f4: Facebook's Warm BLOB Storage System

Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, Sanjeev Kumar

Presentation by Dominik Murzynowski

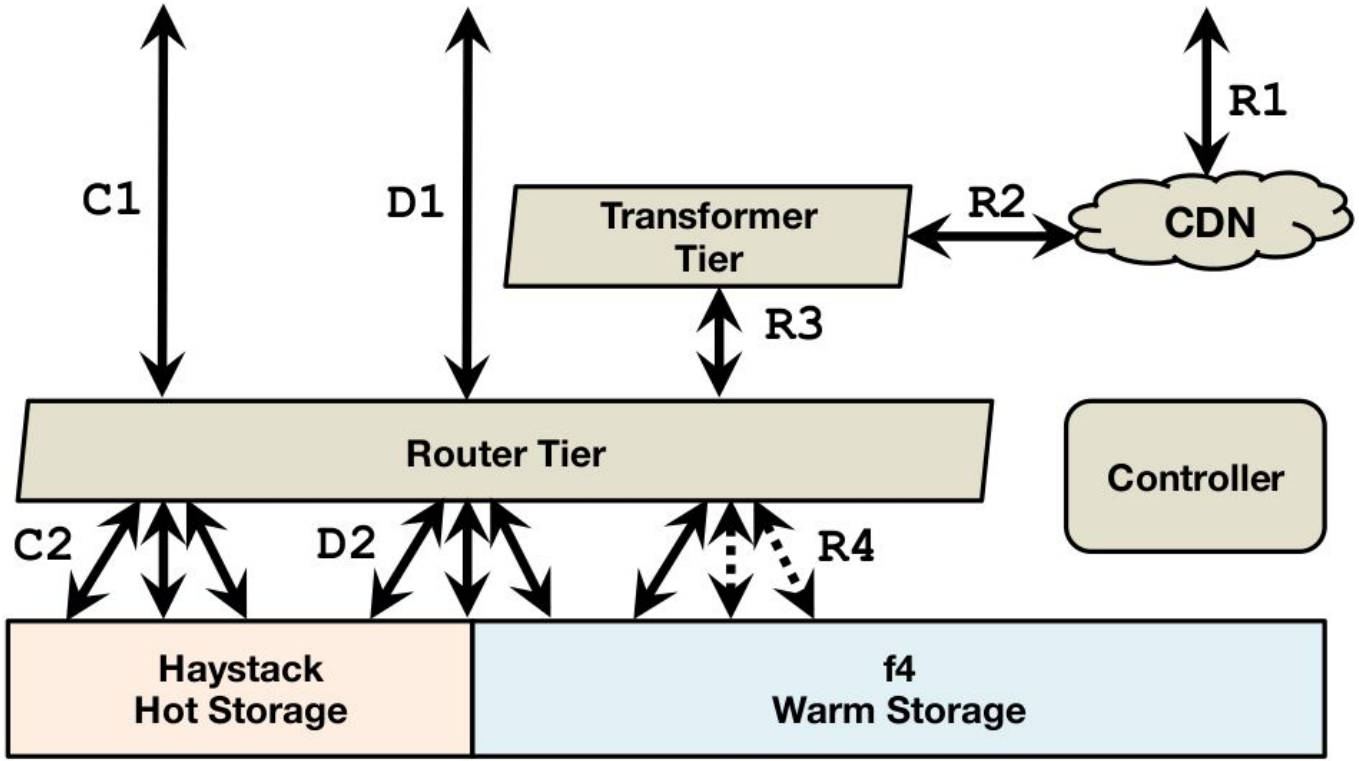
Problem

Most of Facebook’s stored data are Binary Large OBjects (BLOBs).

Haystack, Facebook’s original BLOB storage was found to be inefficient.

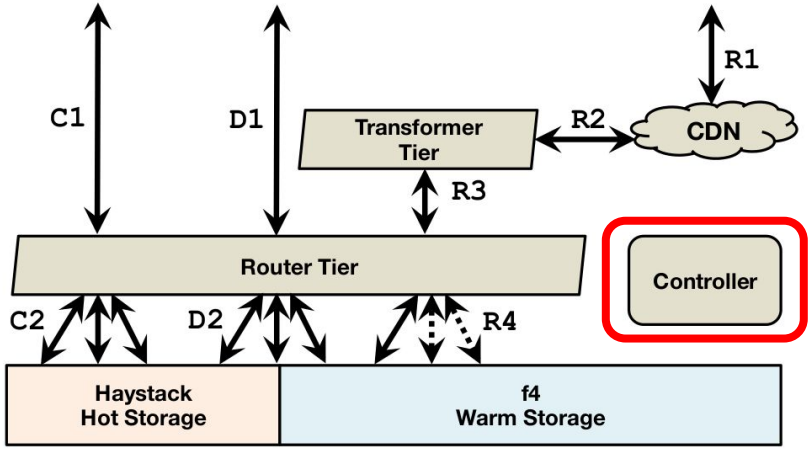
BLOBs are created once, never modified, sometimes deleted and read many times, so the storage engine should provide fast reads.

