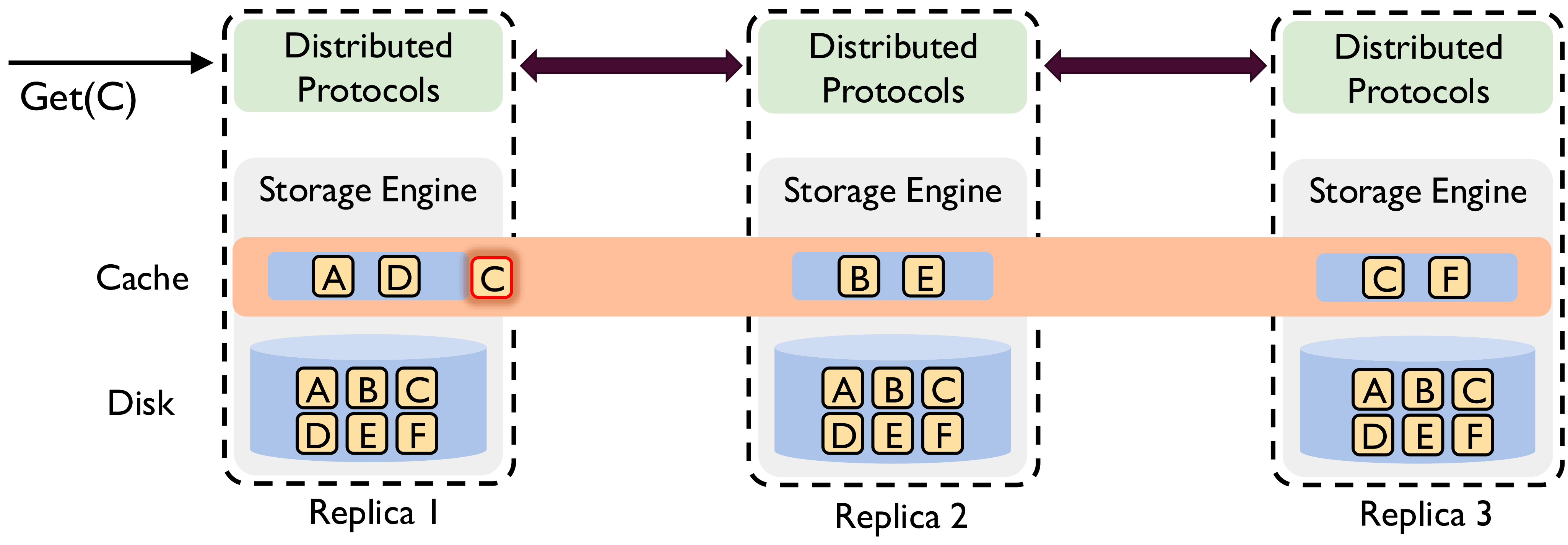


Avoiding Read-Induced Redundancy via Remote Cache Access



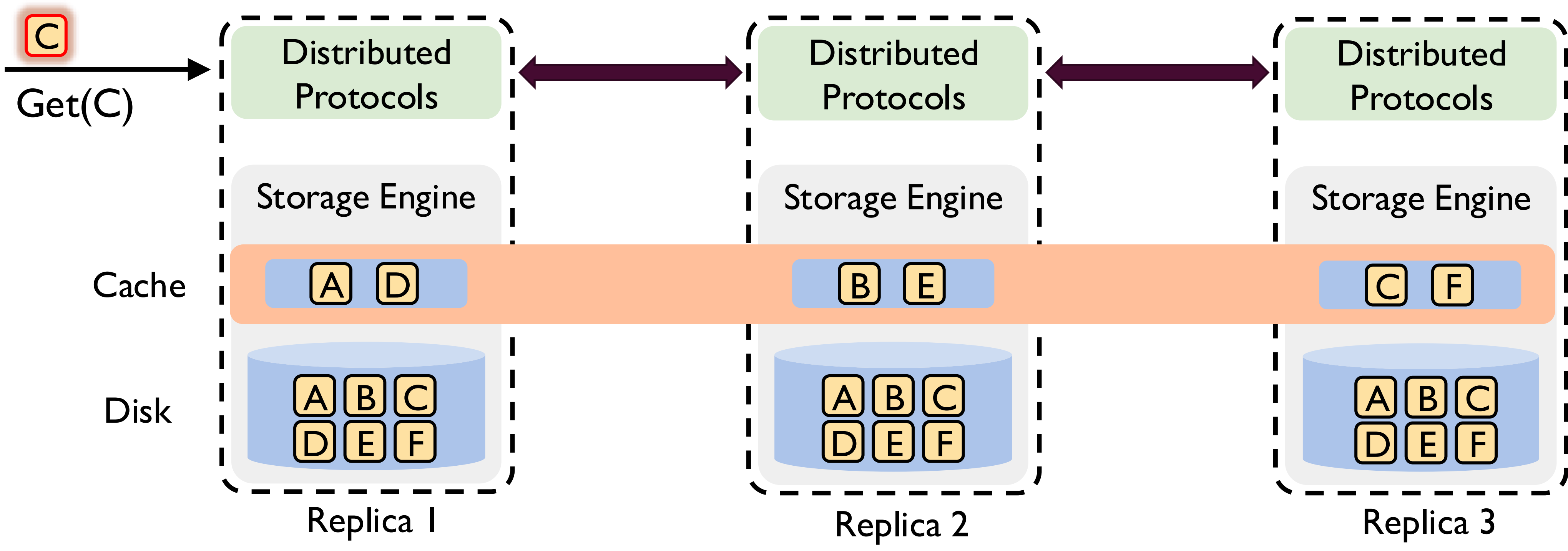
If an object is cached on another replica

Avoiding Read-Induced Redundancy via Remote Cache Access



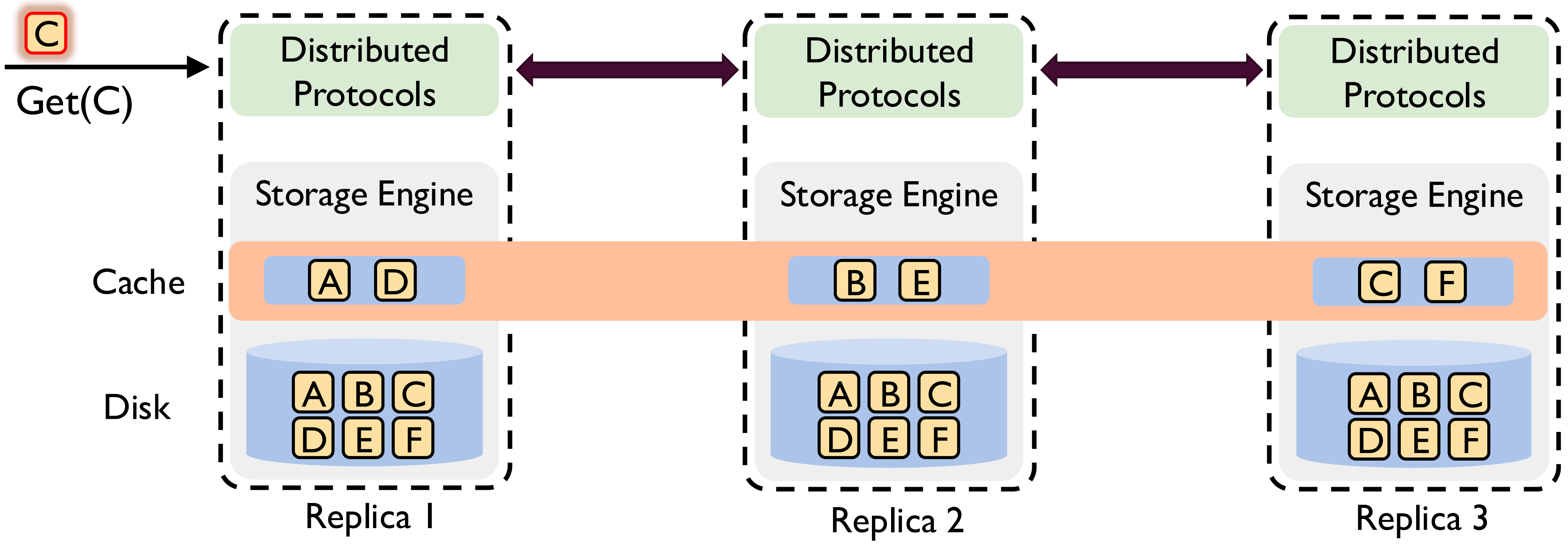
If an object is cached on another replica → **fetch it remotely**

Avoiding Read-Induced Redundancy via Remote Cache Access



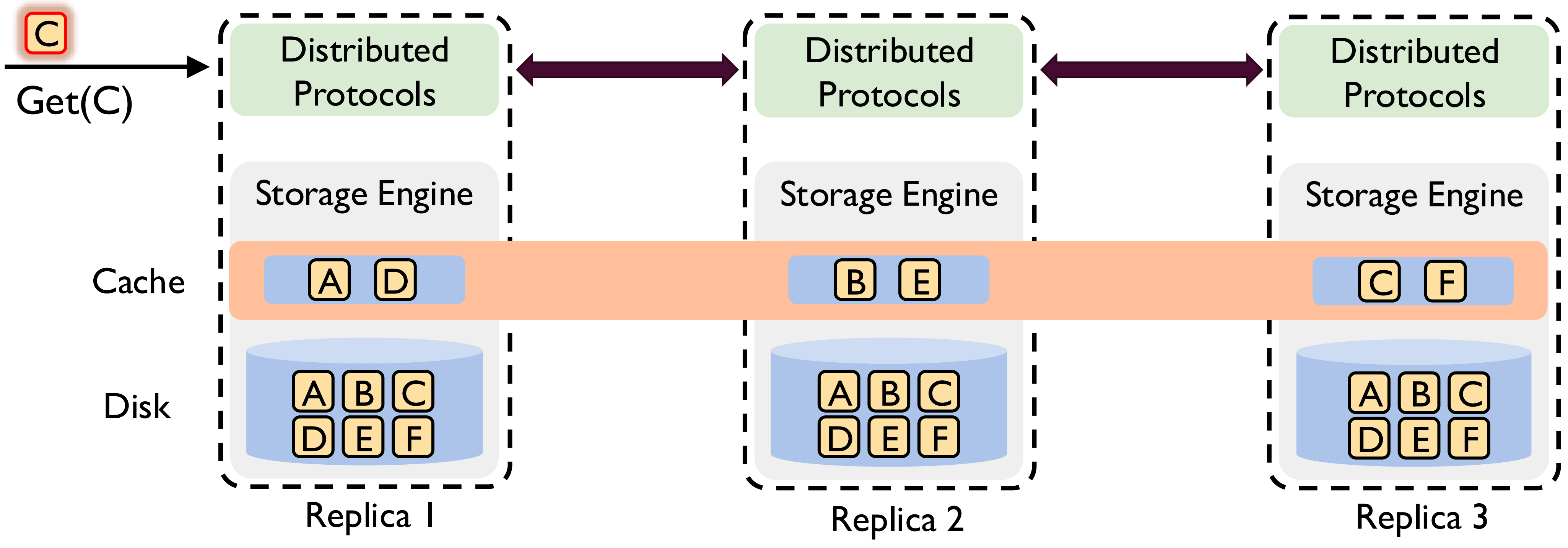
If an object is cached on another replica → **fetch it remotely**
Serve the client without admitting it locally

Avoiding Read-Induced Redundancy via Remote Cache Access



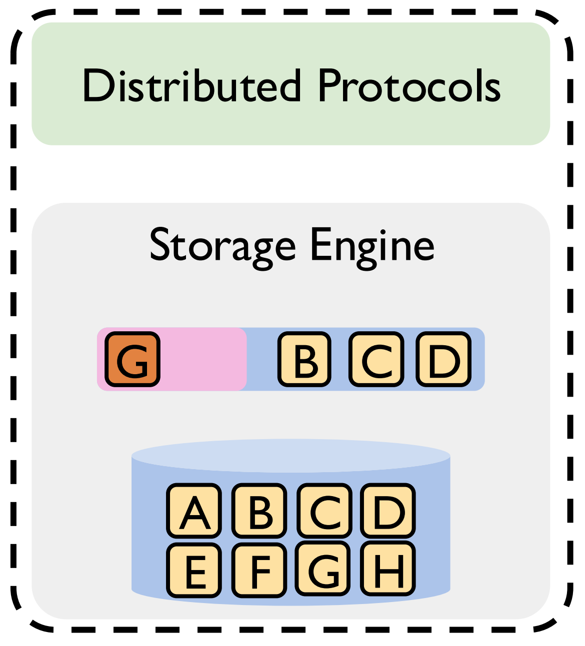
If an object is cached on another replica → **fetch it remotely**
Serve the client without admitting it locally
Skip the disk, avoid the duplicate

Avoiding Read-Induced Redundancy via Remote Cache Access

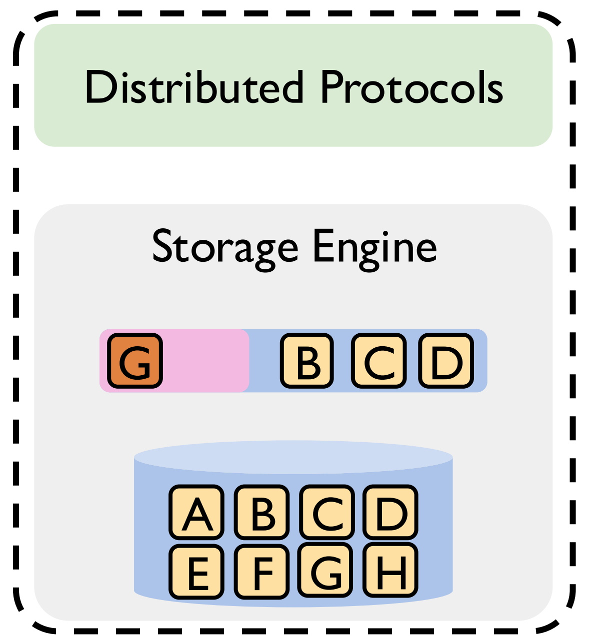


If an object is cached on another replica → **fetch it remotely**
Serve the client without admitting it locally
Skip the disk, avoid the duplicate
Done efficiently via one-sided **RDMA**

