The basics of file systems

Today we are going to discuss Copy-on-Write FSs

ZFS (Zettabyte File System) is a file system originally developed for Sun Solaris.

WAFL (Write Anywhere File Layout) is a file system used by NetApp.

Both of them combine a RAID implementation, a volume manager, and a FS proper.

Requirements

1. A FS must always be consistent.
   - Often, we implement this with journaling or with ideas like Log-structured FS. Workloads that modify a lot of FS metadata suffer from write amplification introduced by the journal. Their writes are doubled.
   - How does one boot from a volume that was not cleanly unmounted?

Requirements

1. A FS must always be consistent.
2. FSCK must be fast. Better yet, there should be no FSCK.
    - Even reading a 10Tb HDD sequentially takes nearly a day. It not acceptable for a reboot to take that long.

Requirements

1. A FS must always be consistent.
2. FSCK must be fast. Better yet, there should be no FSCK.
3. Writes to files must be fast. Metadata-heavy workloads must be fast, too.
    - A journaled FS confirms metadata changes once they are written to the journal. Writes to the journal are sequential and fast enough. However, once the journal overflows, the speed of metadata operations degrades hugely because they need to wait for journal checkpoints, and checkpoints may produce a lot of random IO.

Requirements

1. A FS must always be consistent.
2. FSCK must be fast. Better yet, there should be no FSCK.
3. Writes to files must be fast. Metadata-heavy workloads must be fast, too.
4. It must be easy to use multiple disks for redundancy and speed, and it must be possible to add and replace disks online.
   - It is not always possible to use `lvextend` and `resizefs` to grow a FS online. For example, ext4 supports online growfs only in Linux >= 3.3.
   - Adding a disk to a mdadm-managed RAID6 array needs a full rebuild of the array.

Requirements

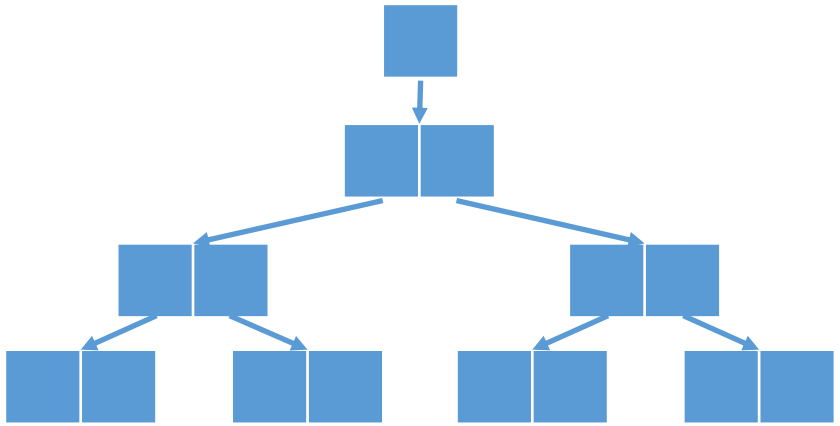
1. A FS must always be consistent.
2. FSCK must be fast. Better yet, there should be no FSCK.
3. Writes to files must be fast. Metadata-heavy workloads must be fast, too.
4. It must be easy to use multiple disks for redundancy and speed, and it must be possible to add and replace disks online..
5. A FS must implement fast snapshots and clones, and must be able to roll back to a previous snapshot. There is a multitude of use-cases that benefit from these features:
   - Containers,
   - Backups,
   - Safe system updates and rolling back after a failed update,
   - Snapshots of directories can’t be implemented at the volume manager level.

Requirements

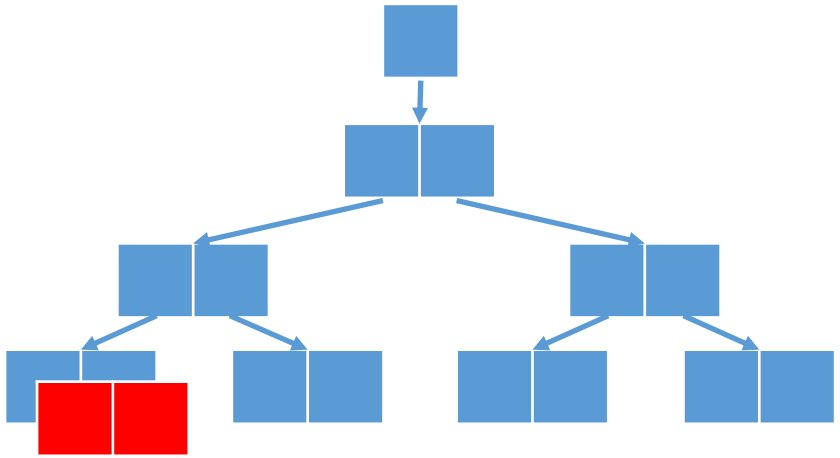
1. A FS must always be consistent.
2. FSCK must be fast. Better yet, there should be no FSCK.
3. Writes to files must be fast. Metadata-heavy workloads must be fast, too.
4. It must be easy to use multiple disks for redundancy and speed, and it must be possible to add and replace disks online.
5. A FS must implement fast snapshots and clones, and must be able to roll back to a previous snapshot.
6. A FS must protect itself from the bit rot (accidental corruption of disk sectors). When doing RAID, it must avoid RAID write holes.
   - Ext4 and XFS may read garbage from disks (cf. the CERN experiment on bit rot in disks).