Facebook's BLOB storage design



Controller

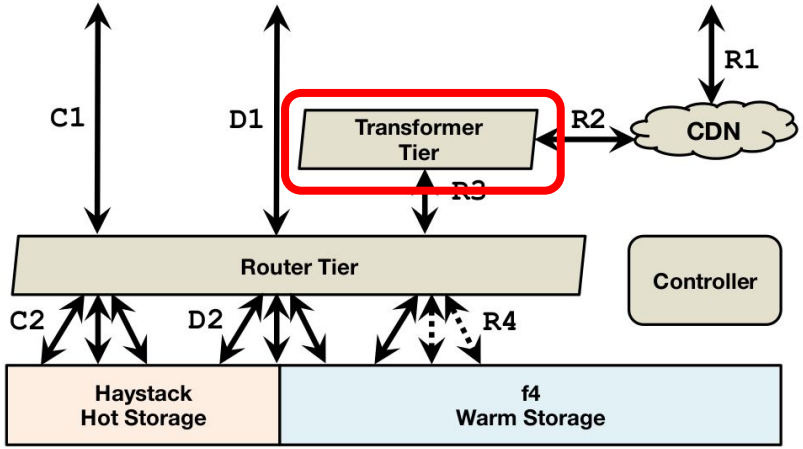
Controller machine ensures smooth functioning of the system.



Transformer Tier

CPU-heavy computational machines.

Handles a set of transformations on BLOBs, like cropping photos, so Storage Tier can focus only on storing data, enabling us to choose more optimal hardware.

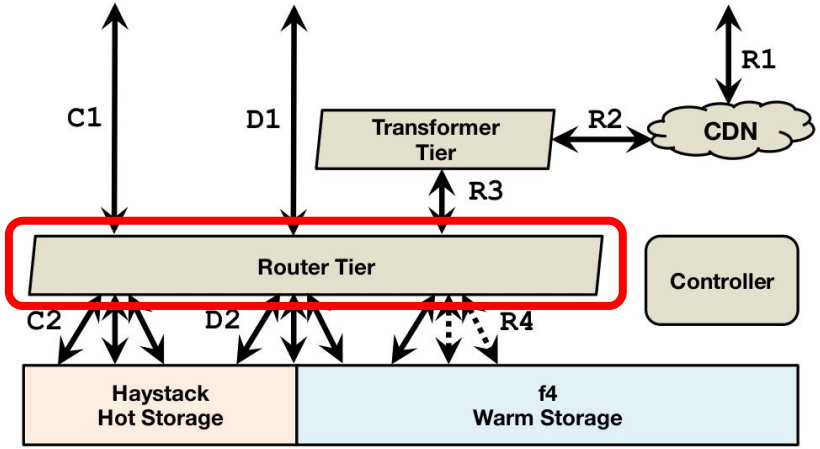


Router Tier

Interface for BLOB storage: hides the implementation of storage from clients, enabling the addition of more storage systems.

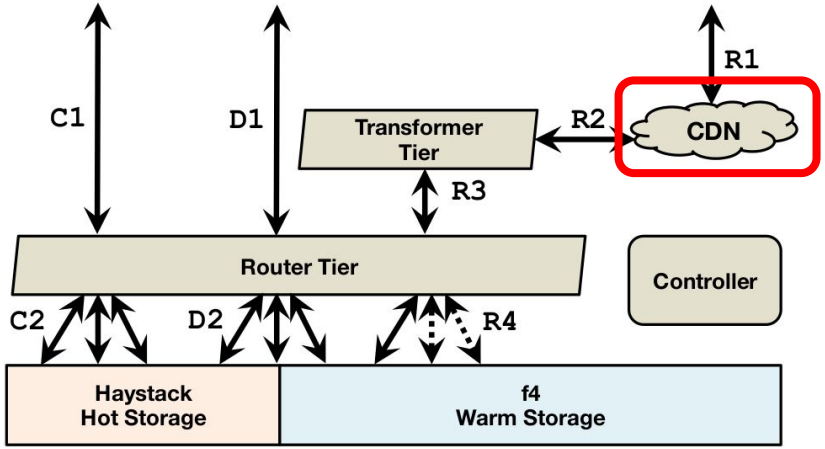
Clients send operations on logical BLOBs.

Router tier machines are identical so the tier can be easily scaled by adding more clones.



Caching stack

Request rates are lowered by checking first if a BLOB is in cache.



