CORTX deduplication architecture overview

nikita.danilov@seagate.com

Goals

- Add data de-duplication capability to CORTX/motr software
- Software-only
    - should work in opensource version
    - *may* use advanced capabilities of Seagate hardware when available
- Online: de-duplicate directly in IO path, not in a separate background process
- Flexible
    - various work-loads (do not assume spatial or temporal properties)
    - hash-block size specifiable by user (user is S3 or NFS or …)
    - sliding windows (rolling hash) support
    - hash strength selectable by user
    - de-duplication domain (server, pool, bucket, …) specifiable by user
    - user specifies the need in collision detection (*full block check*) for each IO

Goals

- Layered
    - de-duplication is transparent after configuration (IO interface is the same)
    - storage for de-duplicated blocks uses normal motr layout (parity de-clustered, with repairs, distributed spare, *etc*.)
- Scalable
    - no meta-data or IO hotspots
- Explore options

Possible goals

- De-duplication for meta-data (key-value stores)?

Implementation overview

- De-duplication is implemented in motr.
- In motr each object (think file) has a layout: an attribute that defines how the object is stored.
- Currently 2 types of layouts are supported:
    - N+K+S striping in distributed parity de-clustered RAID, and
    - "composite layout" for objects composed of other objects (for snapshots, HSM tiering, *etc*.)
- New de-duplication layout type is introduced.
- An object is assigned a layout at the creation time.
- An object with de-duplication layout belongs to a *de-duplication domain*.
- De-duplication is done on top of erasure coding: first de-duplicate, then store unique de-duplicate blocks with erasure coding.

De-duplication domain

- There can be multiple de-duplication domains per cluster
- Each de-duplication domain contains:
  - hash-map: a distributed index mapping block hashes to *block locations*,
  - block-store: a configurable number of *block store objects*, used to store unique blocks within the domain.
- Hash-block size limits and hash function are specified for each domain.
- Hash-map is accessed through the standard motr indexing interface, normal replication and repair mechanisms apply to it.
- Block store objects are stored with a normal N+K+S layout, with striping and repairs.
- Block location is `(block-store-object-idx, offset-in-block-store-object)` pair. It identifies the location of stored hash-block within the domain.
- *Uniqueness counter* (UC) is used to tell hash-colliding blocks.

Sliding window (rolling hash)

- To improve de-duplication efficiency block boundaries are selected by a rolling hash.
- Block boundaries are selected by rolling hash ("X lowest bits of the rolling hash are 0"), subject to block size limits (minimum and maximum).
- This works for sequential IO.
- For non-sequential writes (seek), rolling hash is reset.
- Overwrites are interesting.

De-duplication domain

hash map

hash	UC	block location	ref count
ff24df82df86515b6c2a84cd52e6279d	0	(0, 10)	1
5a11bd26d1df03fcd302f8ca6af65f00	0	(1, 11)	1
897316929176464ebc9ad085f31e7284	0	(2, 2)	3

...

block store objects

obj_0

obj_1

obj_2

...

Collisions with full block check

hash map

hash	UC		block location	ref count
ff24df82df86515b6c2a84cd52e6279d	0	→	(0, 10)	1
ff24df82df86515b6c2a84cd52e6279d	1	→	(1, 11)	1
ff24df82df86515b6c2a84cd52e6279d	2	→	(2, 2)	3

...

Per-object meta-data

- For an object with de-duplication layout, block locations of all its extents are stored as object attributes.
- In motr, object attributes are stored in a global distributed index called *namespace*.
- Key in namespace is (roughly) `object_name+attribute_id`, and the corresponding value is `attribute_value`.

namespace

object_name	attr_id: [start, end]		block location
`bucket/foo`	`DEDUP:[0, 100]`	→	`(0, 10)`
`bucket/foo`	`DEDUP:[101, 109]`	→	`(1, 11)`
`bucket/bar`	`DEDUP:[110, 130]`	→	`(2, 2)`

...

Hash map and attributes

- Hash map is a *distributed index*: stores key-value pairs:
  - Key: `[block-hash, uniqueness-counter]`
  - Value: `[block-store-object, block-store-offset, reference-counter]`
- Index is replicated: each key-value pair has multiple copies, say R=2
- Index is distributed: key-value pairs are distributed over P nodes, P >= R
- Index is implemented by b-trees: fast logarithmic lookups and insertions
- In case of node or device failure, the index is repaired

- Per-object attributes are stored in a similar distributed index

Read path

```
dedup_read(fid, block_nr) {
    dom = dedup_domain_get(fid);
    loc = obj_attr_lookup(fid, "DEDUP:" + string(block_nr));
    return read(dom.block_store_obj[loc.obj_nr], loc.idx);
}
```

- This is simplified pseudo-code. Real entry point is asynchronous and vectored (scatter-gather-scatter).
- Object attribute lookup is asynchronous, wait for completion is needed.
- De-duplication domain structure is initialised during startup, this includes fetching identities (fids) of all block store objects.

