Nodalink

TOWARD QCOW2 DEDUPLICATION

Benoît Canet <benoit.canet@nodalink.com>

benoit on #qemu / oftc

KVM-Forum / October 2013

What is deduplication?

- Factorizes redundant storage blocks
- Saves disk space
- Can be combined with block compression
- Saves money
- Reads identical blocks only once (cached)
- Encourages SSD use as SSD price/MB approaches hard drive price/MB

Possible uses

- File server
- Catia CAD software: 5 fold decrease in disk use
- Factorize guest containers without AUFS
- Archival (when combined with compression)

Why QCOW2?

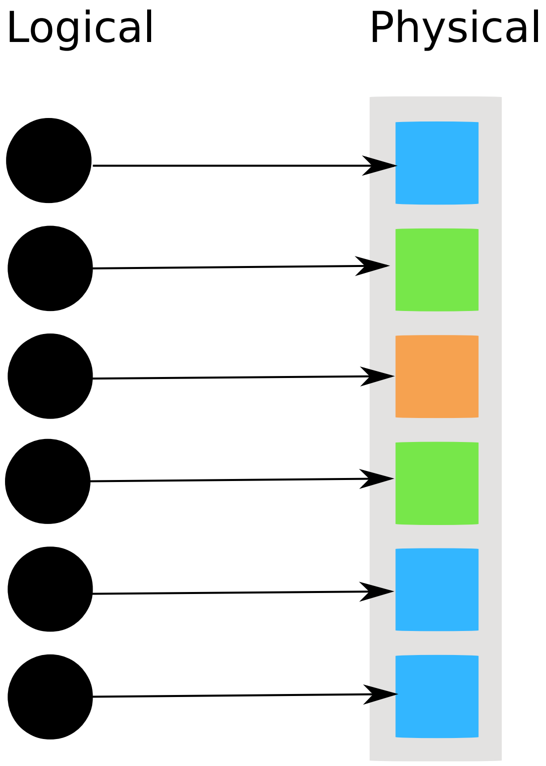
- QEMU code is simpler than kernel code
- QCOW2 has the required infrastructure
- QCOW2 is transparent for the guest
- Could work later over NFS/Gluster/Ceph

How does it work?

- Volume is divided into data blocks
- Use QCOW2 logical to physical mapping
- Identical logical blocks pointing to same physical block
- Use QCOW2 reference count for physical block lifecycle

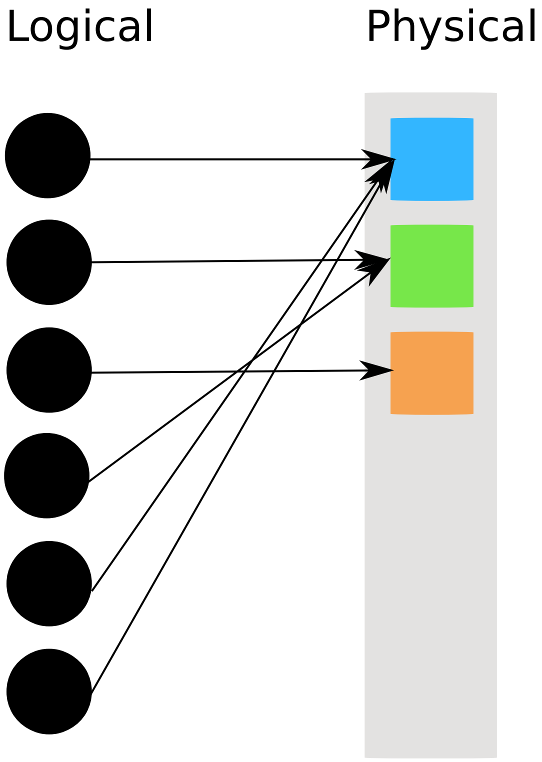
How does it look?