Idea: copy-on-write transactions (ZFS and WAFL)

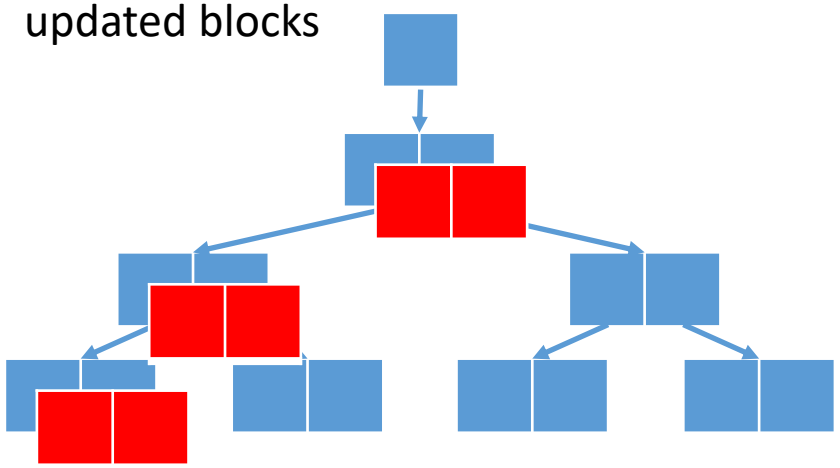
0. The original tree



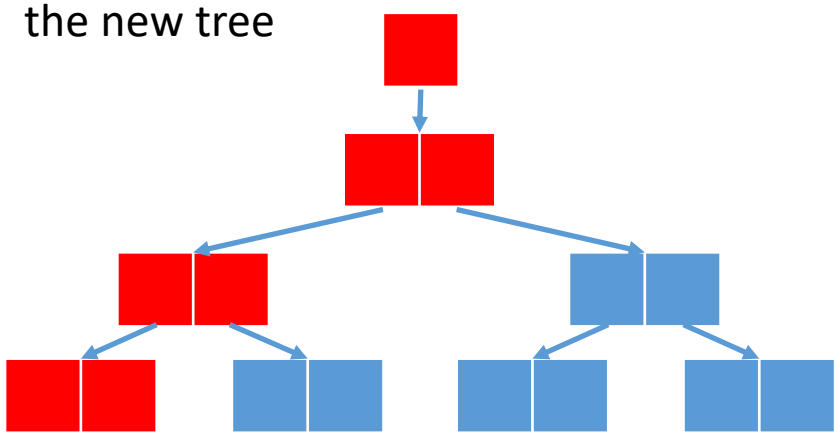
1. Make copies of modified blocks



2. Make copies of blocks that reference updated blocks



3. Overwrite the superblock with the pointer to the new tree



How does copy-on-write implement various requirements that we've added

A FS must always be consistent.	The superblock always points to a consistent tree.
FSCK must be fast. Better yet, there should be no FSCK.	FSCK is not needed.

How does copy-on-write implement various requirements that we've added

A FS must always be consistent.	The superblock always points to a consistent tree.
FSCK must be fast. Better yet, there should be no FSCK.	FSCK is not needed.
Writes to files must be fast. Metadata-heavy workloads must be fast, too.	Copied blocks may be allocated sequentially. Updates to different files may also be written sequentially. **Quiz:** how do we avoid file fragmentation and keep reads from files fast?

How does copy-on-write implement various requirements that we've added

A FS must always be consistent.	The superblock always points to a consistent tree.
FSCK must be fast. Better yet, there should be no FSCK.	FSCK is not needed.
Writes to files must be fast. Metadata-heavy workloads must be fast, too.	Copied blocks may be allocated sequentially. Updates to different files may also be written sequentially. **Quiz:** how do we avoid file fragmentation and keep reads from files fast?
It must be easy to use multiple disks for redundancy and speed, and it must be possible to add and replace disks online.	