Avoiding Write-Induced Redundancy via Selective Quick Demotions

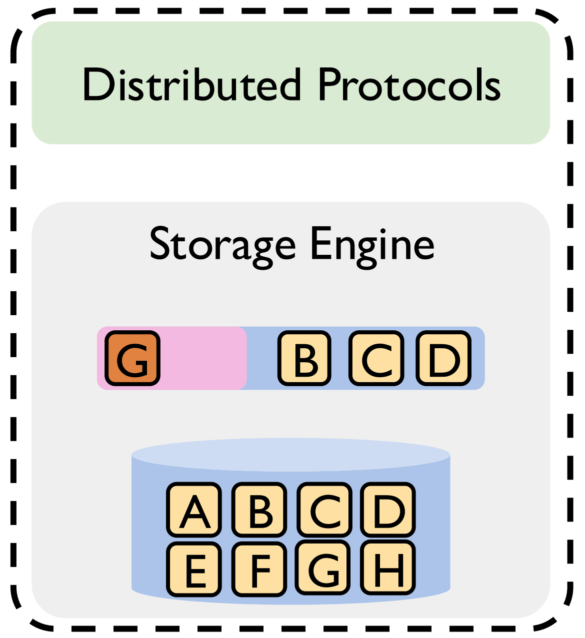


Avoiding Write-Induced Redundancy via Selective Quick Demotions



Writes must be applied at every replica → same object lands in every cache

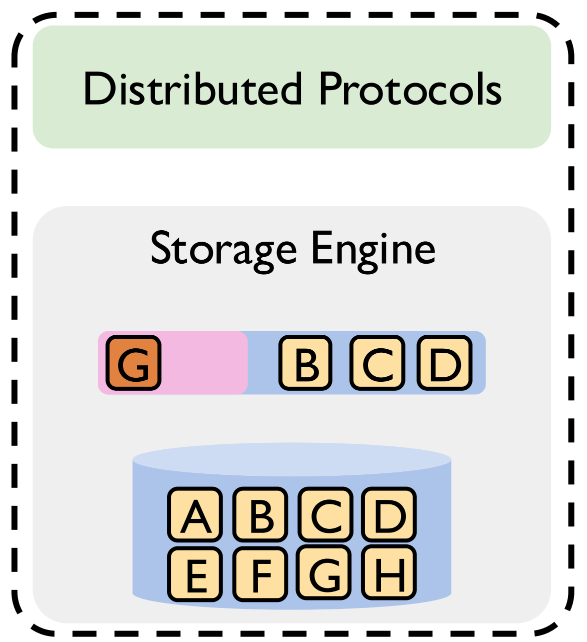
Avoiding Write-Induced Redundancy via Selective Quick Demotions



Writes must be applied at every replica → same object lands in every cache

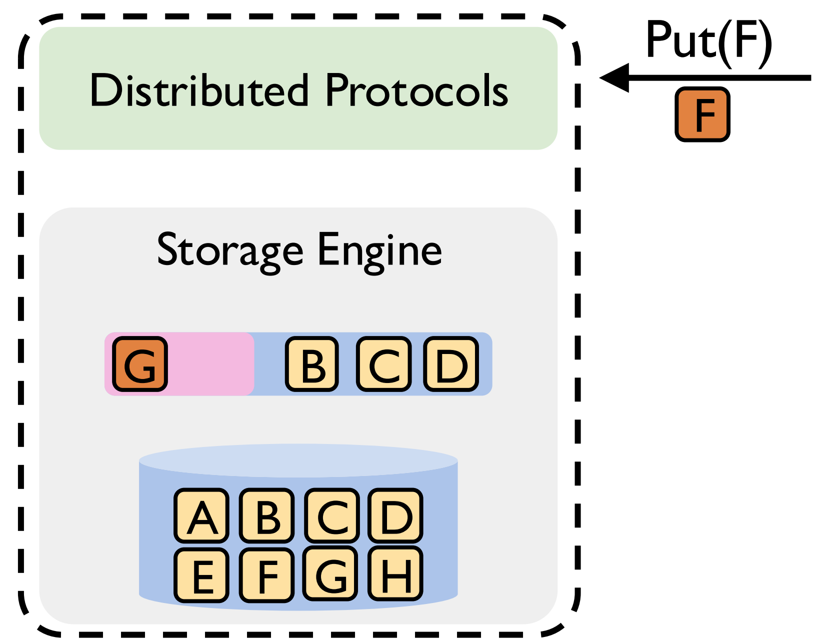
Every replica is aware writes would pollute every cache.

Avoiding Write-Induced Redundancy via Selective Quick Demotions



Writes must be applied at every replica → same object lands in every cache
Every replica is aware writes would pollute every cache.
So **quickly demote** written objects.

Avoiding Write-Induced Redundancy via Selective Quick Demotions

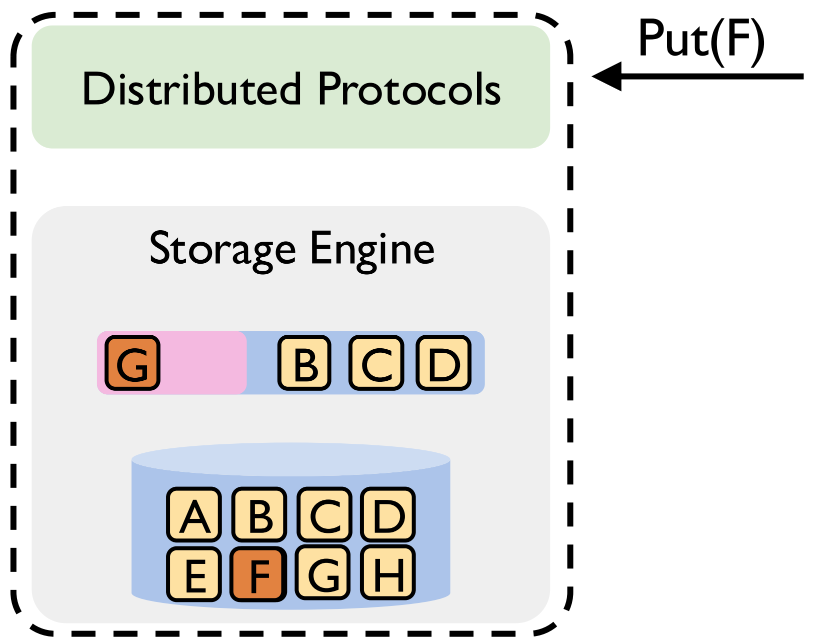


Writes must be applied at every replica → same object lands in every cache

Every replica is aware writes would pollute every cache.

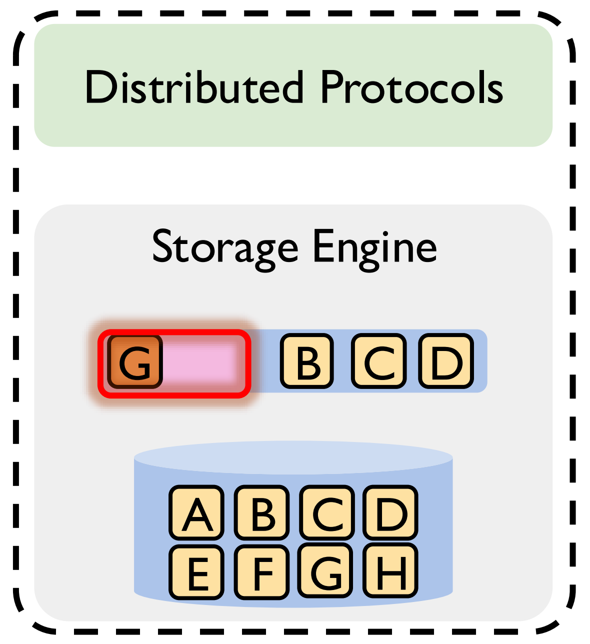
So **quickly demote** written objects.

Avoiding Write-Induced Redundancy via Selective Quick Demotions



Writes must be applied at every replica → same object lands in every cache
Every replica is aware writes would pollute every cache.
So **quickly demote** written objects.
Bypassing the cache kills write performance, no batching, no coalescing

Avoiding Write-Induced Redundancy via Selective Quick Demotions



Writes must be applied at every replica → same object lands in every cache

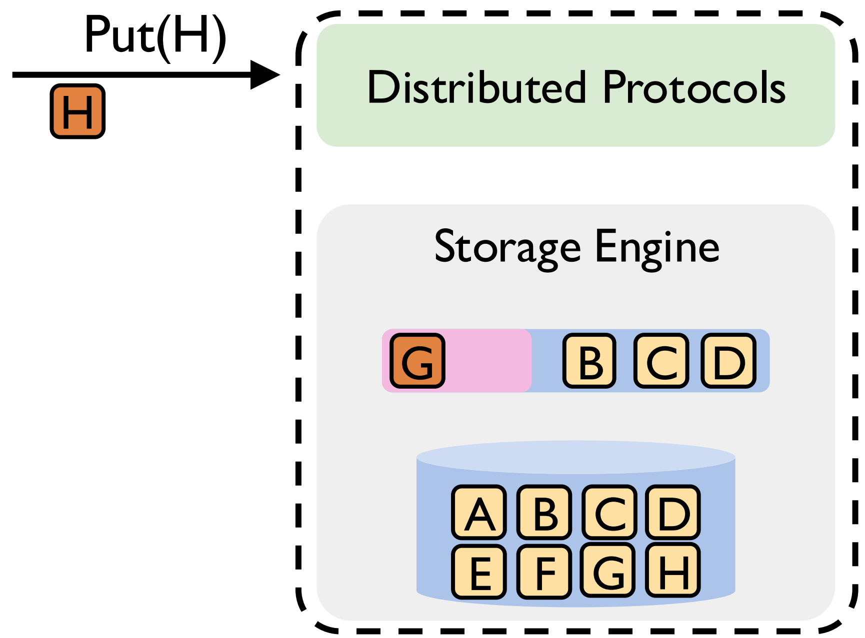
Every replica is aware writes would pollute every cache.

So **quickly demote** written objects.

Bypassing the cache kills write performance, no batching, no coalescing

Implemented via a **tiny queue** (0.01% of cache), absorbs writes, flushes to disk lazily

Avoiding Write-Induced Redundancy via Selective Quick Demotions



Writes must be applied at every replica → same object lands in every cache

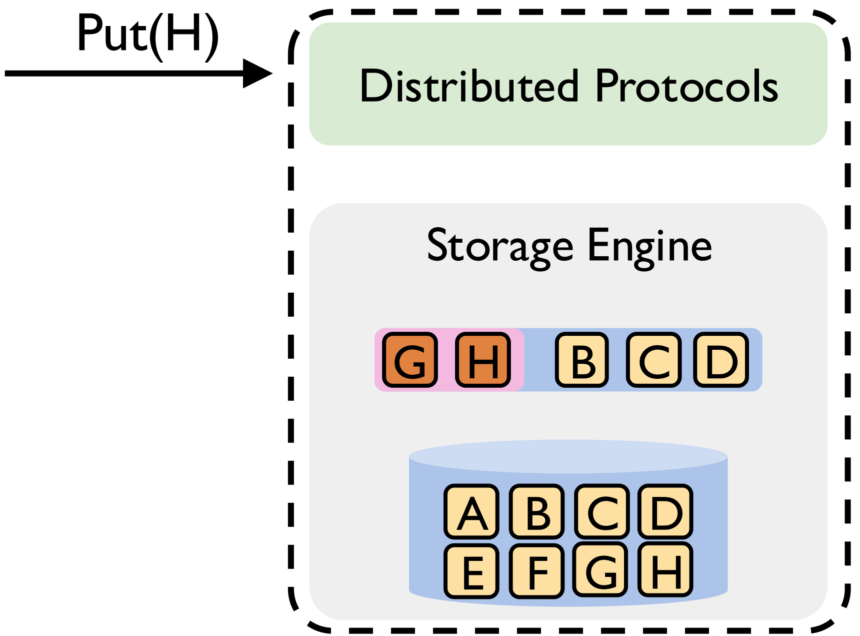
Every replica is aware writes would pollute every cache.

So **quickly demote** written objects.