Without dedupe


Logical

Physical

With dedupe


Logical

Physical

First iteration architecture

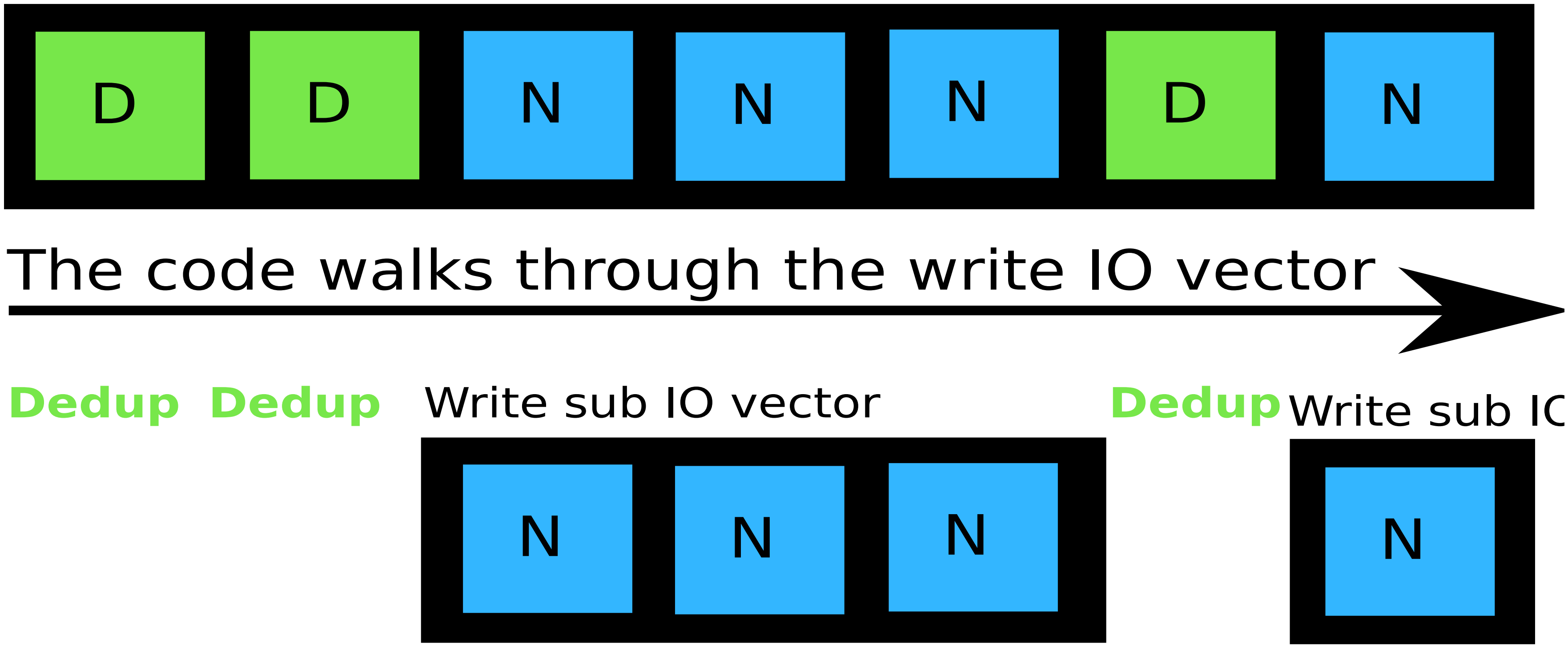
- Use hashes to identify identical blocks
- 256-bit crypto hashes
- Low probability of collision on 1 EB with 4KB clusters: 2.57E-49
- Non-ECC ram bit flip rate: 1.3e-12 upsets/bit/hour
- Manipulate all hashes in an in RAM Gtree
- Save hashes on disk indexed by physical block offset
- Write at 100MB/s on an intel 510 SSD
- QCOW2 read path untouched → Read at full speed

Deduplication algorithm

N = new block

D= duplicated block

Incoming write IO vector



The code walks through the write IO vector

Dedup **Dedup** Write sub IO vector **Dedup** Write sub IO

First iteration shortcomings

- Writes are not at full SSD speed
- Makes random writes
- Crypto hash uses a lot of CPU
- 80 bytes of RAM per 4KB cluster → too much

Second iteration goals

- Building a key-value store into QCOW2
- Need to reduce memory usage
- Need to make memory usage configurable