How does copy-on-write implement various requirements that we've added

A FS must always be consistent.	The superblock always points to a consistent tree.
FSCK must be fast. Better yet, there should be no FSCK.	FSCK is not needed.
Writes to files must be fast. Metadata-heavy workloads must be fast, too.	Copied blocks may be allocated sequentially. Updates to different files may also be written sequentially. **Quiz:** how do we avoid file fragmentation and keep reads from files fast?
It must be easy to use multiple disks for redundancy and speed, and it must be possible to add and replace disks online.	
A FS must implement fast snapshots and clones, and must be able to roll back to a previous snapshot.	Creating a snapshot is free. Just do not remove the previous version of the superblock and the tree that it points to. Rolling back to a snapshot is trivial, too.

How does copy-on-write implement various requirements that we've added

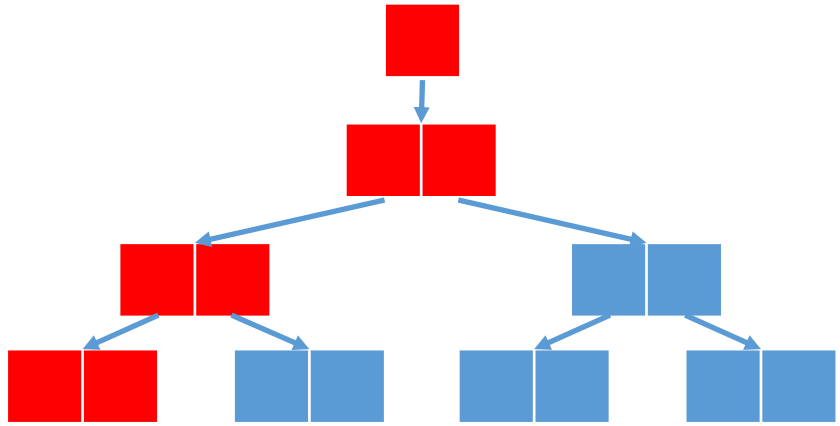
A FS must always be consistent.	The superblock always points to a consistent tree.
FSCK must be fast. Better yet, there should be no FSCK.	FSCK is not needed.
Writes to files must be fast. Metadata-heavy workloads must be fast, too.	Copied blocks may be allocated sequentially. Updates to different files may also be written sequentially. **Quiz:** how do we avoid file fragmentation and keep reads from files fast?
It must be easy to use multiple disks for redundancy and speed, and it must be possible to add and replace disks online.	
A FS must implement fast snapshots and clones, and must be able to roll back to a previous snapshot.	Creating a snapshot is free. Just do not remove the previous version of the superblock and the tree that it points to. Rolling back to a snapshot is trivial, too.
A FS must protect itself from the bit rot.	
When doing RAID, it must avoid RAID write holes.	RAID write hole may be created when blocks are overwritten. A copy-on-write FS never overwrites blocks, hence it cannot create write holes.

Flash memory and SMR drives

Checkpointing changes to a COW FS produces long sequential writes. These are very friendly towards

1. Flash memory where one can overwrite whole erase blocks,
2. SMR disks. These are disks that are split into areas that can be erased and appended to but not overwritten.

See also: https://www.anandtech.com/show/15959/nvme-zoned-namespaces-explained



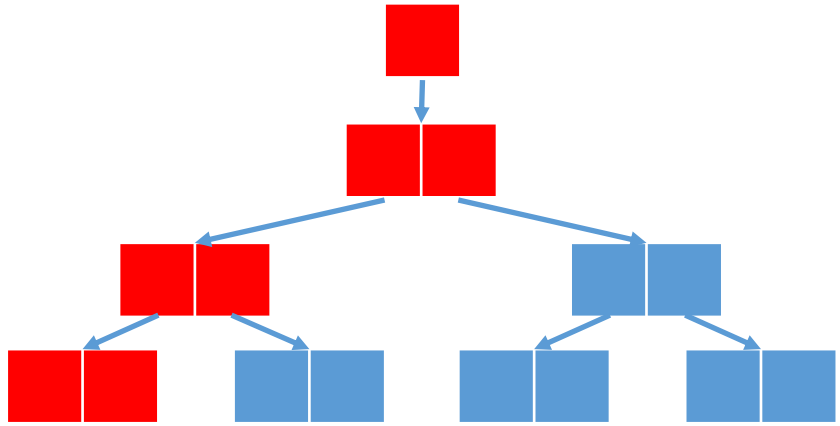
Flash memory and SMR drives

Checkpointing changes to a COW FS produces long sequential writes. These are very friendly towards

1. Flash memory where one can overwrite whole erase blocks,
2. SMR disks. These are disks that are split into areas that can be erased and appended to but not overwritten.