Simplest write path

```
dedup_write(fid, block_nr, data) {
    dom = dedup_domain_get(fid);
    (loc, ref) = index_lookup(dom.hashmap, dom.hash(data));
    if (loc == NOT_FOUND) {
        loc = block_location_alloc(dom);
        ref = 1;
        write(dom.block_store_obj[loc.obj_nr], loc.idx, data);
    } else
        ref++;
    index_insert(dom.hashmap, dom.hash(data), (loc, ref));
    obj_attr_add(fid, "DEDUP:" + string(block_nr), loc);
}
```

- `block_location_alloc()` allocates new block location in the domain. This can be done either by synchronous lookup or by granting ranges of locations to clients.
- For simplicity, overwrite of the same block in an object is not considered.

Write with full block verification

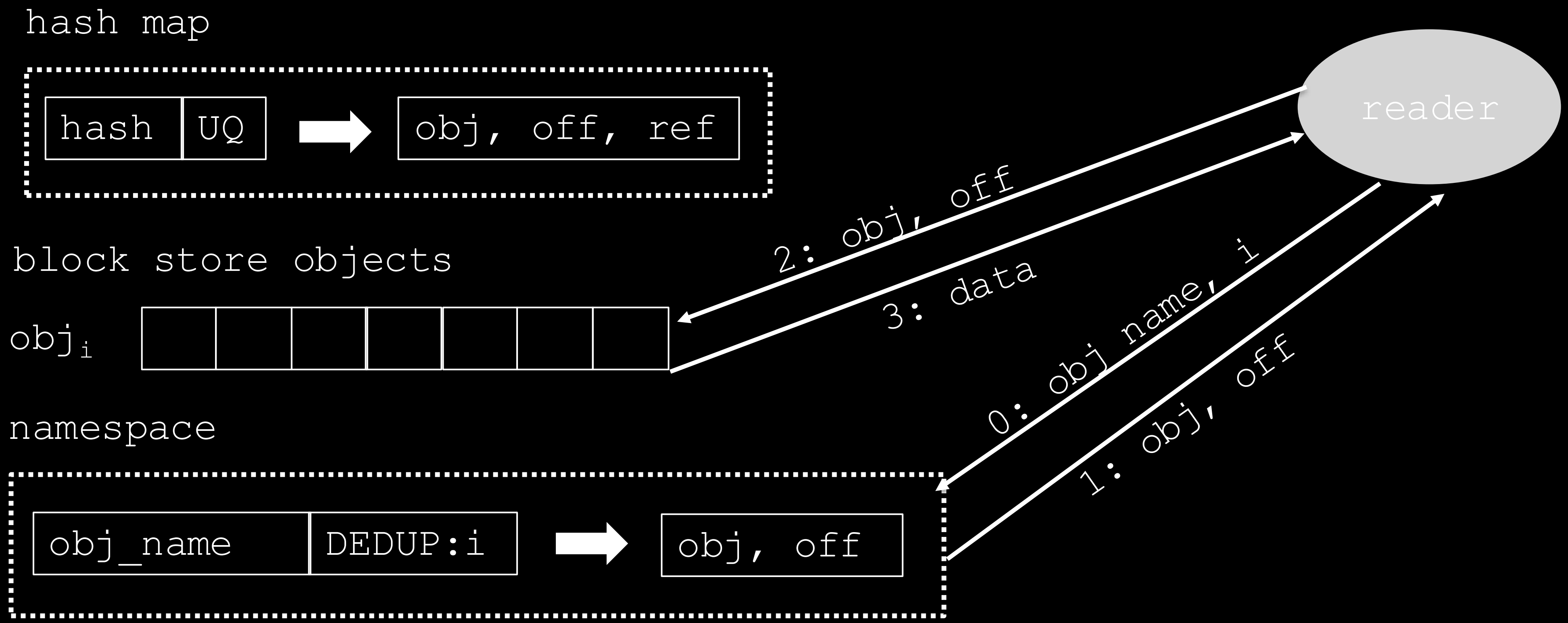
```
dedup_write(fid, block_nr, data, verify) {
    dom = dedup_domain_get(fid);
    (loc, ref) = index_lookup(dom.hashmap, dom.hash(data));
    if (loc == NOT_FOUND) {
        loc = block_location_alloc(dom);
        ref = 1;
        write(dom.block_store_obj[loc.obj_nr], loc.idx);
    } else if (verify) {
        (loc, ref) = collision_resolve(...);
    } else
        ref++;
    index_insert(dom.hashmap, dom.hash(data), (loc, ref));
    obj_attr_add(fid, "DEDUP:" + string(block_nr), loc);
}
```