Bypassing the cache kills write performance, no batching, no coalescing

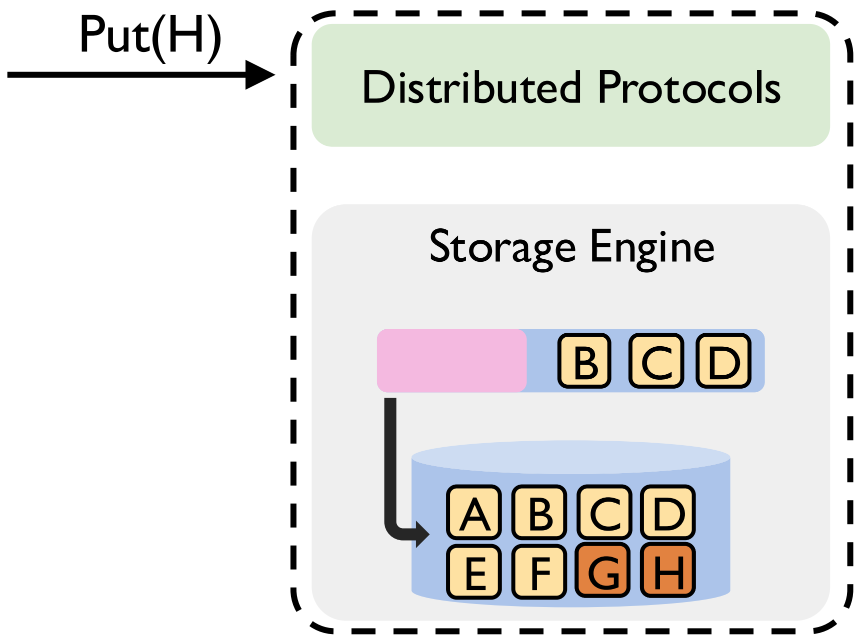
Implemented via a **tiny queue** (0.01% of cache), absorbs writes, flushes to disk lazily

Avoiding Write-Induced Redundancy via Selective Quick Demotions



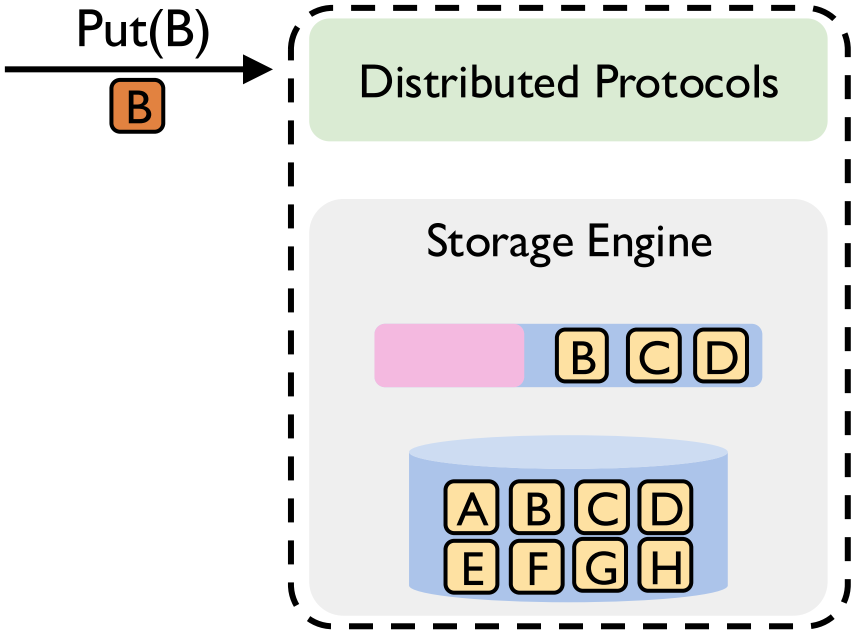
Writes must be applied at every replica → same object lands in every cache
Every replica is aware writes would pollute every cache.
So **quickly demote** written objects.
Bypassing the cache kills write performance, no batching, no coalescing
Implemented via a **tiny queue** (0.01% of cache), absorbs writes, flushes to disk lazily

Avoiding Write-Induced Redundancy via Selective Quick Demotions



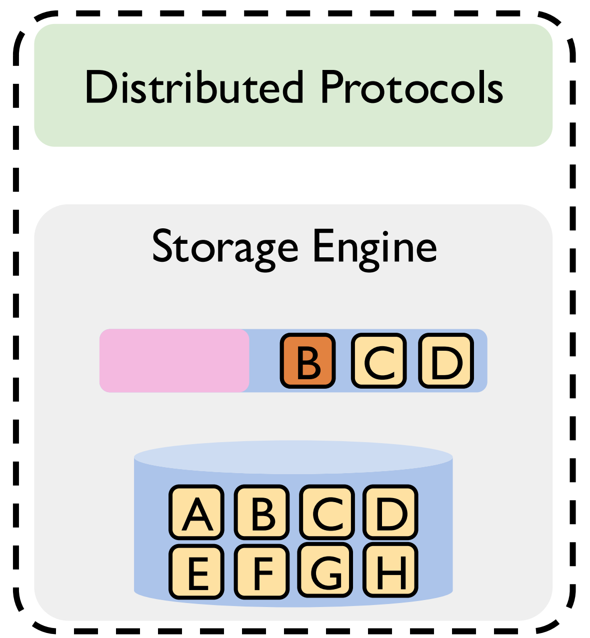
Writes must be applied at every replica → same object lands in every cache
Every replica is aware writes would pollute every cache.
So **quickly demote** written objects.
Bypassing the cache kills write performance, no batching, no coalescing
Implemented via a **tiny queue** (0.01% of cache), absorbs writes, flushes to disk lazily

Avoiding Write-Induced Redundancy via Selective Quick Demotions



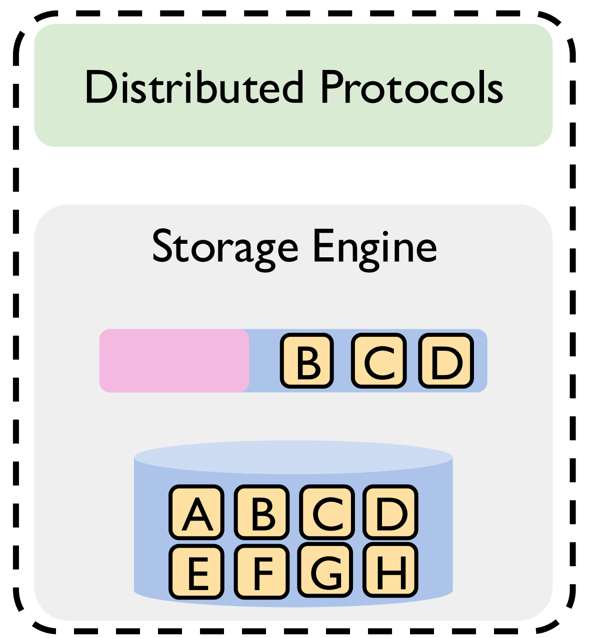
Writes must be applied at every replica → same object lands in every cache
Every replica is aware writes would pollute every cache.
So **quickly demote** written objects.
Bypassing the cache kills write performance, no batching, no coalescing
Implemented via a **tiny queue** (0.01% of cache), absorbs writes, flushes to disk lazily
Selective: if already in main cache, just update in place, no new redundancy

Avoiding Write-Induced Redundancy via Selective Quick Demotions



Writes must be applied at every replica → same object lands in every cache
Every replica is aware writes would pollute every cache.
So **quickly demote** written objects.
Bypassing the cache kills write performance, no batching, no coalescing
Implemented via a **tiny queue** (0.01% of cache), absorbs writes, flushes to disk lazily
Selective: if already in main cache, just update in place, no new redundancy

Avoiding Write-Induced Redundancy via Selective Quick Demotions



Writes must be applied at every replica → same object lands in every cache

Every replica is aware writes would pollute every cache.

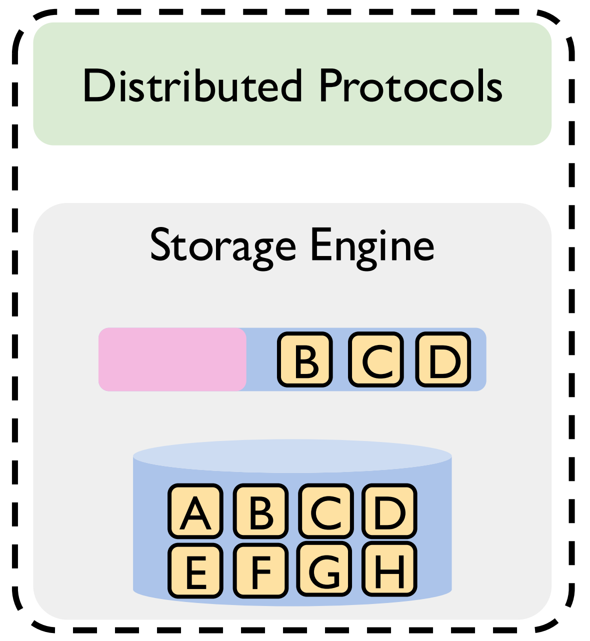
So **quickly demote** written objects.

Bypassing the cache kills write performance, no batching, no coalescing

Implemented via a **tiny queue** (0.01% of cache), absorbs writes, flushes to disk lazily

Selective: if already in main cache, just update in place, no new redundancy

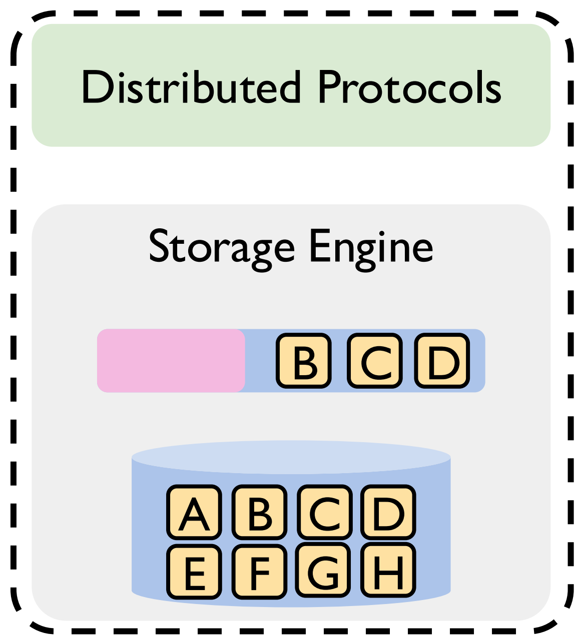
Avoiding Write-Induced Redundancy via Selective Quick Demotions



Write-induced redundancy **bounded** to the tiny queue — much smaller than the main cache

Writes must be applied at every replica → same object lands in every cache
Every replica is aware writes would pollute every cache.
So **quickly demote** written objects.
Bypassing the cache kills write performance, no batching, no coalescing
Implemented via a **tiny queue** (0.01% of cache), absorbs writes, flushes to disk lazily
Selective: if already in main cache, just update in place, no new redundancy

Avoiding Write-Induced Redundancy via Selective Quick Demotions

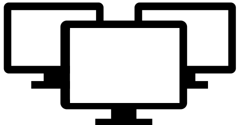


Write-induced redundancy **bounded** to the tiny queue — much smaller than the main cache

Write performance stays **close** to the original

Writes must be applied at every replica → same object lands in every cache
Every replica is aware writes would pollute every cache.
So **quickly demote** written objects.
Bypassing the cache kills write performance, no batching, no coalescing
Implemented via a **tiny queue** (0.01% of cache), absorbs writes, flushes to disk lazily
Selective: if already in main cache, just update in place, no new redundancy

LDC Read and Write Path



Distributed Protocols | Distributed Protocols | Distributed Protocols

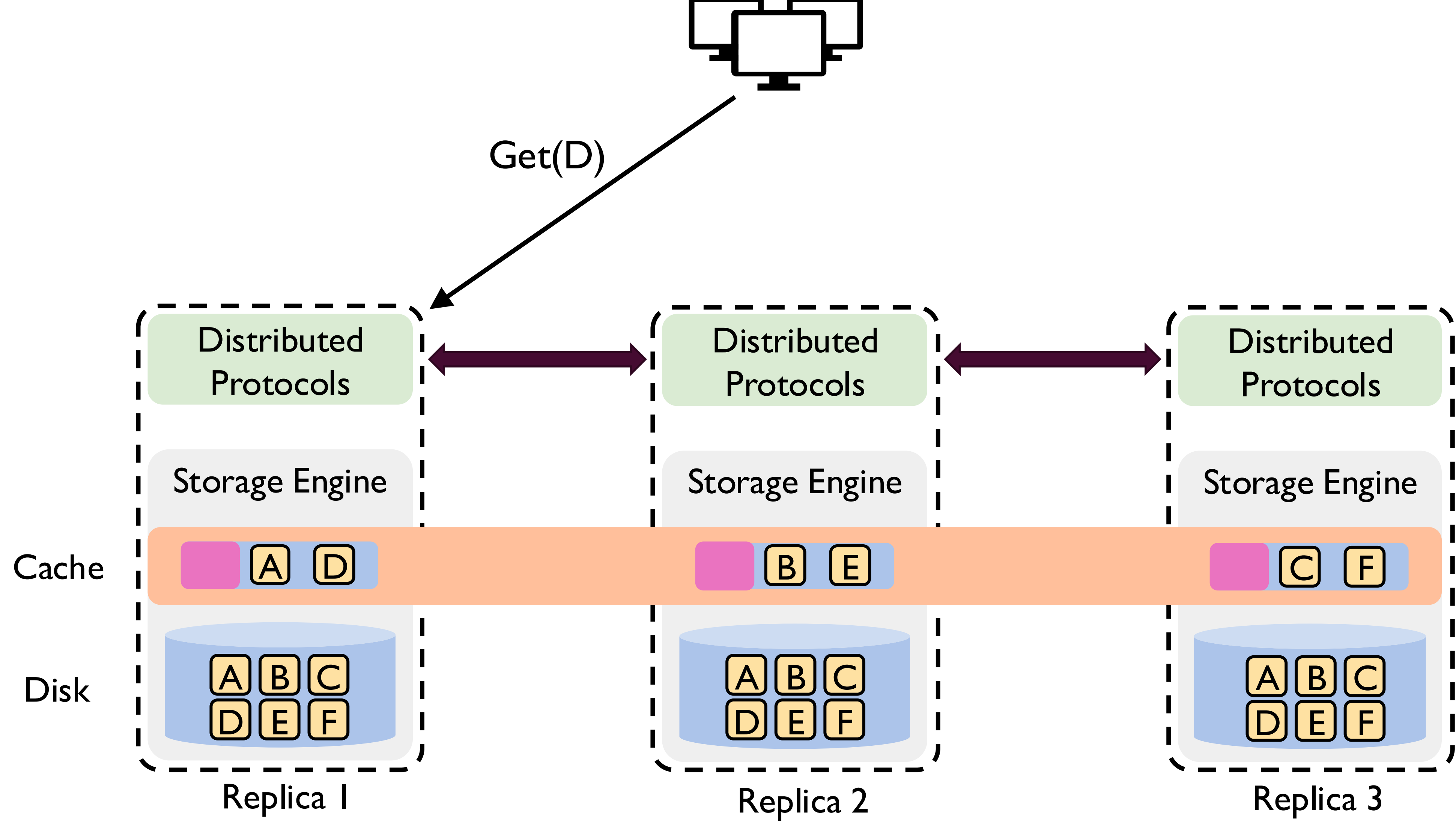
Storage Engine | Storage Engine | Storage Engine