See also: https://www.anandtech.com/show/15959/nvme-zoned-namespaces-explained

There is problem, though: to reuse an erase block or a zone of an SMR disk, one needs to free all FS blocks in that erase block. There needs to be a GC process.



Storage pool allocator

ZFS has a built-in volume manager. At the top level, a ZFS pool has a number of vdevs (virtual devices) that receive writes in a striped fashion. When ZFS needs to allocate space, SPA chooses a vdev to allocate that space from. The choice of a vdev depends on:

1. available free space,
2. available IOPS of the vdev.

Storage pool allocator

ZFS has a built-in volume manager. At the top level, a ZFS pool has a number of vdevs (virtual devices) that receive writes in a striped fashion. When ZFS needs to allocate space, SPA chooses a vdev to allocate that space from. The choice of a vdev depends on:

1. available free space,
2. available IOPS of the vdev.

Note: if the volume manager is separate from a FS, then the FS has no way to know how FS blocks map to disks blocks. Thus, it has no way to balance the IOPS between disks.

Note: this lack of information harms all abstraction levels. For example, if md detects a parity error in a RAID stripe, it cannot know the reason. Is it a data block that got corrupted, or a parity block?

Storage pool allocator

ZFS has a built-in volume manager. At the top level, a ZFS pool has a number of vdevs (virtual devices) that receive writes in a striped fashion. When ZFS needs to allocate space, SPA chooses a vdev to allocate that space from. The choice of a vdev depends on:

1. available free space,
2. available IOPS of the vdev.

A vdev may be:

1. a single disk,
2. a mirror of disks (or other vdevs),
3. RAID5 or RAID6 of disks (or other vdevs).

Storage pool allocator

ZFS has a built-in volume manager. At the top level, a ZFS pool has a number of vdevs (virtual devices) that receive writes in a striped fashion. When ZFS needs to allocate space, SPA chooses a vdev to allocate that space from. The choice of a vdev depends on:

1. available free space,
2. available IOPS of the vdev.

A vdev may be:

1. a single disk,
2. a mirror of disks (or other vdevs),
3. RAID5 or RAID6 of disks (or other vdevs).

Note: it is possible to create arbitrary trees of vdevs but that is not really used. One typically configures a list of mirrors or a list of raidzs.

Storage pool allocator

ZFS has a built-in volume manager. At the top level, a ZFS pool has a number of vdevs (virtual devices) that receive writes in a striped fashion. When ZFS needs to allocate space, SPA chooses a vdev to allocate that space from. The choice of a vdev depends on:

1. available free space,
2. available IOPS of the vdev.

A vdev may be:

1. a single disk,
2. a mirror of disks (or other vdevs),
3. RAID5 or RAID6 of disks (or other vdevs).

For the purposes of space allocation, each top-level vdev is presented as a number of slab allocators. ZFS calls them metaslabs.

Storage pool allocator

ZFS has a built-in volume manager. At the top level, a ZFS pool has a number of vdevs (virtual devices) that receive writes in a striped fashion. When ZFS needs to allocate space, SPA chooses a vdev to allocate that space from. The choice of a vdev depends on:

1. available free space,
2. available IOPS of the vdev.

A vdev may be:

1. a single disk,
2. a mirror of disks (or other vdevs),
3. RAID5 or RAID6 of disks (or other vdevs).

For the purposes of space allocation, each top-level vdev is presented as a number of slab allocators. ZFS calls them metaslabs.

Reminder: a slab allocator is just an array of fixed-size blocks (slabs) that can allocate and free those blocks. ZFS uses slabs that range from 512B to 128K.

Note: compare this to golang's per-size class mspans. Are there similar mechanisms in the JVM?

Storage pool allocator

ZFS has a built-in volume manager. At the top level, a ZFS pool has a number of vdevs (virtual devices) that receive writes in a striped fashion. When ZFS needs to allocate space, SPA chooses a vdev to allocate that space from. The choice of a vdev depends on:

1. available free space,
2. available IOPS of the vdev.