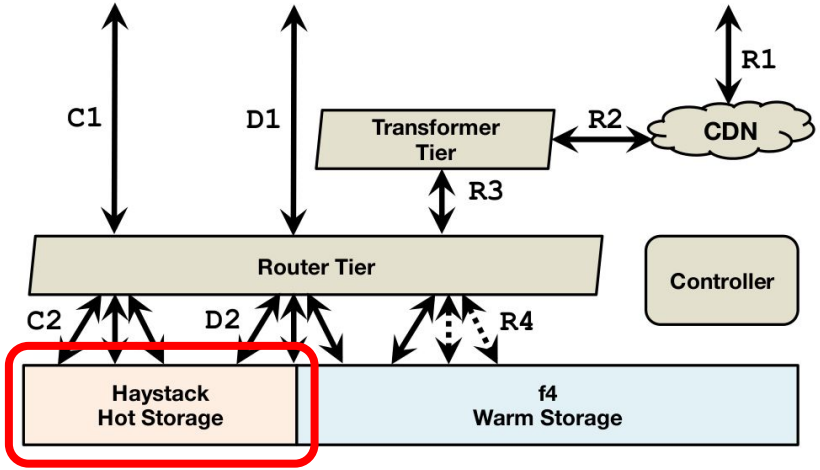
Haystack

Original BLOB storage by Facebook.

Designed to fully utilize IOPS.

High fault tolerance to disk, host, rack and datacenter failure by RAID-6 and triple replication of files.

Replication factor of 3.6x: 3x from file replication and 1.2x from RAID-6 with 12 disks.

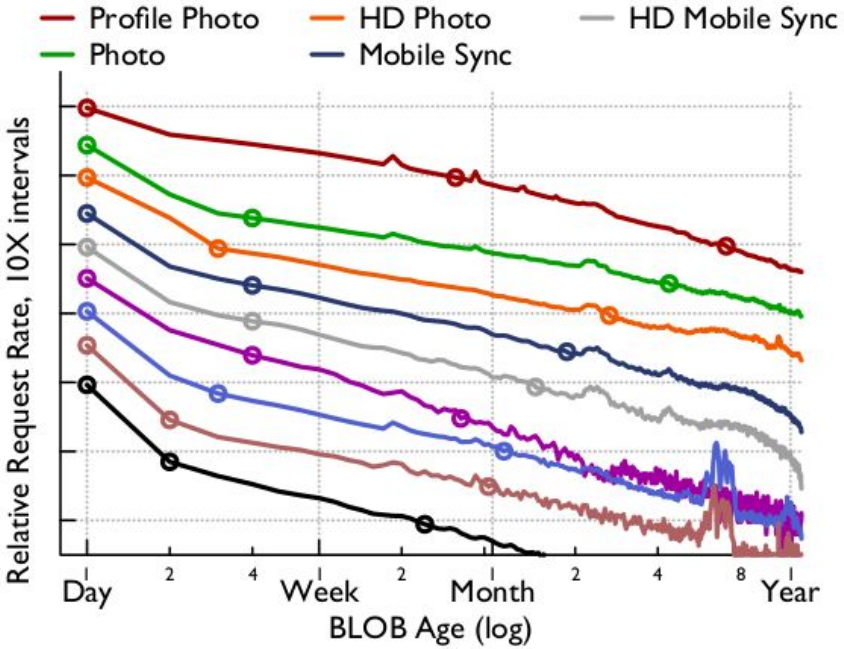


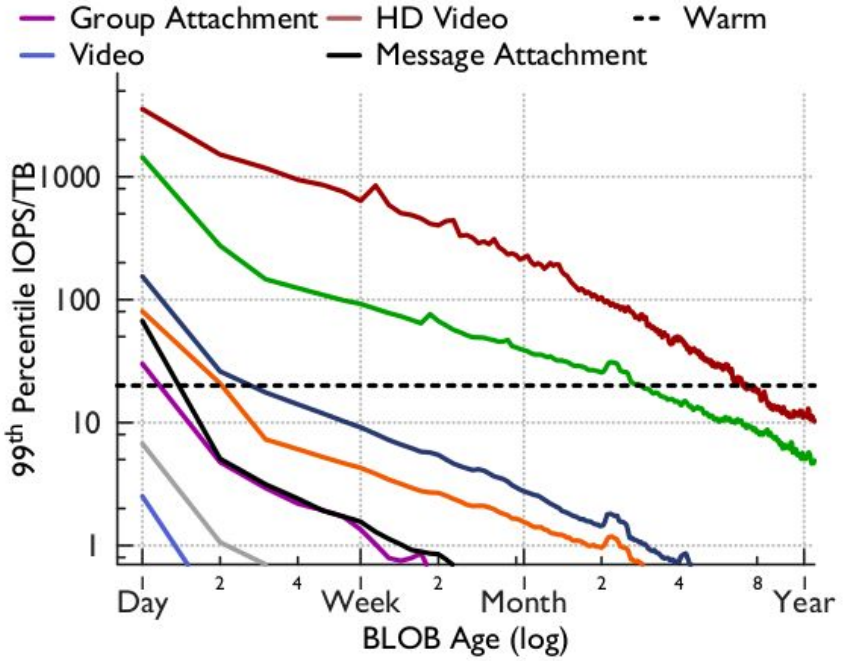
Do we really need full IOPS utilization?

The trade-off of maintaining high utilization and fault tolerance is relatively high replication factor.