SSD storage specificity

- Large sequential writes (Speed)
- No random writes (NAND wear-out)
- Can do fast random reads
- Random reads must be done in parallel to go fast
- Limited number of rewrite cycles (3,000)

Hash storage alternatives

- Disk hash table
- B-tree variants
- SILT
- BufferHash
- QCOW2 key value store

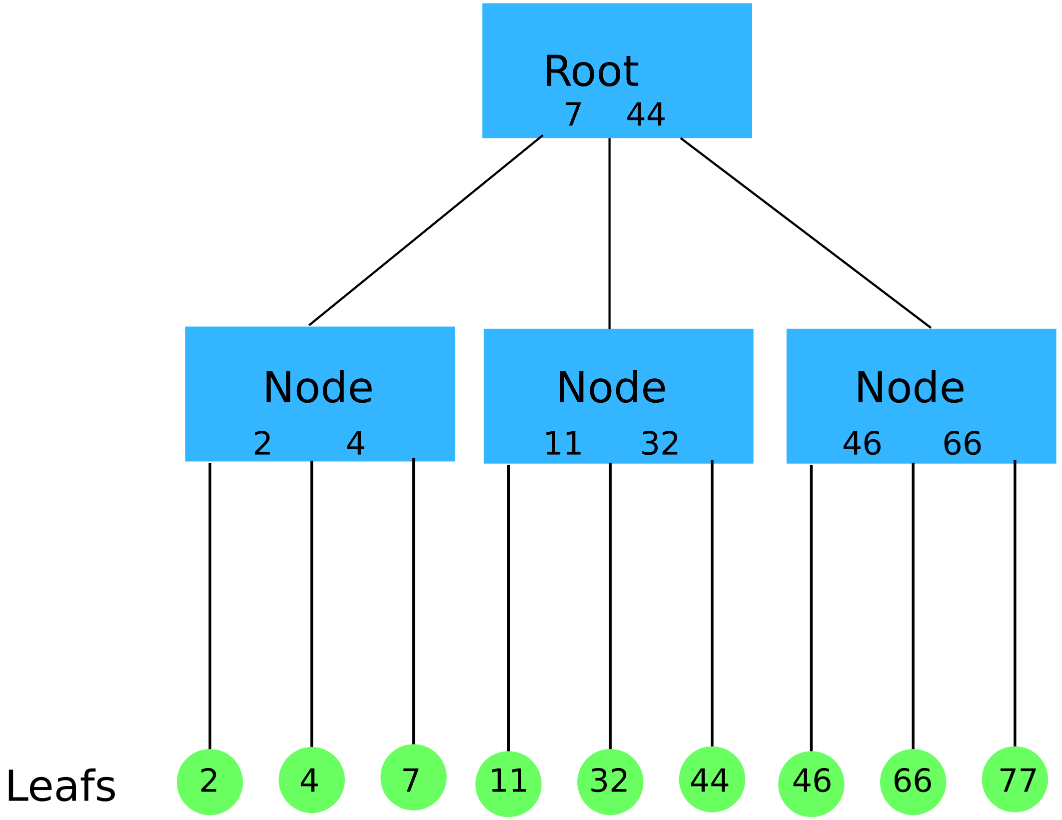
Disk hash table

- A collection of buckets containing hashes

Disk hash table

- Pro: O(1) lookup, O(1) insertion
- Con: Generates lots of random writes
- Con: Sparse hash table is inefficient
- Con: Disk Hash tables don't grow well
- Con: Write amplification

B-tree



B-tree

- Pro: Well known structure (BAYER -1972)
- Con: O(log(n)) lookup not O(1)
- Con: Complex locking protocols
- Con: Generates lots of random writes
- Con: Write amplification

SILT

- SILT is a memory-efficient, high-performance key-value store
- Pro: Made for deduplication needs
- Pro: Made for SSD
- Pro: O(1) lookup
- Pro: Amortized insertions
- Con: complexity → need to simplify

BufferHash

- Another research paper
- Ancestor of SILT
- Pro: Also done for SSD
- Pro: Lots of good ideas
- Combine these two great projects
- Specialize deduplication for SSD usage