A vdev may be:

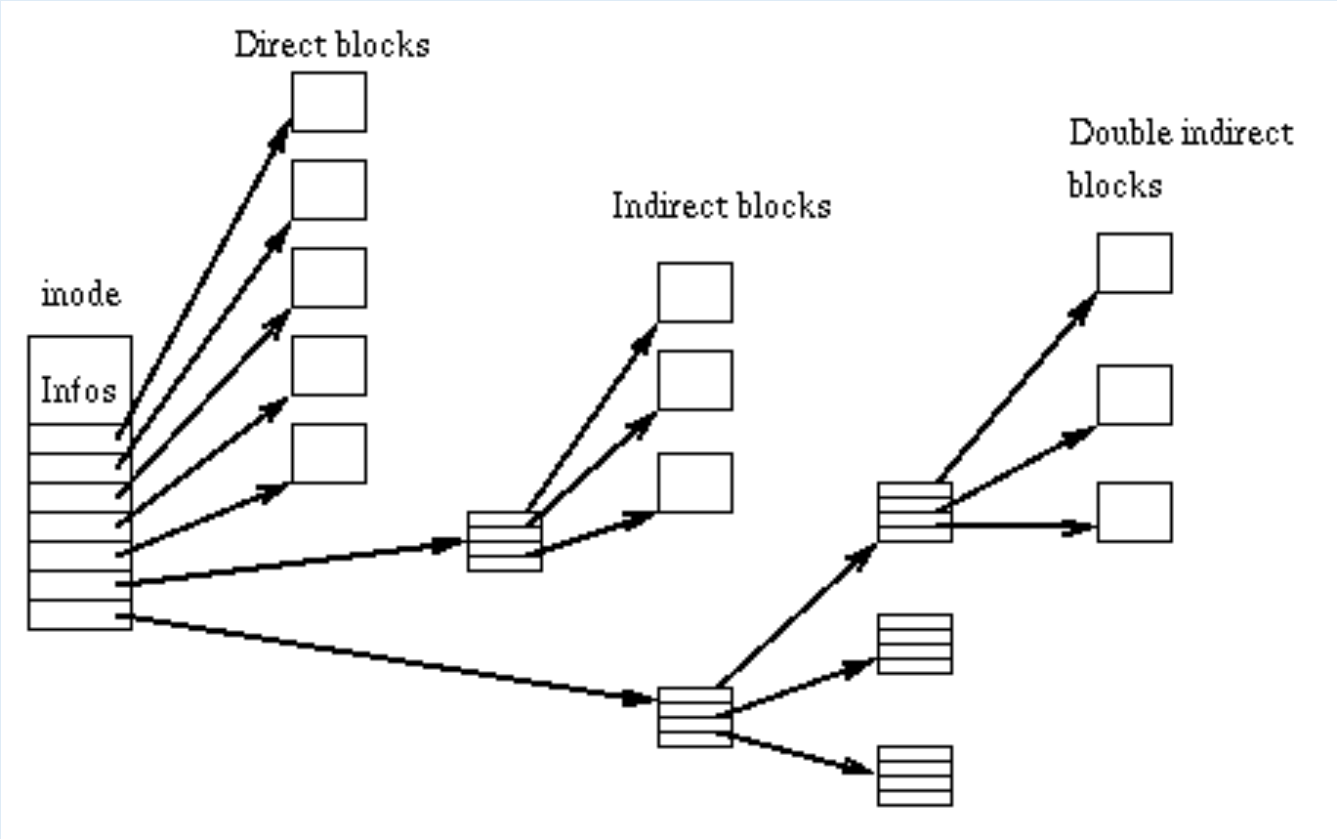
1. a single disk,
2. a mirror of disks (or other vdevs),
3. RAID5 or RAID6 of disks (or other vdevs).

For the purposes of space allocation, each top-level vdev is presented as a number of slab allocators. ZFS calls them metaslabs.

Reminder: it is useful to regard the storage pool allocator as `malloc()/free()` for the disk space.

Data management unit

Reminder: in ext2, the content of a file is referenced by a variable-height tree of direct and indirect pointers. Blocks pointers are plain integers and may point anywhere on the disk, even to FS metadata regions.



Data management unit

In ZFS, block pointers are arranged as a perfectly balanced tree. They also facilitate many consistency checks.

ZFS block pointer

vdev0					pad	asize	
G	offset0						
vdev1					pad	asize	
G	offset1						
vdev2					pad	asize	
G	offset2						
BDX	lvl	type	cksum	E	comp	psize	lsize
spare							
spare							
physical birth time							
logical birth time							
fill count							
checksum[0]							
checksum[1]							
checksum[2]							
checksum[3]							

Data management unit

In ZFS, block pointers are arranged as a perfectly balanced tree. They also facilitate many consistency checks.