Cache: A D | B E | C F

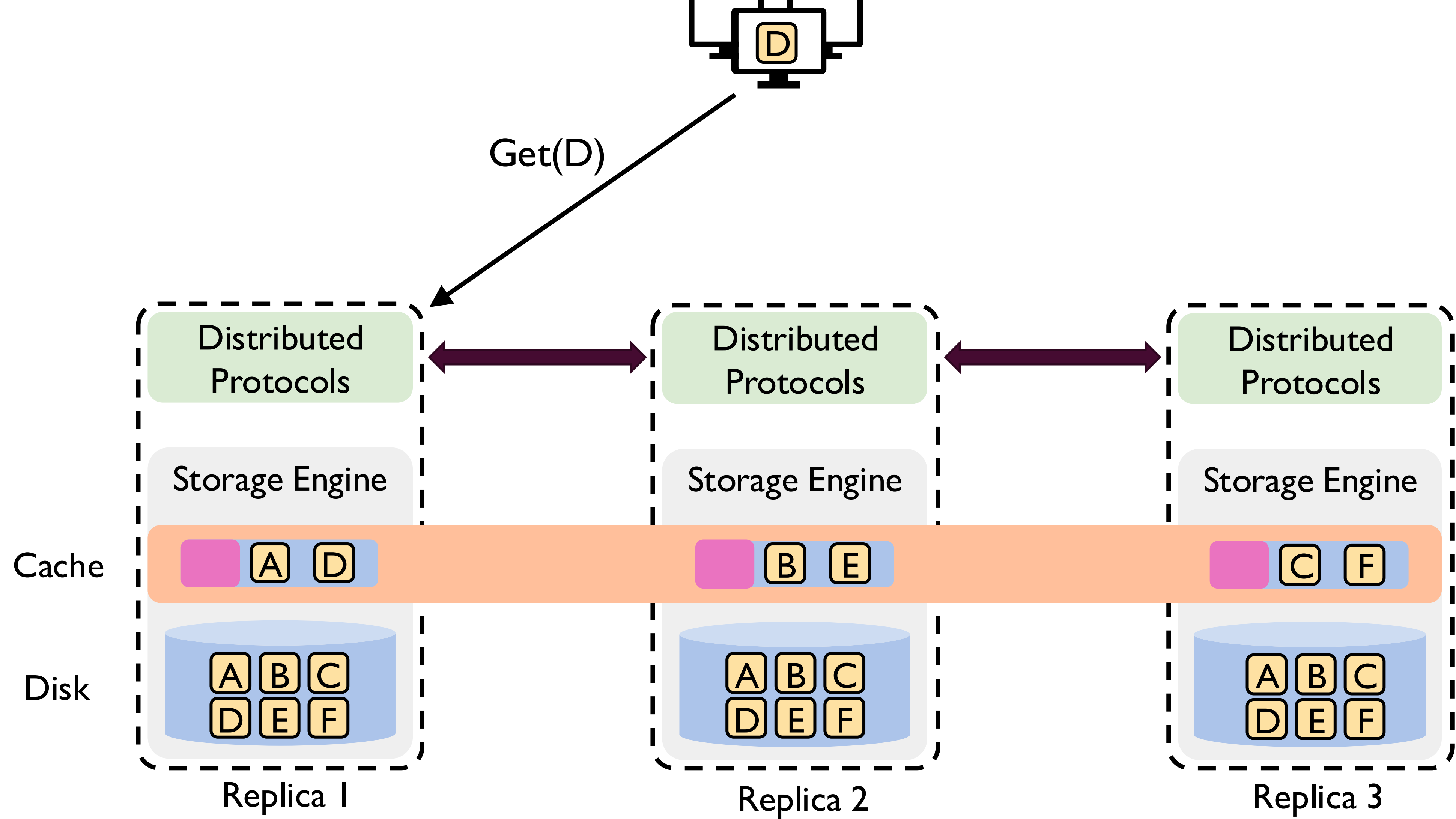
Disk: A B C D E F | A B C D E F | A B C D E F

Replica 1 | Replica 2 | Replica 3

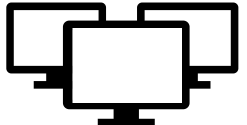
LDC Read and Write Path



LDC Read and Write Path



LDC Read and Write Path



Distributed Protocols | Distributed Protocols | Distributed Protocols

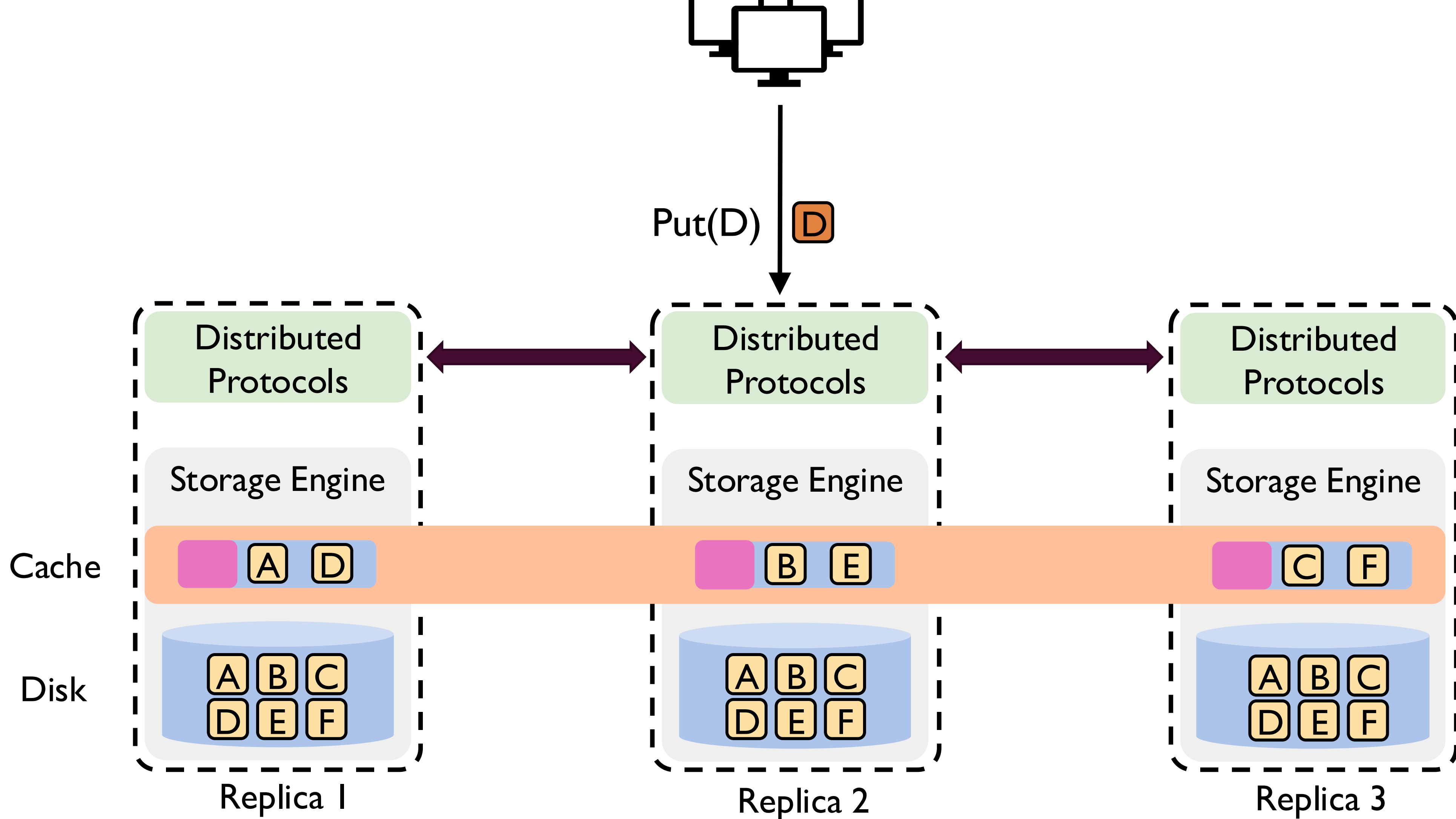
Storage Engine | Storage Engine | Storage Engine

Cache: A D | B E | C F

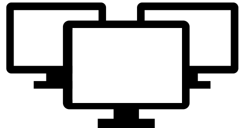
Disk: A B C D E F | A B C D E F | A B C D E F

Replica 1 | Replica 2 | Replica 3

LDC Read and Write Path



LDC Read and Write Path

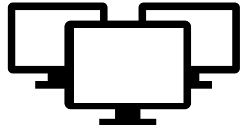


Distributed Protocols
D
Storage Engine
Cache
A D
Disk
A B C
D E F
Replica 1

Distributed Protocols
D
Storage Engine
B E
A B C
D E F
Replica 2

Distributed Protocols
D
Storage Engine
C F
A B C
D E F
Replica 3

LDC Read and Write Path



Distributed Protocols | Distributed Protocols | Distributed Protocols

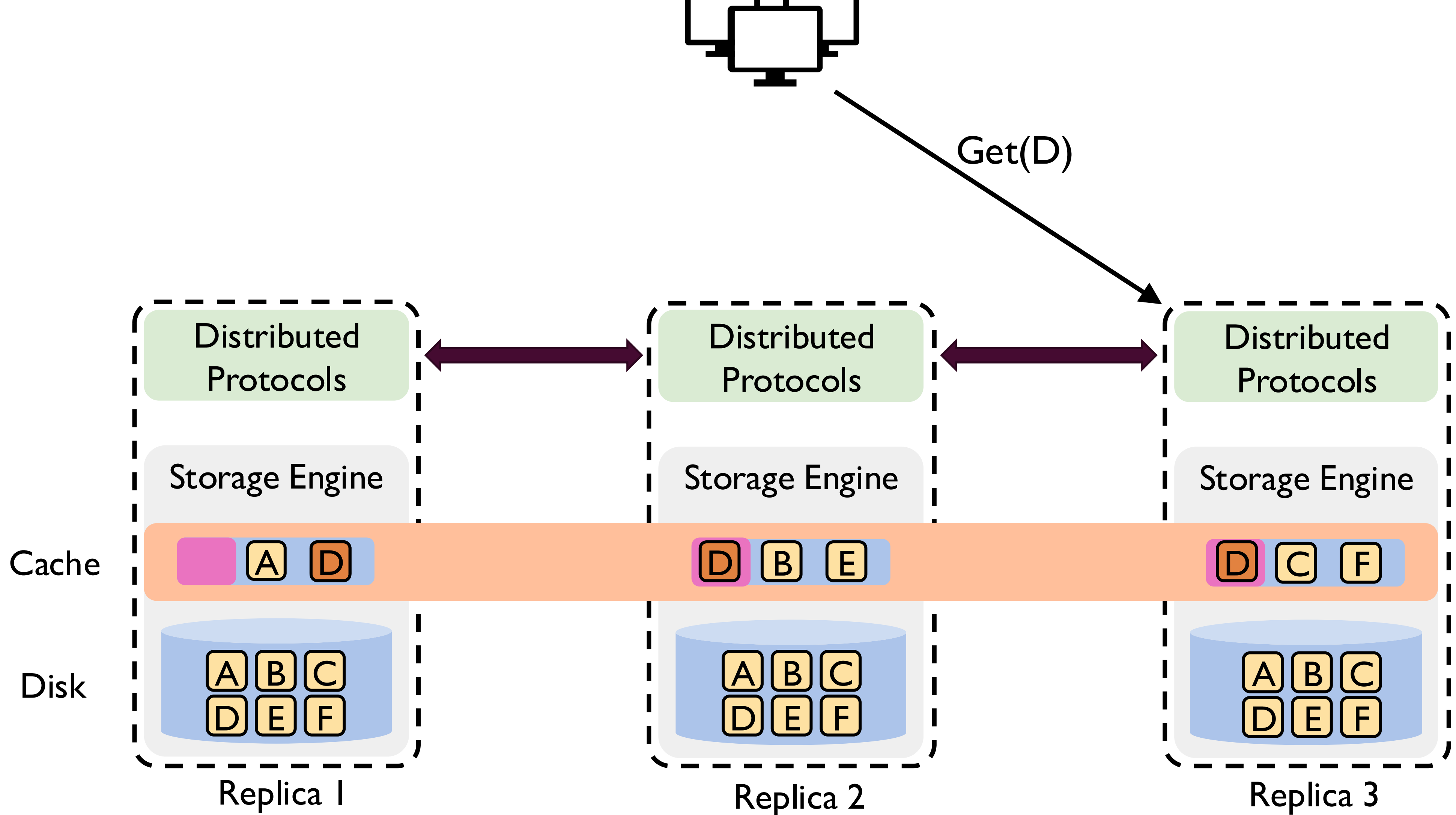
Storage Engine | Storage Engine | Storage Engine

Cache: A D | D B E | D C F

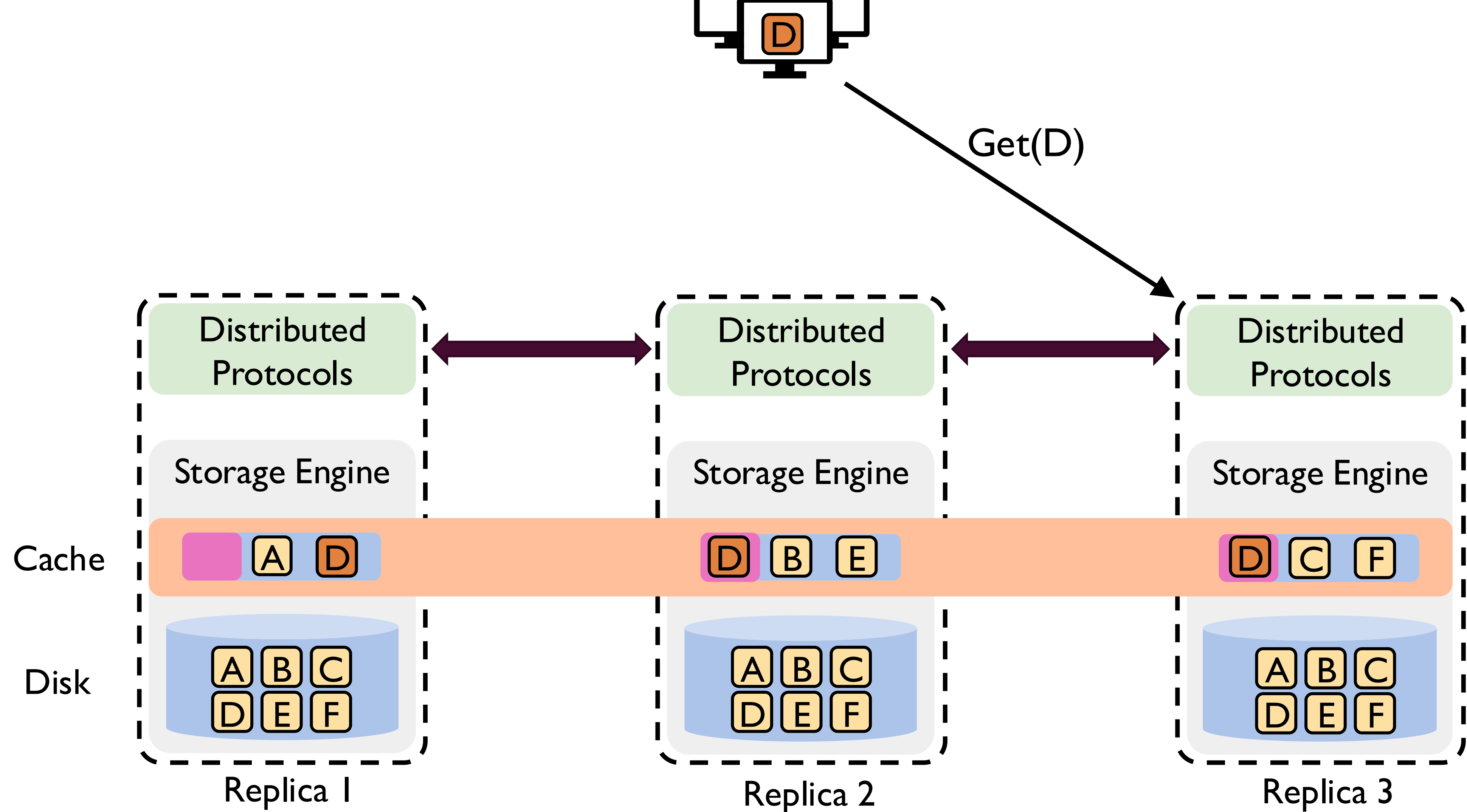
Disk: A B C D E F | A B C D E F | A B C D E F