QCOW hash store

- Optimized for SSD
- Two simple stages
- Takes only around 4 bytes of RAM per 4KB cluster
- No write amplification
- Amortized writes
- O(1) lookup
- Memory usage can be configurable

Inserting into the hash store

- Insertions use only large sequential writes
- No write amplification

Stage 1

- Write new hashes into a log
- Build a hash table of the new hashes in RAM

Stage 1

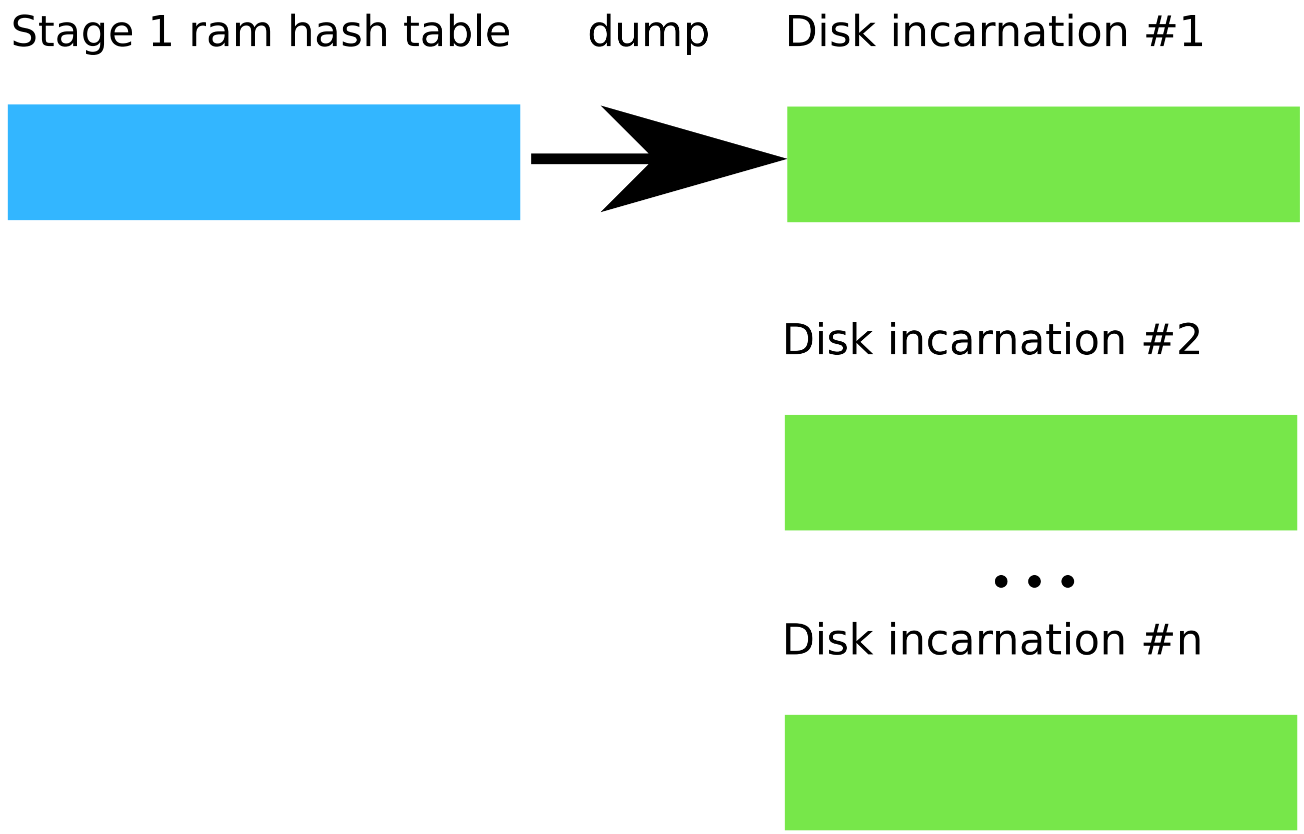
Index into in RAM hash table

Write on disk log: hash table rebuild from it on restart

Stage 2

- Convert Stage 1 hash table into an incarnation