Write with full block verification

```
collision_resolve(...)
{
    for ((uc, loc, ref) in index_iterate(dom.hashmap, dom.hash(data))) {
        other = read(dom.block_store_obj[loc.obj_nr], loc.idx);
        if (data == other)
            return (uc, loc, ref + 1); /* Full match found. */
    }
    return (uc++, block_location_alloc(dom), 1);
}
```

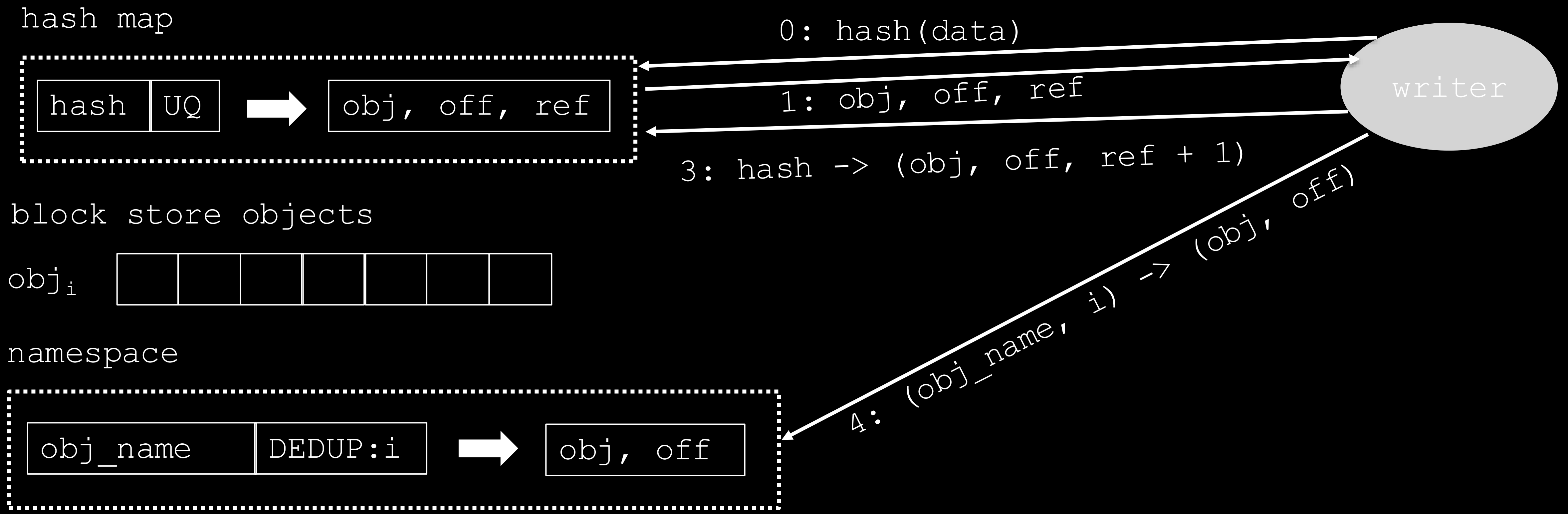
- Iterate over all recorded blocks with the same hash.
- If full match is found, reuse it.
- Otherwise, create a new record with the same hash and different UQ.