Replica 1 | Replica 2 | Replica 3

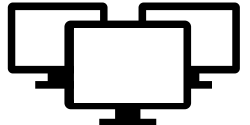
LDC Read and Write Path



LDC Read and Write Path



LDC Read and Write Path



Distributed Protocols | Distributed Protocols | Distributed Protocols

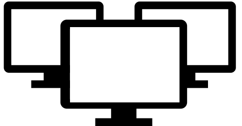
Storage Engine | Storage Engine | Storage Engine

Cache: A D | D B E | D C F

Disk: A B C D E F | A B C D E F | A B C D E F

Replica 1 | Replica 2 | Replica 3

LDC Read and Write Path



Distributed Protocols | Distributed Protocols | Distributed Protocols

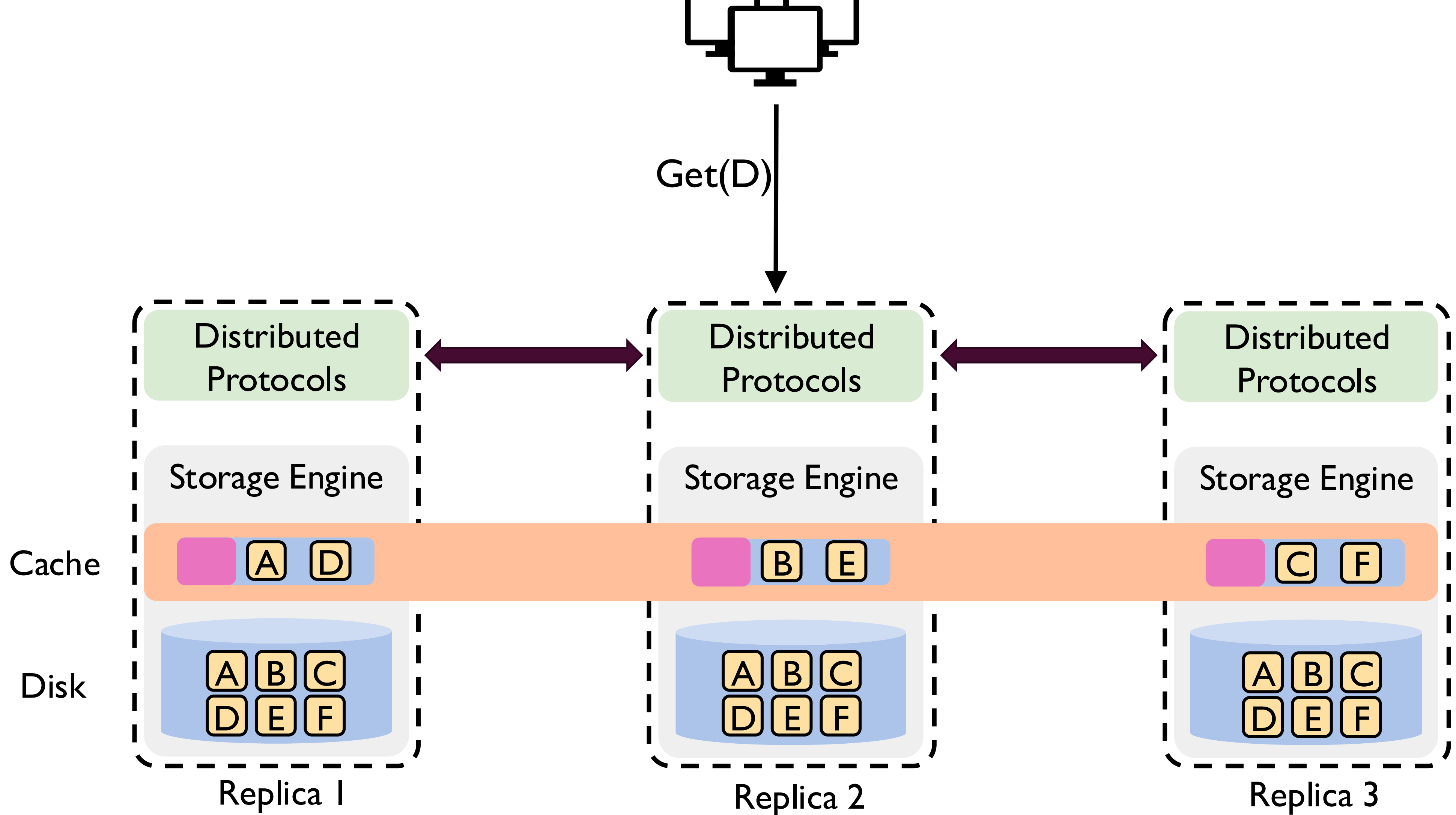
Storage Engine | Storage Engine | Storage Engine

Cache: A D | B E | C F

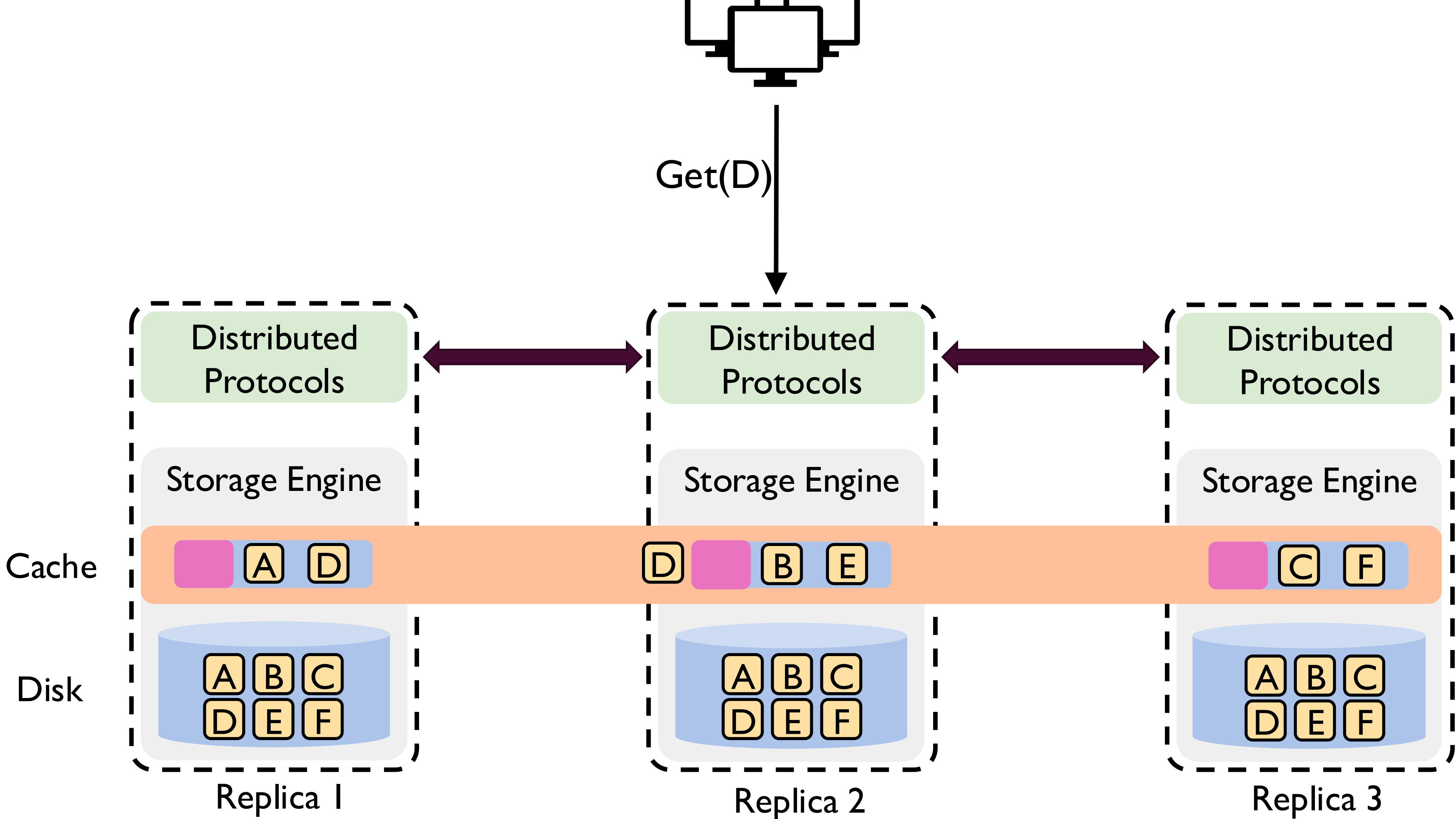
Disk: A B C D E F | A B C D E F | A B C D E F

Replica 1 | Replica 2 | Replica 3

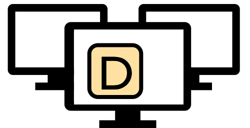
LDC Read and Write Path



LDC Read and Write Path



LDC Read and Write Path



Distributed Protocols | Distributed Protocols | Distributed Protocols

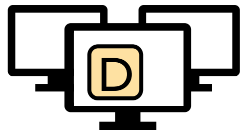
Storage Engine | Storage Engine | Storage Engine

Cache: A D | B E | C F

Disk: A B C D E F | A B C D E F | A B C D E F

Replica 1 | Replica 2 | Replica 3

LDC Read and Write Path



Distributed Protocols | Distributed Protocols | Distributed Protocols

Storage Engine | Storage Engine | Storage Engine

Cache: A D | B E | C F

Disk: A B C D E F | A B C D E F | A B C D E F

Replica 1 | Replica 2 | Replica 3

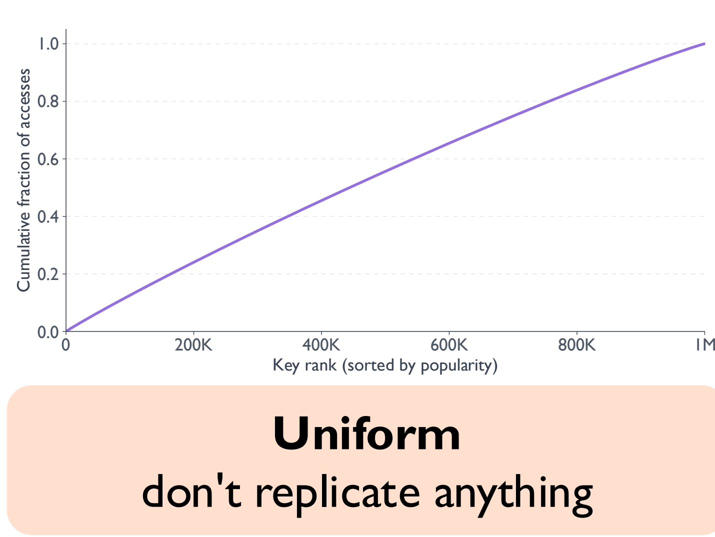
Cost-Benefit Analyzer – Is Eliminating Redundancy Always Good?

Cost-Benefit Analyzer – Is Eliminating Redundancy Always Good?

Every replica has 10% of the dataset size as cache

Cost-Benefit Analyzer – Is Eliminating Redundancy Always Good?

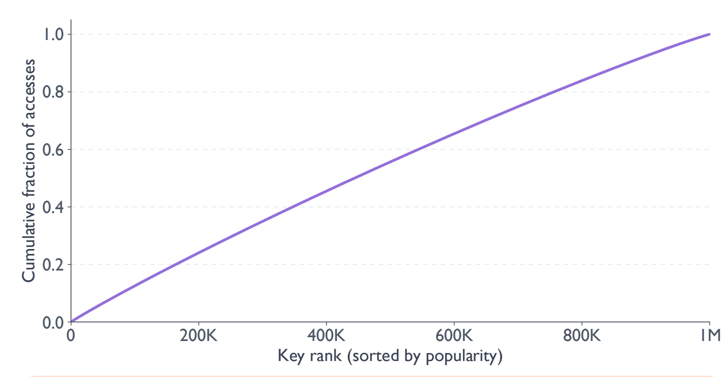
Every replica has 10% of the dataset size as cache



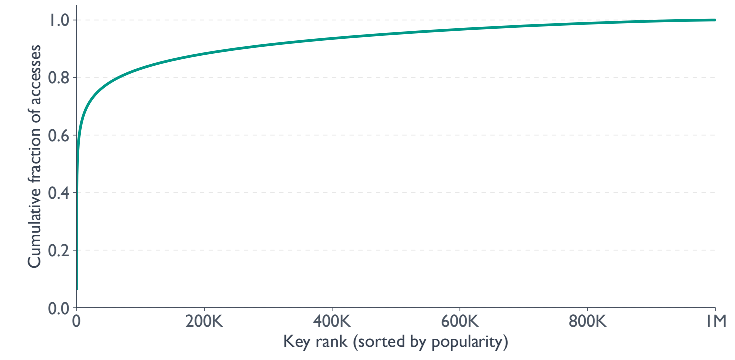
Uniform
don't replicate anything

Cost-Benefit Analyzer – Is Eliminating Redundancy Always Good?