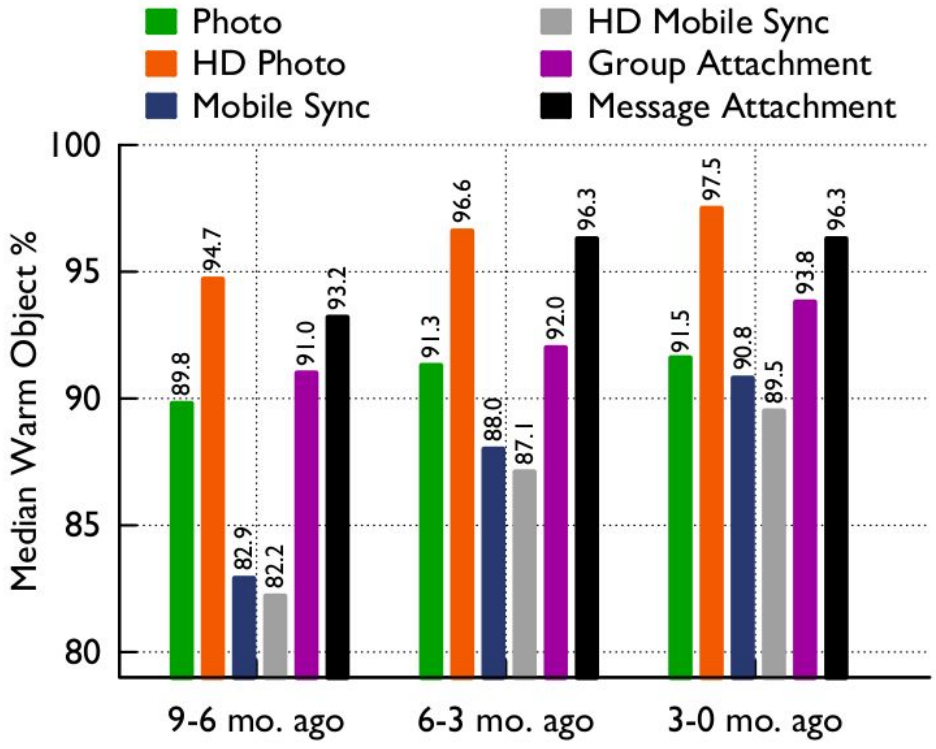
What if we switch to slower system, but enable lower replication factor?

Hot and warm BLOBs





Hot and warm BLOBs



f4 design

System comprised of number of *cells* where each cell lives entirely in one datacenter and is comprised of homogenous software (currently 14 racks of 15 hosts each, with 30 4TB drives per host).

Reed-Solomon(10,4) and XOR coding to provide fault tolerance with replication factor of 2.1: 1.4 from Reed-Solomon coding and 1.5 from XOR.