- Collect incarnations

Stage 2


Stage 1 ram hash table

dump

Disk incarnation #1

Disk incarnation #2

• • •

Disk incarnation #n

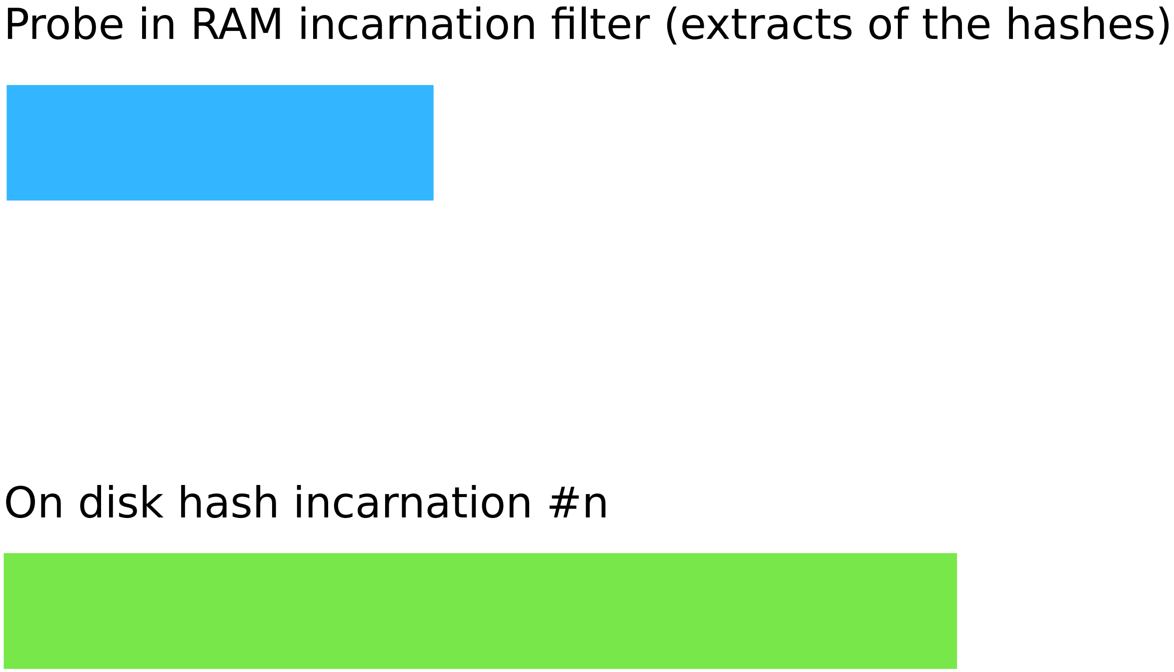
Querying

- First query Stage 1
- Next query every Stage 2 incarnation
- Query from newest to oldest
- Queries can be done in O(1) with RAM filters

How to speed up Stage 2 queries

- One filter per incarnation
- Filters loaded into RAM
- A filter is an extract of an incarnation
- Same as the incarnation, only smaller
- Use smaller hashes at the same position
- Smaller hashes are slices of the hashes