ZFS block pointer

vdev0					pad	asize	
G	offset0						
vdev1					pad	asize	
G	offset1						
vdev2					pad	asize	
G	offset2						
BDX	**lvl**	**type**	cksum	E	comp	psize	lsize
spare							
spare							
physical birth time							
logical birth time							
fill count							
checksum[0]							
checksum[1]							
checksum[2]							
checksum[3]							

- `type` holds the type of a block being pointed to,
- `lvl` is the level of indirection,
- `fill_count` is the number of non-zero blocks in the subtree being pointed to.

Data management unit

In ZFS, block pointers are arranged as a perfectly balanced tree. They also facilitate many consistency checks.

ZFS block pointer

vdev0					pad	asize	
G	offset0						
vdev1					pad	asize	
G	offset1						
vdev2					pad	asize	
G	offset2						
BDX	lvl	type	cksum	E	**comp**	**psize**	**lsize**
spare							
spare							
physical birth time							
logical birth time							
fill count							
checksum[0]							
checksum[1]							
checksum[2]							
checksum[3]							

- `lsize` is the logical size of a block being pointed to,
- `psize` is the physical size of a block being pointed to (may be smaller if the block is compressed),
- `comp` chooses the compression algorithm.

Data management unit

In ZFS, block pointers are arranged as a perfectly balanced tree. They also facilitate many consistency checks.

ZFS block pointer

vdev0					pad	asize	
G	offset0						
vdev1					pad	asize	
G	offset1						
vdev2					pad	asize	
G	offset2						
BDX	lvl	type	**cksum**	E	comp	psize	lsize
spare							
spare							
physical birth time							
logical birth time							
fill count							
checksum[0]							
checksum[1]							
checksum[2]							
checksum[3]							

- each block pointer stores the hash of a block being pointed to,
- effectively, this turns the whole FS into a Merkle tree,
- observe that the data and the hash are stored separately; this way there is less chance that both will be corrupted.

Data management unit

In ZFS, block pointers are arranged as a perfectly balanced tree. They also facilitate many consistency checks.

ZFS block pointer

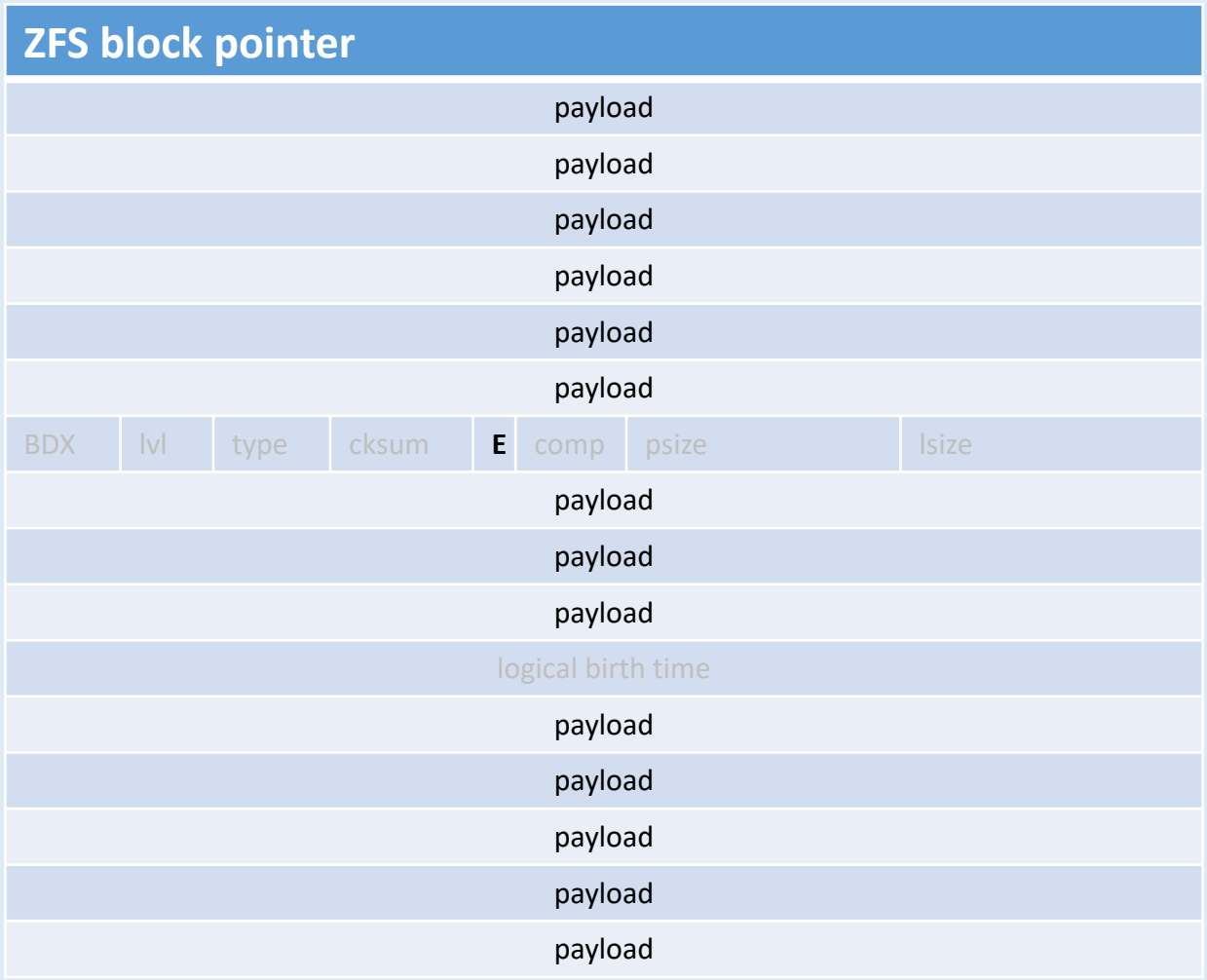
vdev0						pad	asize
G	**offset0**						
vdev1						pad	asize
G	**offset1**						
vdev2						pad	asize
G	**offset2**						
BDX	lvl	type	cksum	E	comp	psize	lsize
spare							
spare							
physical birth time							
logical birth time							
fill count							
checksum[0]							
checksum[1]							
checksum[2]							
checksum[3]							