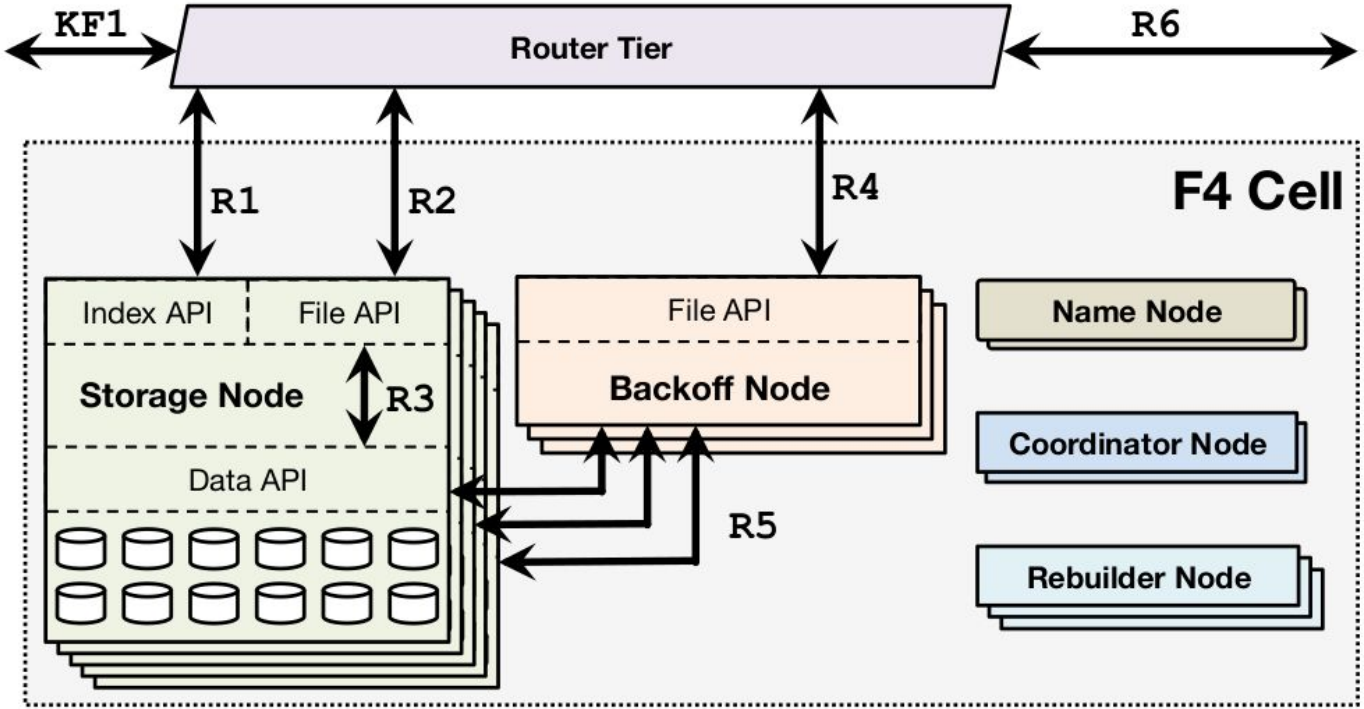
Data structure

Only locked volumes are stored in f4, with read-only index and data and journal removed.

BLOBs are encrypted with per-single BLOB key stored in external database, deletes are managed by removing a key.

Index file is triple-replicated, data files are stored via a Reed-Solomon(10,4) code. Logical volume is split to 10 blocks, 4 parity blocks are calculated and the resulting 14 blocks are called a *stripe*.

Cell architecture

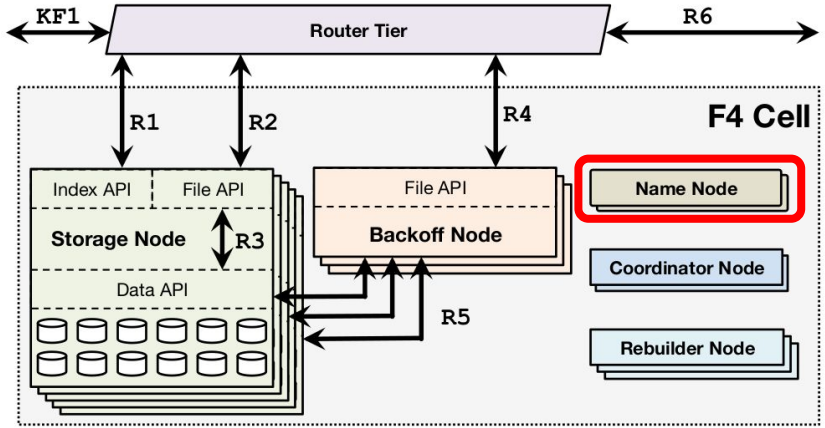


Name node

Maintains mapping between data blocks and their locations in storage nodes.

Mapping is distributed to storage nodes.

Fault tolerance by primary-backup setup.



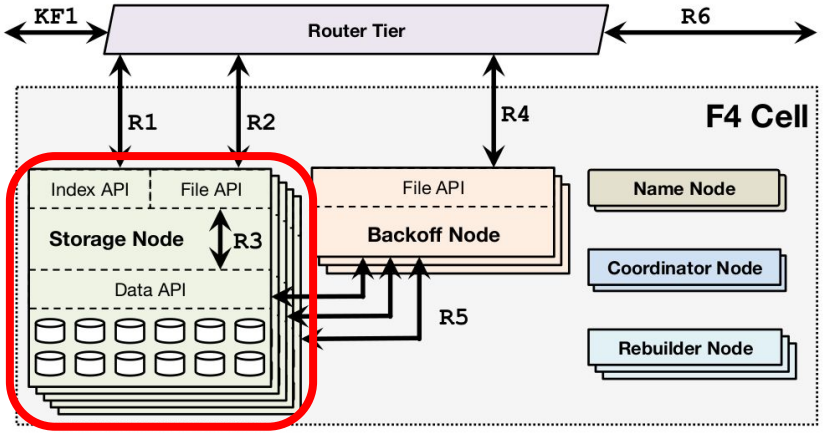
Storage node

Exposes two APIs: an *Index API* providing existence and location information for volumes, and *File API* for access to data.

Reads are normally handled by:

- Getting location of volume with requested file
- Getting file from its standard volume location

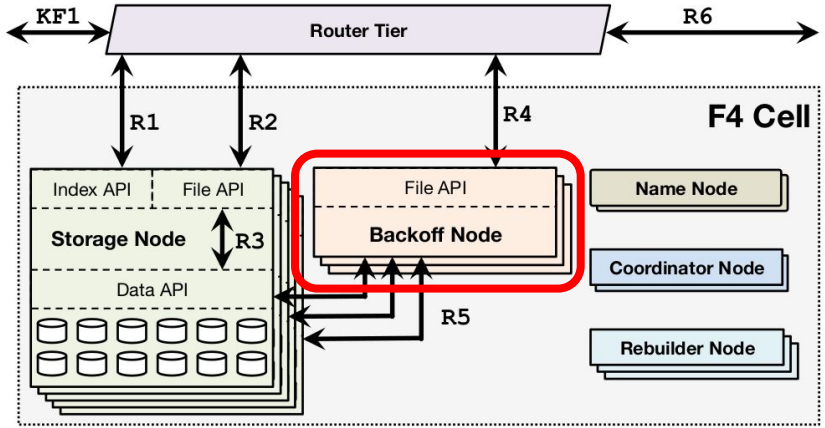
If node fails, then another workflow using backoff nodes is used.