Every replica has 10% of the dataset size as cache



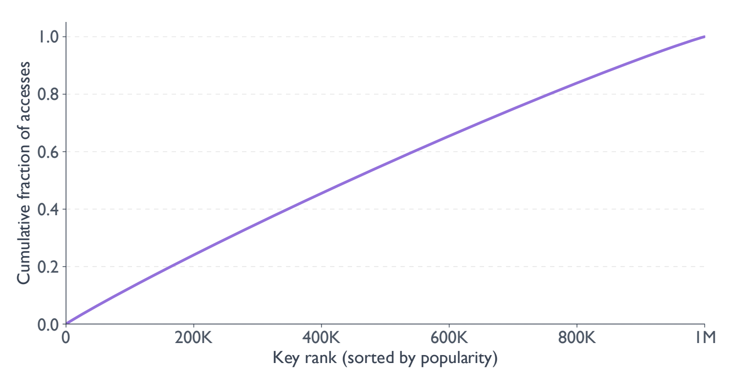
Uniform
don't replicate anything



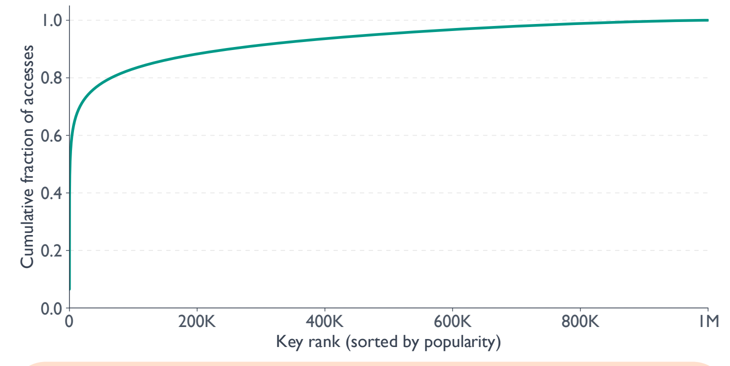
Zipfian
replicate a few, keep the tail covered

Cost-Benefit Analyzer – Is Eliminating Redundancy Always Good?

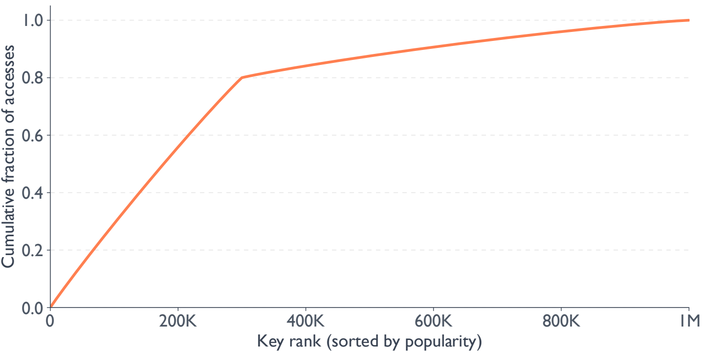
Every replica has 10% of the dataset size as cache



Uniform
don't replicate anything



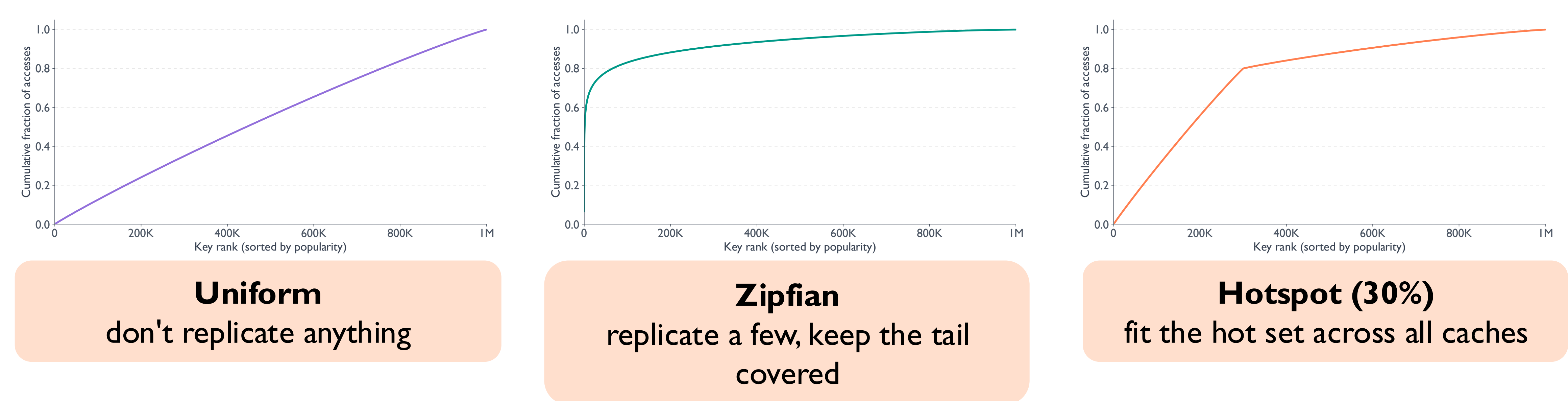
Zipfian
replicate a few, keep the tail covered



Hotspot (30%)
fit the hot set across all caches

Cost-Benefit Analyzer – Is Eliminating Redundancy Always Good?

Every replica has 10% of the dataset size as cache



CBA picks the right balance **automatically**.
Admitting locally only when the gain from future local hits outweighs the coverage lost by evicting other objects.

Outline

- Introduction
- Study
- Our Idea - Logically Disaggregated Cache
- Implementations
- Evaluation

Three Implementations

Twig-KV – Eventually consistent system

Craq-KV – Strongly consistent system

RethinkDB – Production system

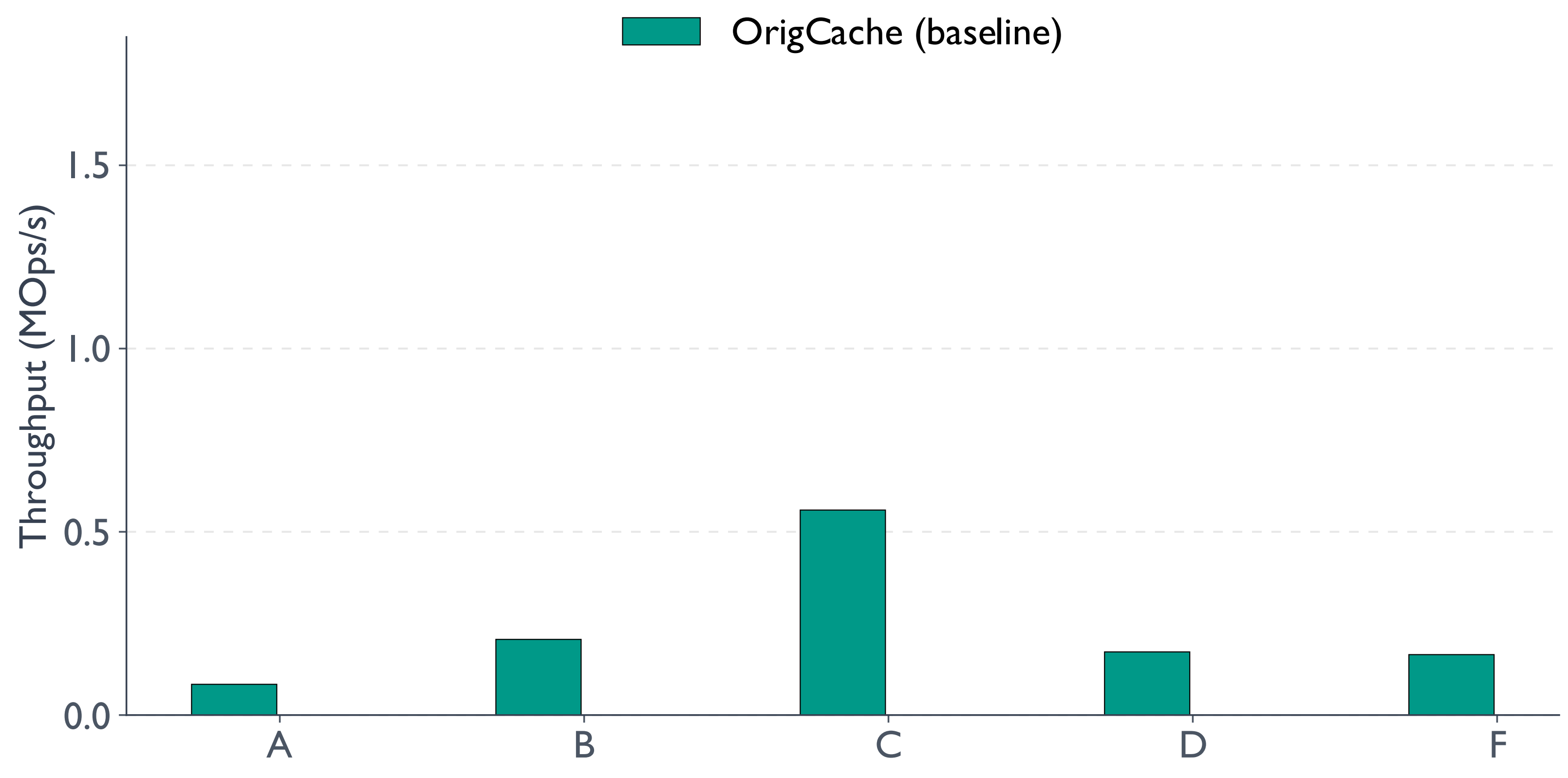
Outline

- Introduction
- Study
- Our Idea - Logically Disaggregated Cache
- Implementations
- Evaluation

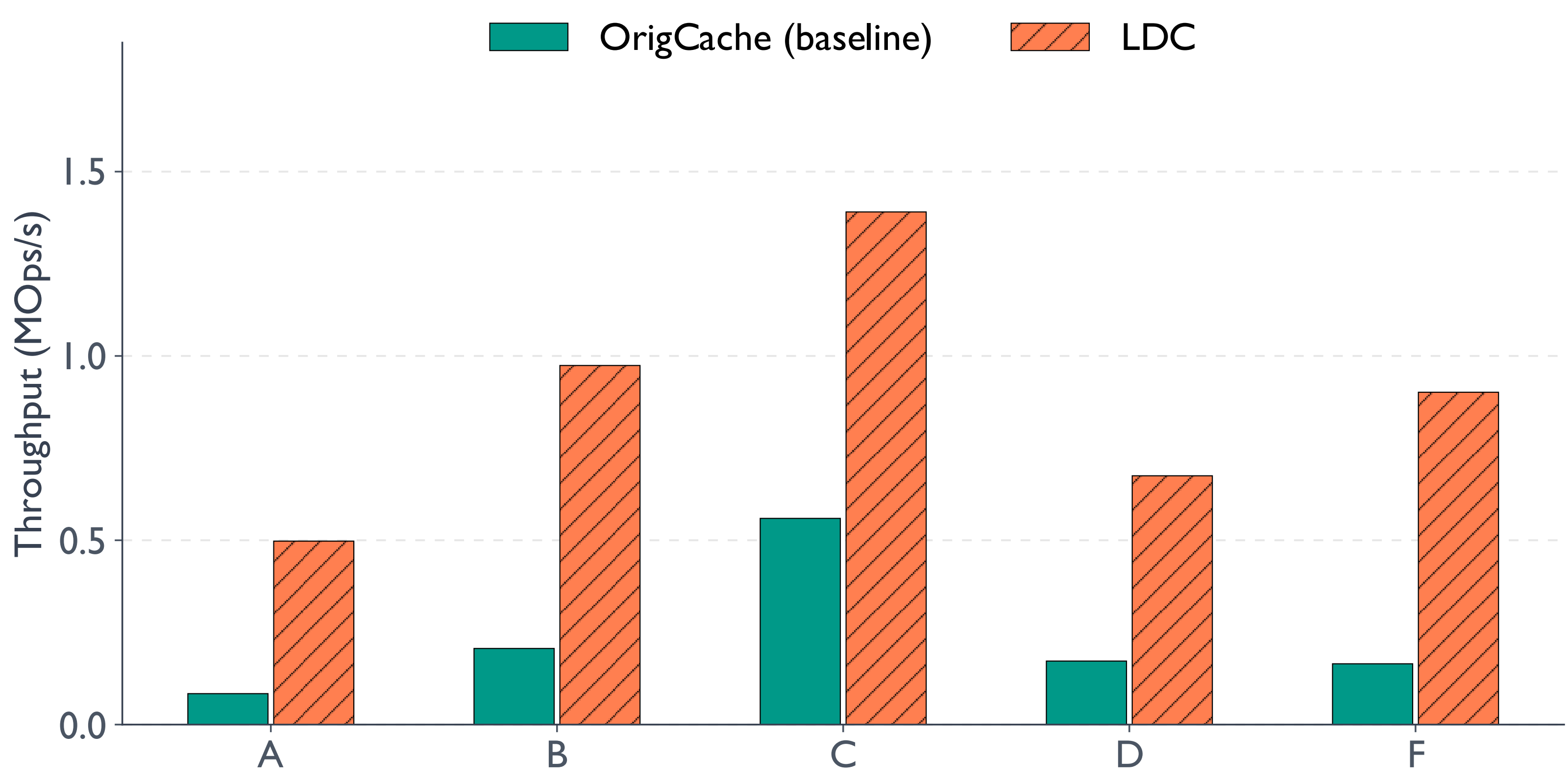
Evaluation – Setup

- Dataset: 10M key-value pairs, 24B key size, 100B value size
- OrigCache: system with original siloed caches
- LDC: system with LDC