A block pointer may reference up to 3 replicas of a block. By default,

- user data has 1 replica,
- FS metadata have 2 replicas,
- ZFS pool metadata have 3 replicas.

Data management unit

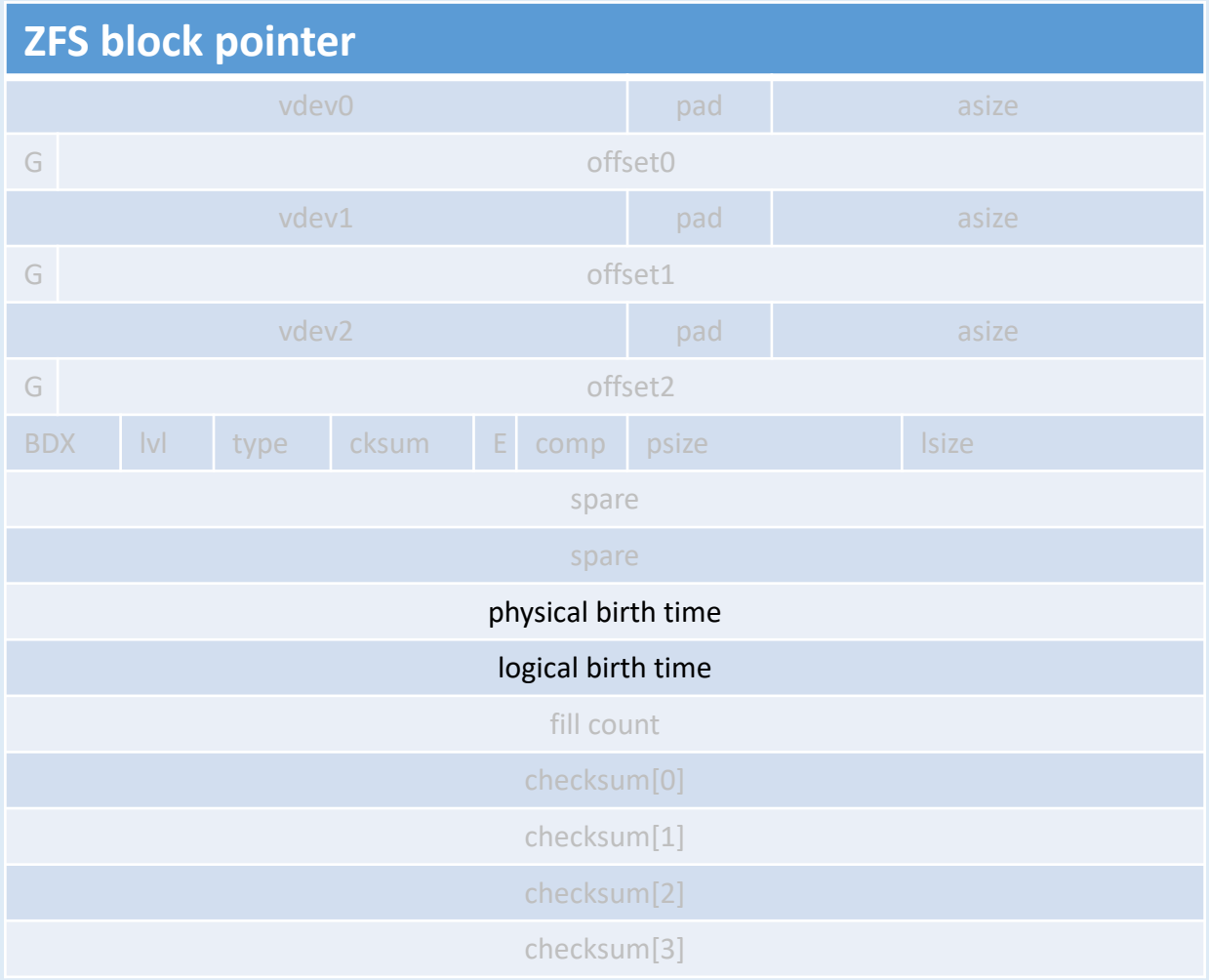
In ZFS, block pointers are arranged as a perfectly balanced tree. They also facilitate many consistency checks.