Backoff node

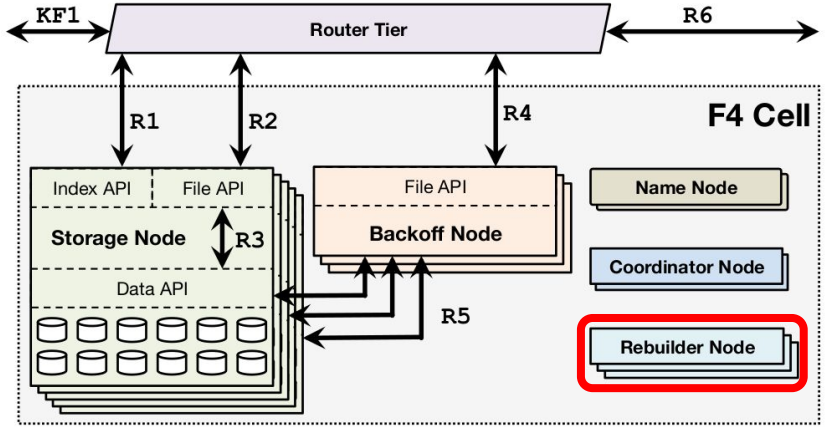
In case of storage node fail, request is redirected to backoff node with the same information about volume location (data file, offset and length).

The node loads companion data from other blocks in stripe and reconstructs requested BLOB online.



Rebuilder nodes

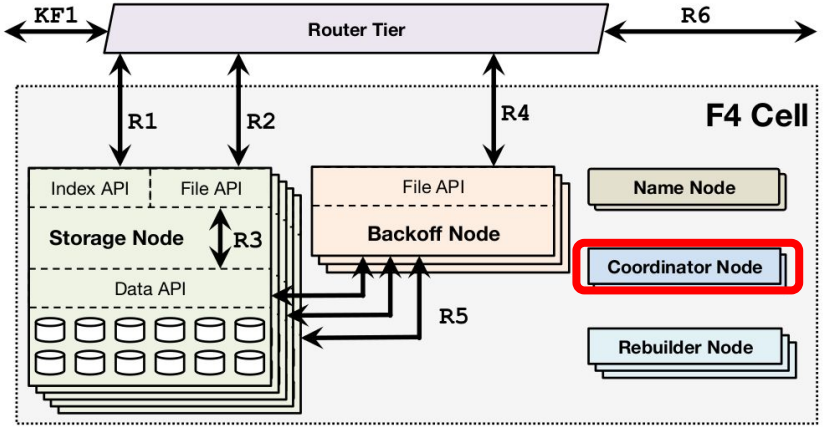
Storage-less, CPU-heavy machines handling failure detection and data reconstruction from parity blocks of stripe.



Coordinator node

Schedules maintenance tasks such as data rebuilding and ensuring that chances of data unavailability is minimised.

Resolves *violations* when two blocks of a stripe are on the same rack and rebalances load between storage nodes.



Maintaining fault tolerance

Primary concern (as this happens most often) is a rack fail.

Stripe blocks are initially distributed on many racks, so that two different blocks are on different racks.

Additionally, sometimes whole datacenter fail can occur. We pair cells and for each pair, a datacenter with XOR of data blocks from the cells is created.

Maintaining fault tolerance

As stated before, if disk, host or rack fails, a backoff node is used to recreate BLOB online from companion and parity blocks.

If datacenter fails, the buddy and XOR datacenters are used to recreate BLOB online on XOR datacenter.