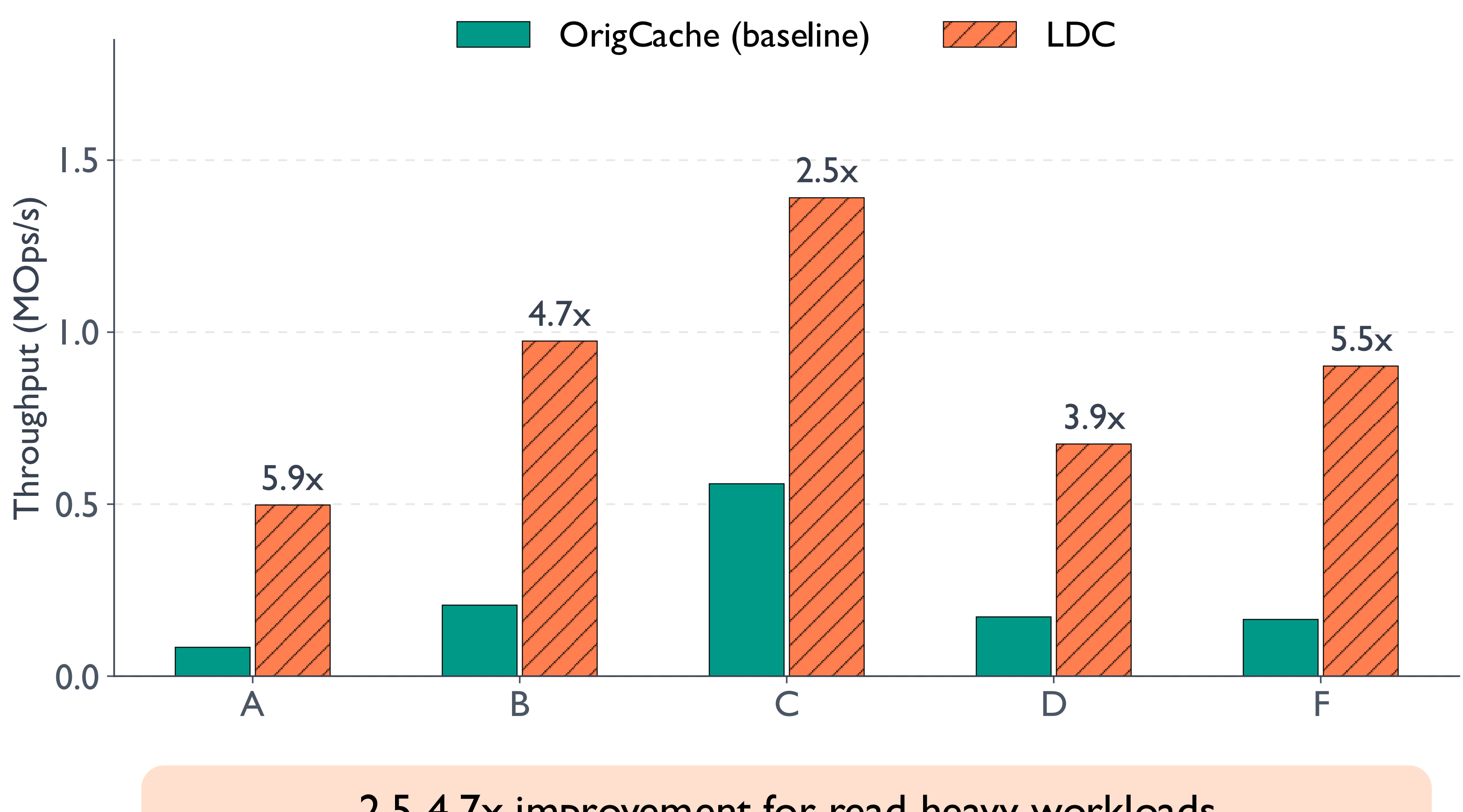
Evaluation - YCSB Benchmark – Craq-KV



Evaluation - YCSB Benchmark – Craq-KV

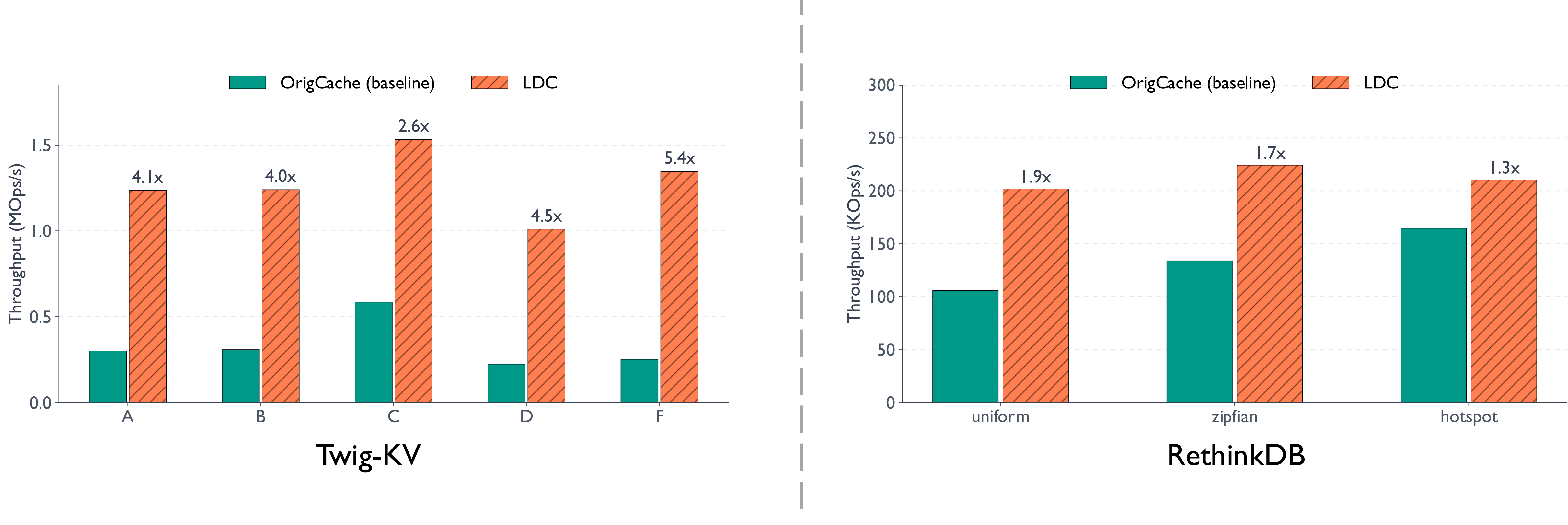


Evaluation - YCSB Benchmark – Craq-KV



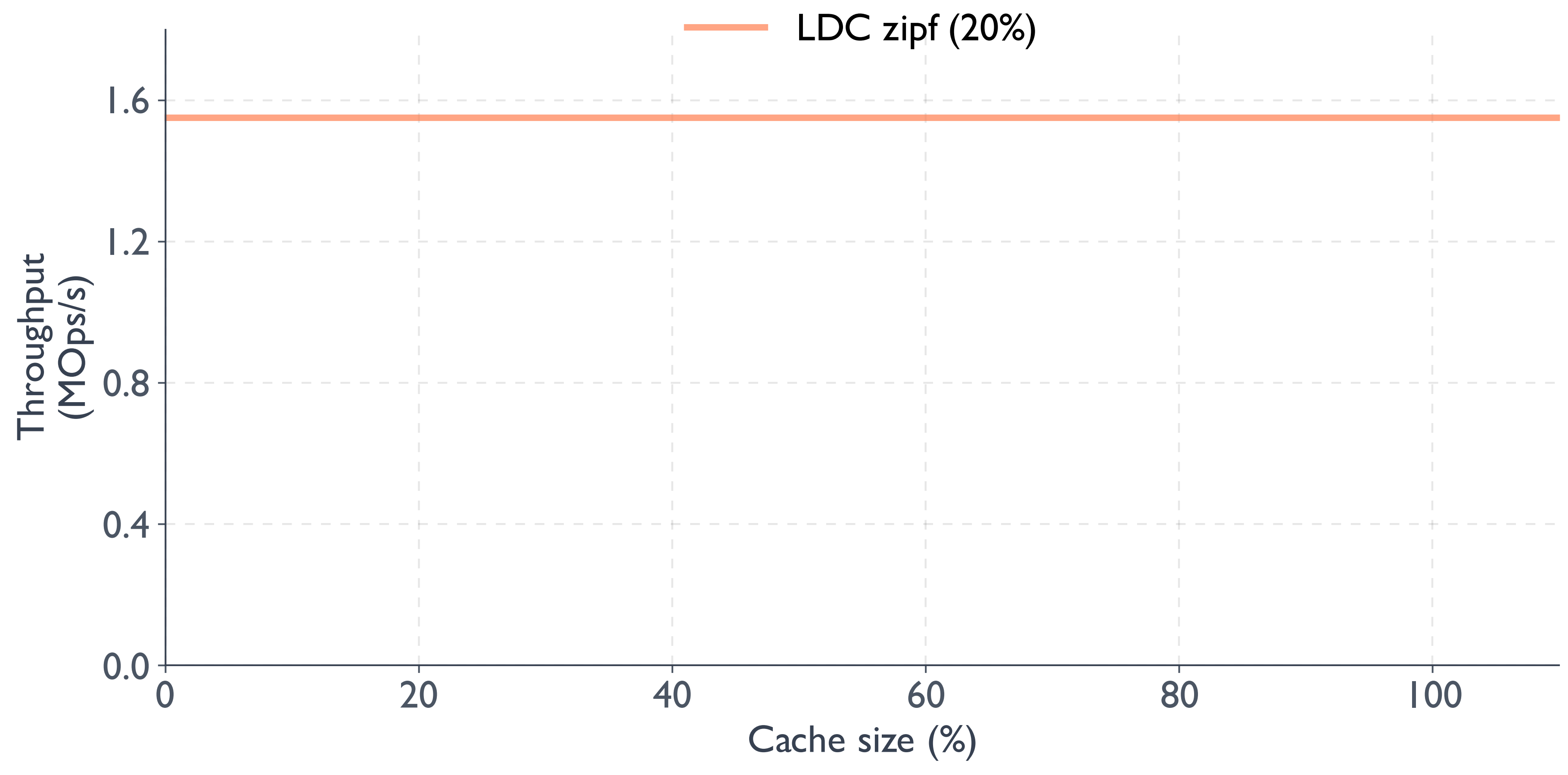
2.5-4.7x improvement for read-heavy workloads
Upto 5.9x for write-heavy workloads

Evaluation - YCSB Benchmark

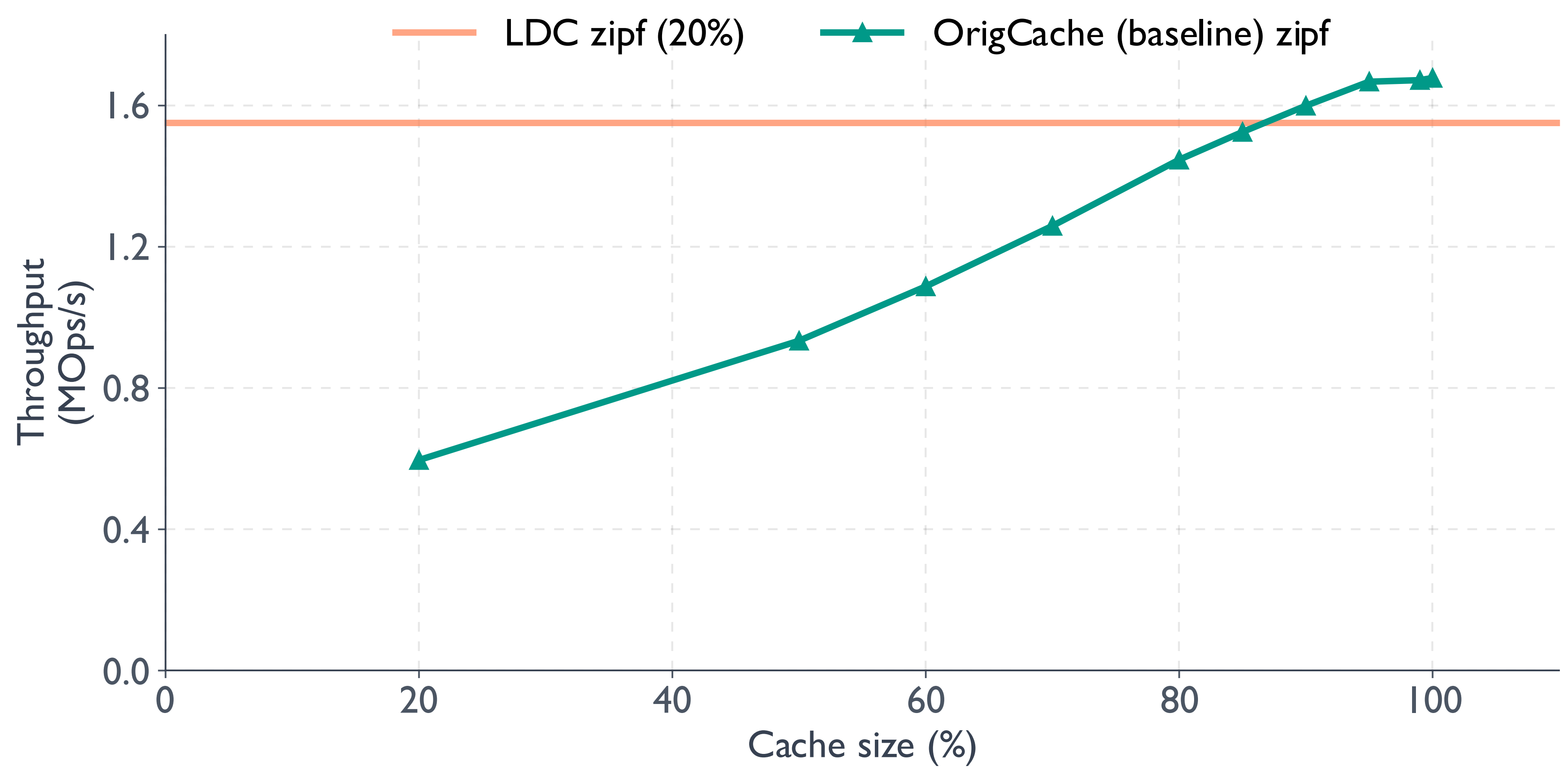


Trend holds true across weakly-consistent systems and production-grade systems as well