Data-flow

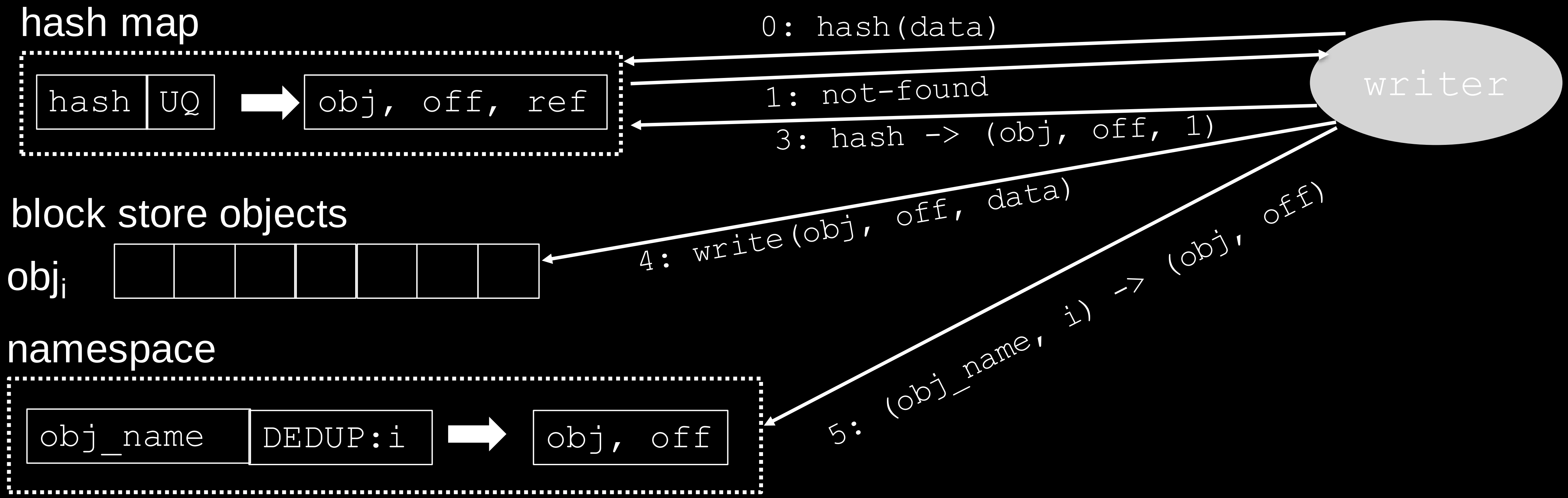


hash map
hash
UQ
obj, off, ref
block store objects
obj_i
namespace
obj_name
DEDUP:i
obj, off
reader
2: obj, off
3: data
0: obj name, i
1: obj, off

Data-flow: hash hit



Data-flow: hash miss



Cost: read

	Network			Storage		
	msg	**max**	**byte**	**iop**	**max**	**byte**
`obj_attr_lookup(...)`	1	1	4KB	logN	logN	4KB
`read(dom.block_store_obj...)`	1	1	bs	logN+1	logN+1	4KB+bs
Total	2	2	4KB+bs	2logN+1	2logN+1	8KB+bs

- logN – typical number of uncached b-tree nodes in a tree traversal.
- bs – hash block size.
- max - number of synchronous round-trips (some messages can be parallel).

Cost: write-no-verify

	Network			Storage		
	msg	max	byte	iop	max	byte
`index_lookup(...)`	1	1	4KB	logN	logN	4KB
`block_location_alloc(...)`	1/A	1/A	4KB/A			
`index_insert(...)`	K_M+1	1	(K_M+1)*4KB			
`obj_attr_add(...)`	K_M+1	1	(K_M+1)*4KB			
`write(...)`	(K_D+1)*f	f	(K_D+1)*bs*f			
Total	f*(K_D+1)+2K_M+3+1/A	2, A=infty				

- A - amortisation factor of block location allocator.
- K_M - meta-data replication factor (meta-data striping is $1+K_M$).
- K_D - block store objects are striped with $N_D+K_D+S_D$.
- f - fraction of unique (not duplicate) blocks.

Choices and directions

- Hash-block size
- Hash function (hash size, strength)
- Full block verification?
- Free blocks on truncate?
- Hash map can be striped across a large number of devices. This gives large aggregate IOPS budget. Can this be used as an alternative to storing global hash index on a fast expensive storage (ssd, nvram)? (Helps throughput, not latency.)
- Our IO is relatively expensive (involves network transfer). Because of this, elimination of extra IO by better de-duplication (better sliding window, *etc*.) is more important than in on-device de-duplication.

THE END

Write with server-side support

```
dedup_write(fid, block_nr, data) {
    dom = dedup_domain_get(fid);
    (loc, ref) = dedup_lookup(dom.hashmap, dom.hash(data));
    obj_attr_add(fid, "DEDUP:" + string(block_nr), loc);
    write(dom.block_store_obj[loc.obj_nr], loc.idx, data);
}
```

- `dedup_lookup()` sends a request to dedup service. This request:
    - Makes lookup in the local hashmap,
    - If hash is not found, allocates a new block location and inserts in the hash map
    - If hash is found, increments the reference counter
- This reduces the network latency

Cost: write with server-side support

	Network			Storage		
	msg	**max**	**byte**	**iop**	**max**	**byte**
`dedup_lookup(...)`	K_M+1	1		logN	logN	4KB
`obj_attr_add(...)`	K_M+1	1	$(K_M+1)*4KB$			
`write(...)`	$f*(K_D+1)$	f	$(K_D+1)*bs$			
Total	$f*(K_D+1)+2K_M+3$	2+f				

- ?

AGENDA

Goals

Implementation

Analysis

Conclusion