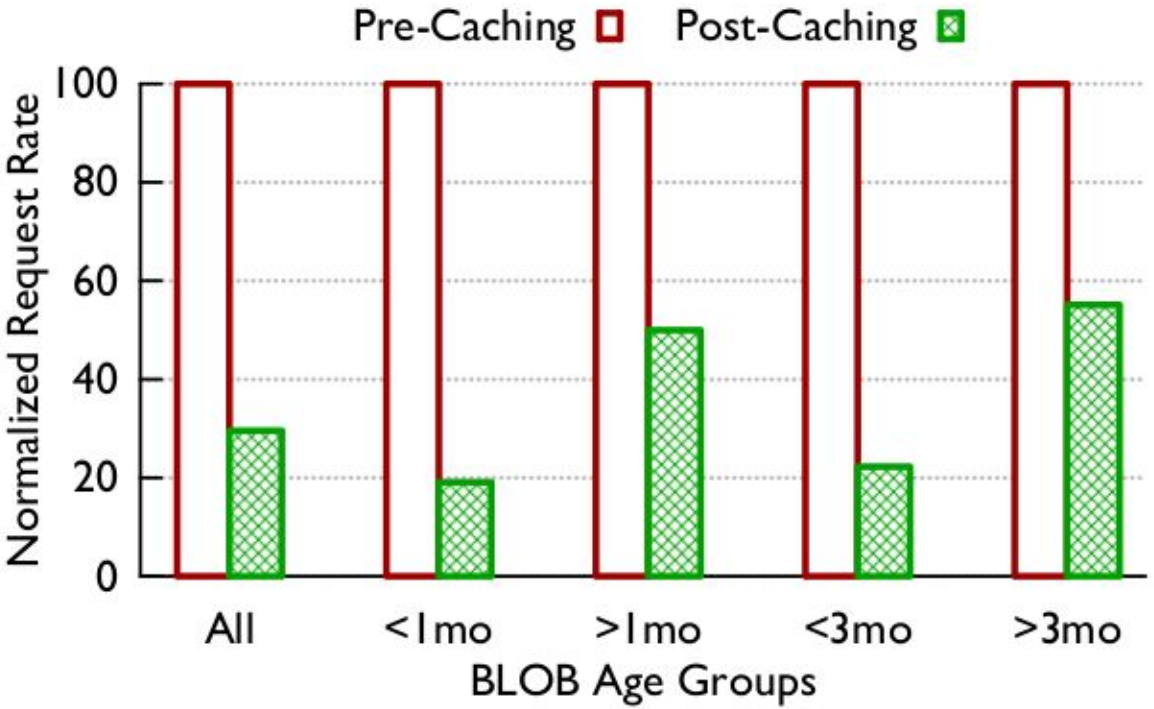
The fault tolerance of f4 is even bigger than Haystack’s despite lower replication factor.

Everybody needs friends

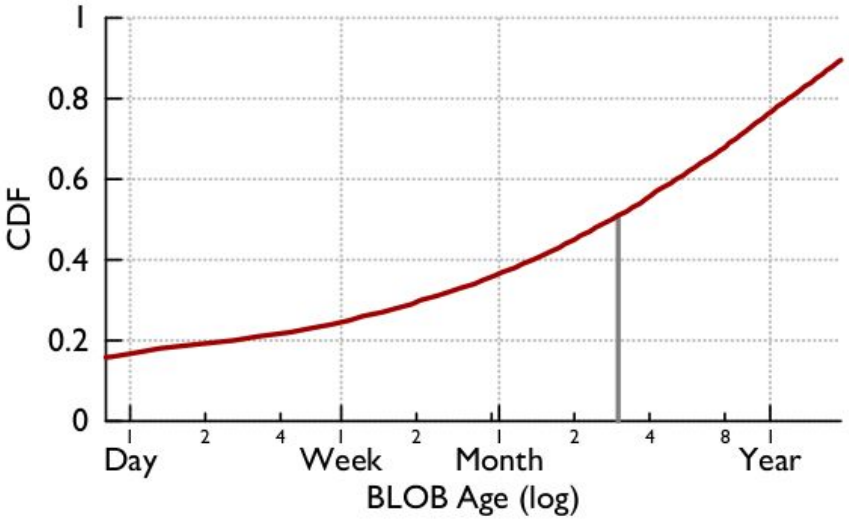
The idea of lowering the replication factor sounds nice, but can we easily lower the throughput without it affecting clients?

f4 is only a part of storage architecture of Facebook and other modules enable concept of warm storage.

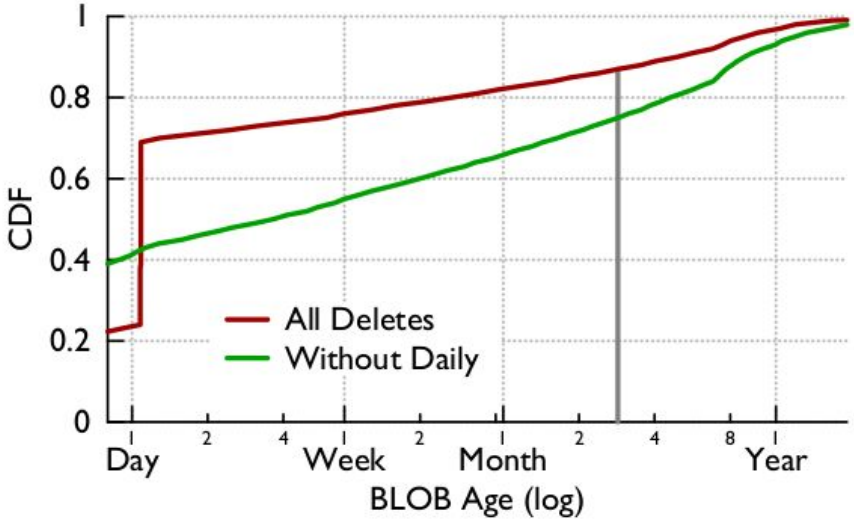
Everybody needs friends: cache



Everybody needs friends: Haystack



(b) CDF of age of BLOB reads.



(c) CDF of age of BLOB deletes.