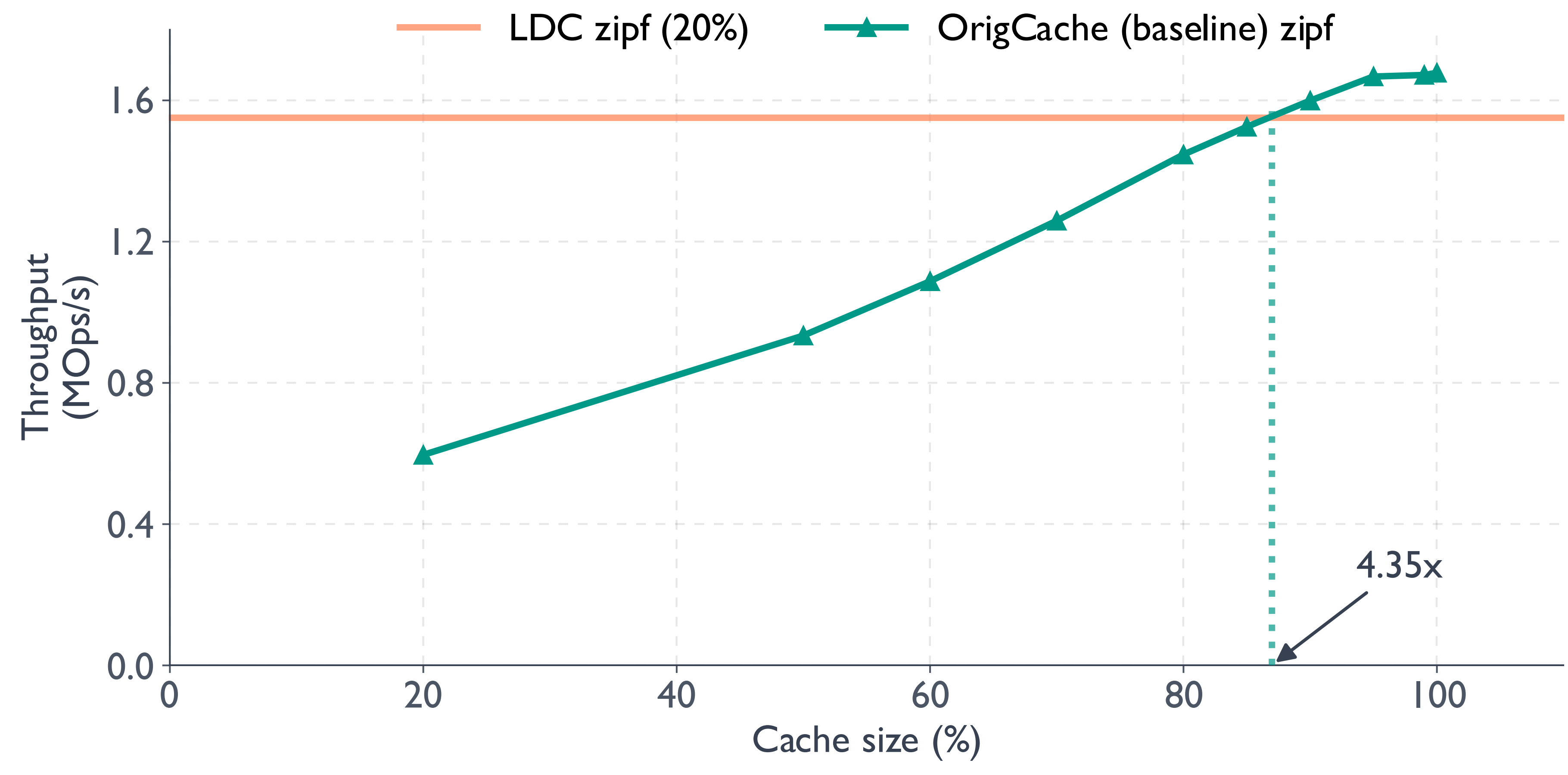
Evaluation – At What Cache Size Does The Unmodified System Match The Performance of The LDC Version



Evaluation – At What Cache Size Does The Unmodified System Match The Performance of The LDC Version



Evaluation – At What Cache Size Does The Unmodified System Match The Performance of The LDC Version



OrigCache requires 4.35x larger per-replica caches to match LDC performance

More Results In The Paper

- **Microbenchmarks**
  - Read-only across uniform/zipfian/hotspot
  - Write/read latency and write-around comparison
- **Scalability**
  - Replica scaling (3 → 5 replicas)
  - NVMe SSD performance
  - Larger datasets (25M, 50M keys)
- **Comparisons**
  - Key-based routing (writes, nested queries, joins)
  - Cooperative caching (n-chance)
  - Better local cache policies (S3-FIFO)
- **Real-World Workloads**
  - 50 traces from Twitter, Meta, Alibaba
  - CBA efficacy analysis

Summary

Summary

Problem: Embedded caches in replicated storage systems waste memory — replicas manage their caches in silos, redundantly caching the same objects

Summary

Problem: Embedded caches in replicated storage systems waste memory — replicas manage their caches in silos, redundantly caching the same objects

Logically Disaggregated Cache (LDC): Unify per-replica caches into a single logical cache

- **Remote Access** — reduces read redundancy, handles opaque queries
- **Selective Quick Demotion** — a tiny queue absorbs writes without polluting the cache
- **Cost-Benefit Analyzer** — balances coverage and redundancy across workloads

Summary

Problem: Embedded caches in replicated storage systems waste memory — replicas manage their caches in silos, redundantly caching the same objects

Logically Disaggregated Cache (LDC): Unify per-replica caches into a single logical cache

- **Remote Access** — reduces read redundancy, handles opaque queries
- **Selective Quick Demotion** — a tiny queue absorbs writes without polluting the cache
- **Cost-Benefit Analyzer** — balances coverage and redundancy across workloads

Results:

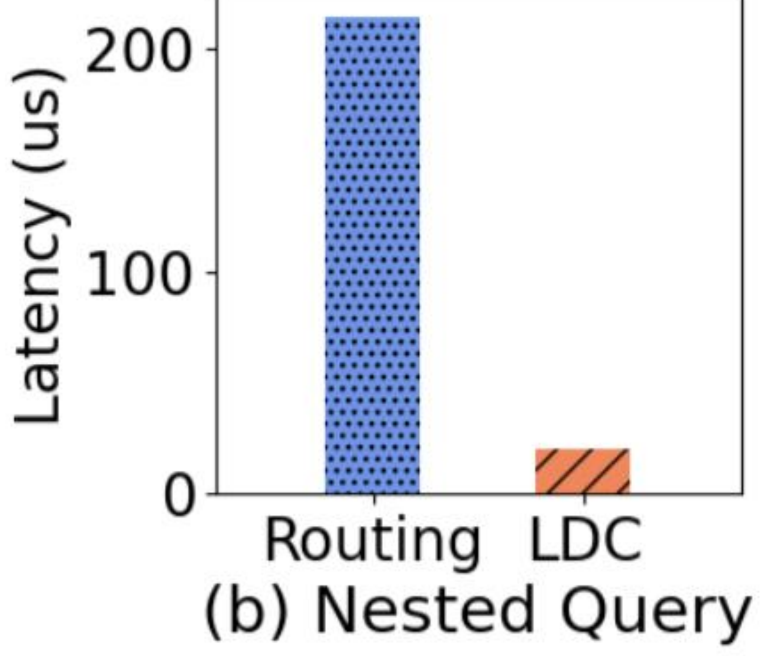
- Up to **5.9× throughput.**

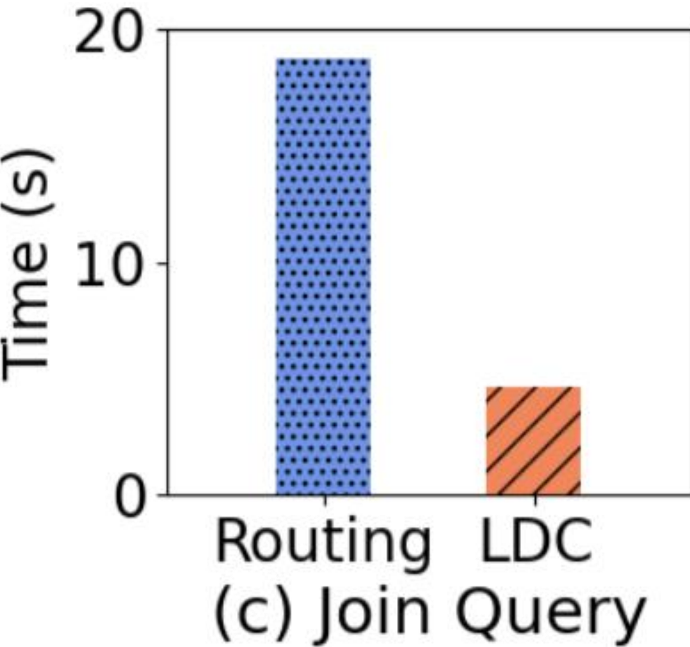
Backup Slides

Why Don't Techniques from External Caches Work

Does not handle writes

Needs accurate routing





CBA – Cost benefit Analysis

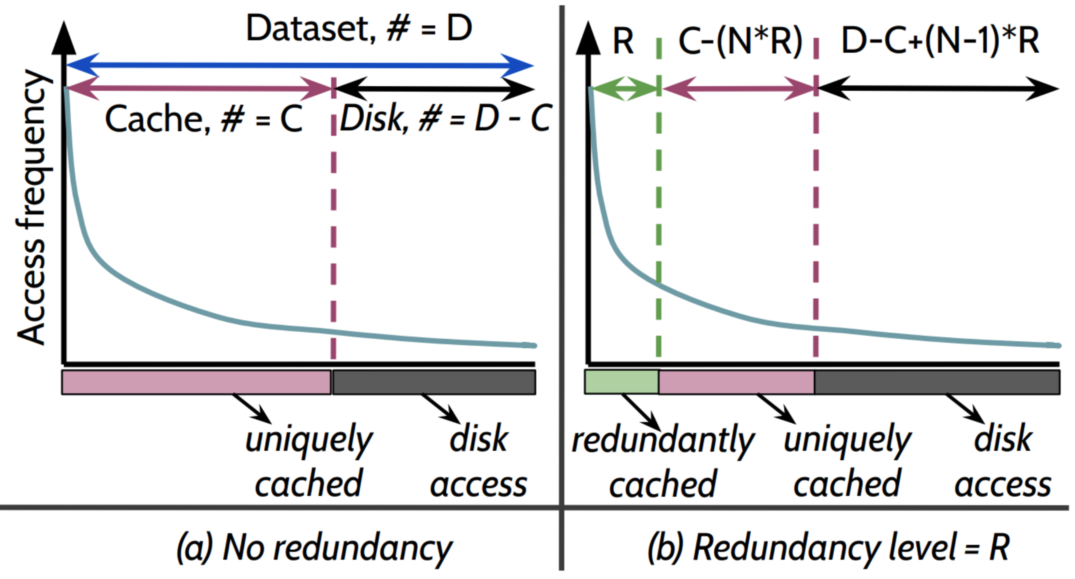
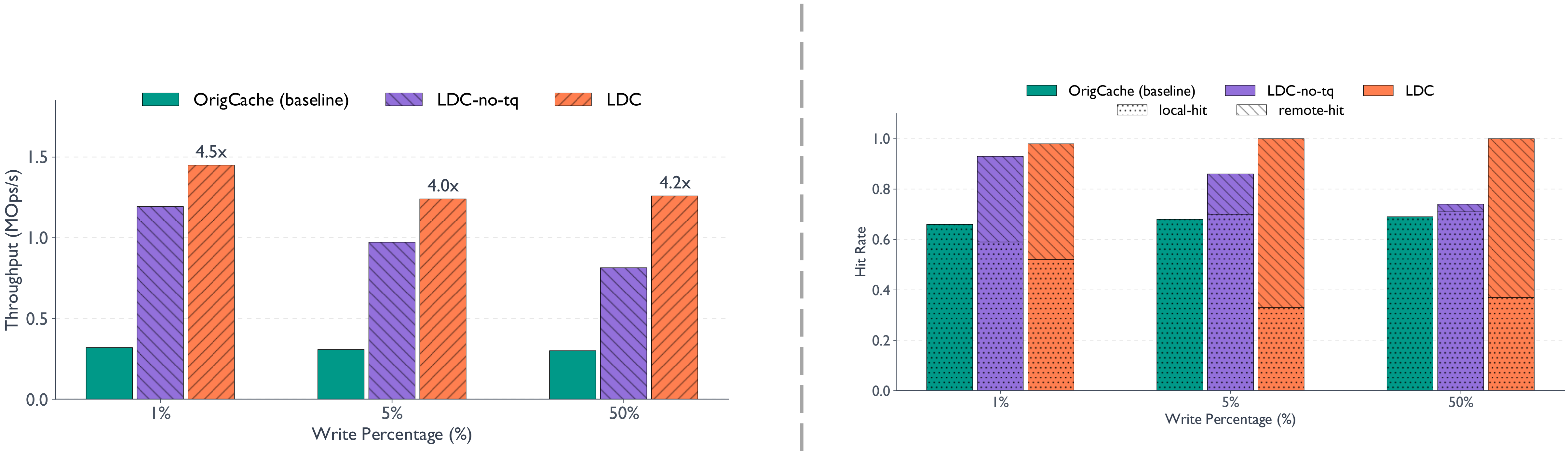


Figure 5. Cost-Benefit Analysis. *The figure shows the analyses performed by the CBA. In (a), the CBA explores the point with no redundancy. In (b), the CBA explores the impact of redundantly caching top R objects on all N replicas, which decreases cache coverage by* $(N-1)*R$.

Evaluation – What Is The Benefit of Quick Demotion



LDC trades of local hits for remote hits and achieves higher total hit rates by avoiding write-induced redundancy

Evaluation – How LDC scales