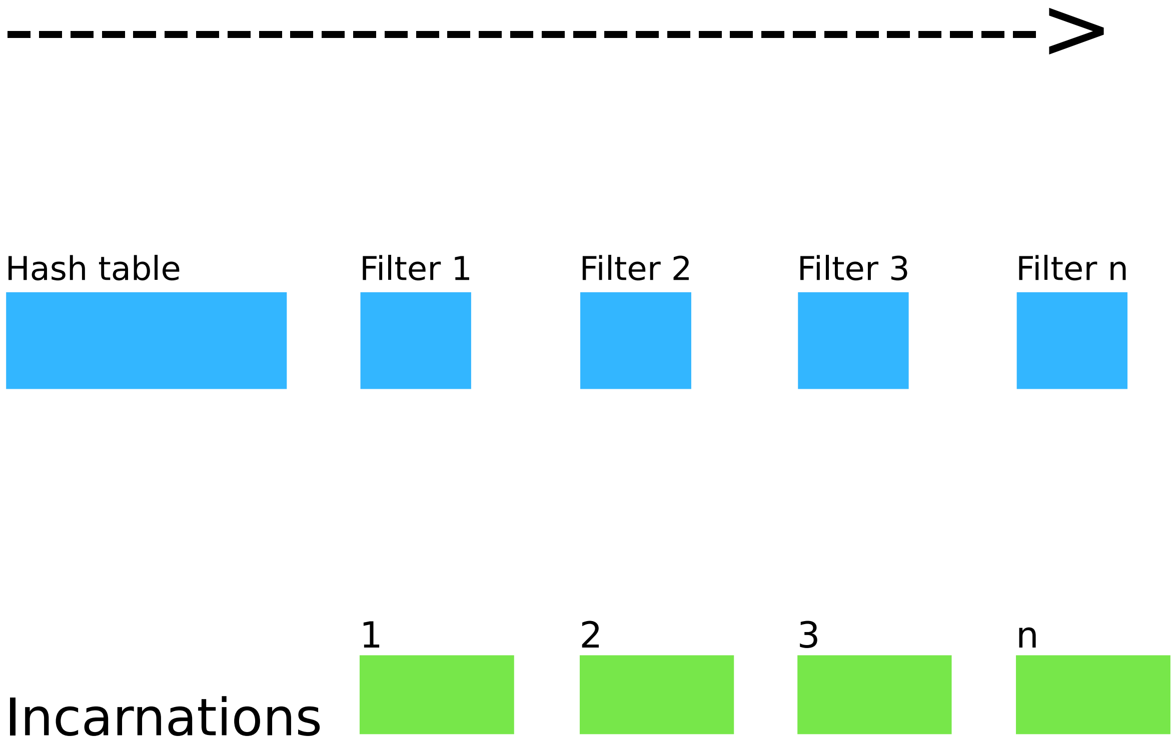
A Stage 2 query probe



Probe in RAM incarnation filter (extracts of the hashes)

On disk hash incarnation #n

Store queries



Hash table

Filter 1

Filter 2

Filter 3

Filter n

Incarnations

1

2

3

n

Memory usage control

- Oldest in RAM filters can be unloaded at will
- Memory usage will decrease
- Only the deduplication ratio will be impacted

Current status

- QCOW2 key-value store implemented
- First round of patches need to be merged

Third iteration (after merge)

- SSDs need parallelization to read fast
- Current algorithm is sequential so it is slow
- Dedupe algorithm code will need a rewrite
- Need a faster 256-bit hash function (cityhash?)

Does it work at all?

Let's do a simple test

Host preparation

- On the host:
- `# qemu-img  create -f qcow2_dedup test.qcow2 10G`
- `# qemu … -drive file=test.qcow2,if=virtio,cache=none`

On the guest

```
root@debian:~# mkfs.ext4 /dev/vdb
mount /dev/vdb /mnt
root@debian:~# du -sh /usr/
927M    /usr/
root@debian:~# cp /usr/ /mnt/1 -a
root@debian:~# cp /usr/ /mnt/2 -a
root@debian:~# cp /usr/ /mnt/3 -a
root@debian:~# cp /usr/ /mnt/4 -a
root@debian:~# du -sh /mnt/
3.6G    /mnt/
root@debian:~# sync
```

Back to the host

-
  ```
  # du -sh  test.qcow2
  1.1GB    test.qcow2
  ```
- 2.5GB of disk space saved on 3.6GB

Sponsor:

OUTSCALE

Contact: benoit.canet@nodalink.com

Questions?

References

- SSD: http://en.wikipedia.org/wiki/Solid-state_drive
- B-tree: www.cs.aau.dk/~simas/aalg06/UbiquitBtree.pdf
- SILT: http://www.cs.cmu.edu/~dga/papers/silt-sosp2011.pdf
- BufferHash: http://pages.cs.wisc.edu/~akella/papers/bufferhash-nsdi10.pdf
- Venti: http://www.cs.bell-labs.com/sys/doc/venti/venti.html