Performance

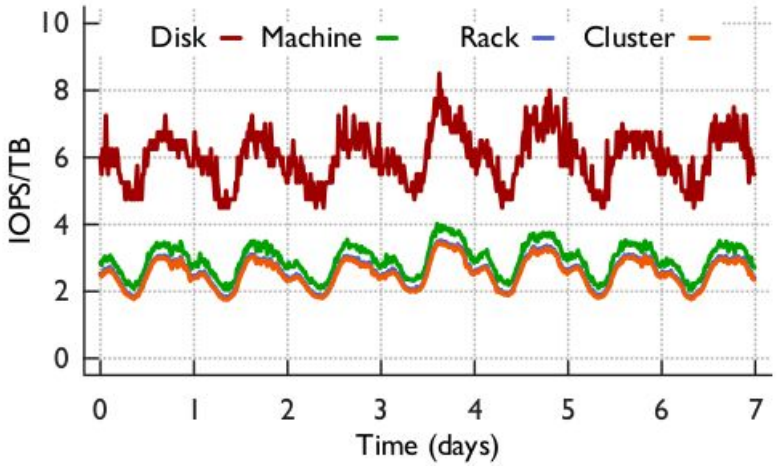


Figure 12: Maximum request rates over a week to f4's most loaded cluster.

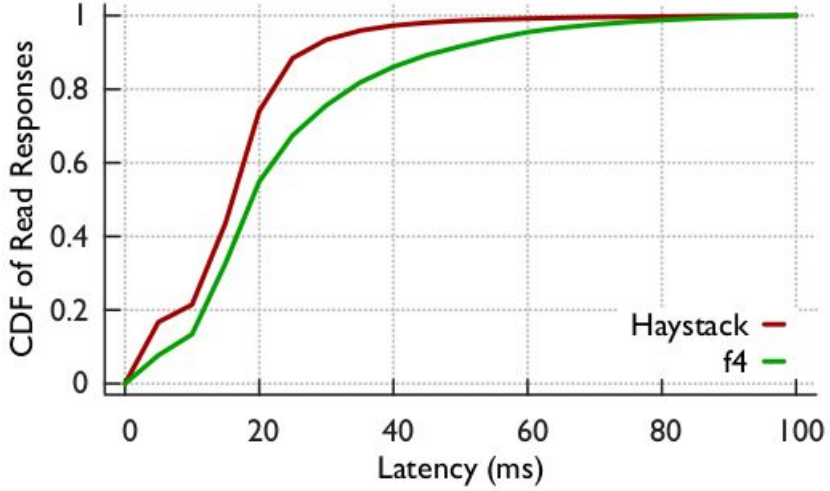


Figure 13: CDF of local read latency for Haystack/f4.

More about fault tolerance

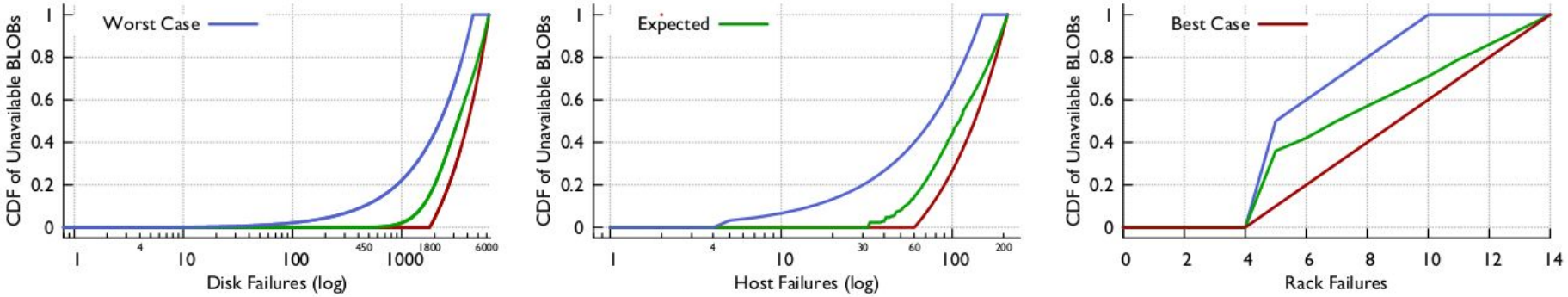


Figure 14: Fraction of unavailable BLOBs for a f4 cell with N disk, host, and rack failures.

What did we want to achieve?

All this work was towards saving space. It’s high time to calculate how much we saved out of 65 PB of warm data.

First, remember that we don’t actually remove physical BLOBs in f4, only handles to them. The measured space of deleted data is 6.8%.

Reduction = (3.6 - 2.1 * 1 / (1 - 0.068)) * 65 PB = 87.5 PB

Experiences

“An early version of f4 used journal files to track deletes in the same way that Haystack does. This single read-write file was at odds with the rest of the f4 design, which is read-only.”

“In particular, HDFS used a thread for each parallel network IO request and Java’s multithreading did not scale well to a large number of parallel requests, which resulted in an increasing back-log of network IO requests. We worked around this with a two-part read, described in Section 5.3, that avoids proxying the read through HDFS.”

www.cs.princeton.edu/~wlloyd/papers/f4-osdi14.pdf

Figures in presentation are from linked paper.