Flag E means that this block pointer is not really a pointer but contains **e**mbedded data.

Data management unit

In ZFS, block pointers are arranged as a perfectly balanced tree. They also facilitate many consistency checks.



- Physical birth time is the number of a transaction that allocated a block being pointed to.
- Logical birth time is the number of a transaction that allocated this block pointer.

These numbers make it possible to find the differences between trees efficiently, which is needed for space reclamation during removal of snapshots.

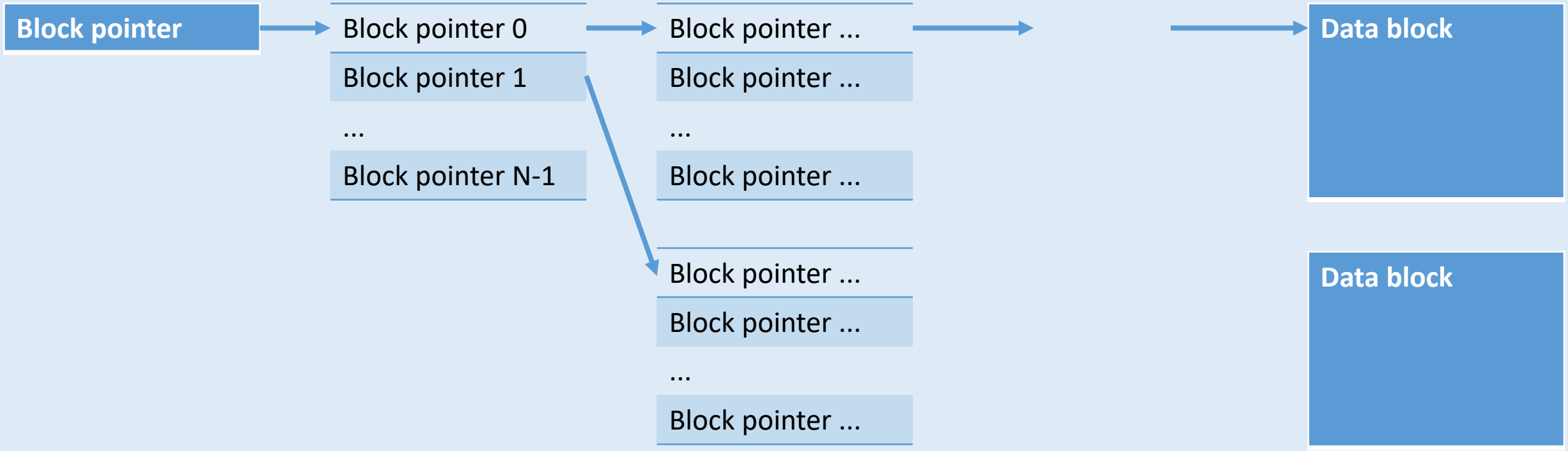
These numbers are different iff a block being pointed to was deduplicated.

See also: https://papers.freebsd.org/2019/BSDCan/ahrens-How_ZFS_Snapshots_Really_Work.files/ahrens-How_ZFS_Snapshots_Really_Work.pdf

More on transaction IDs and snapshotting: https://postgrespro.com/blog/pgsql/5967899

Data management unit and “file system objects”

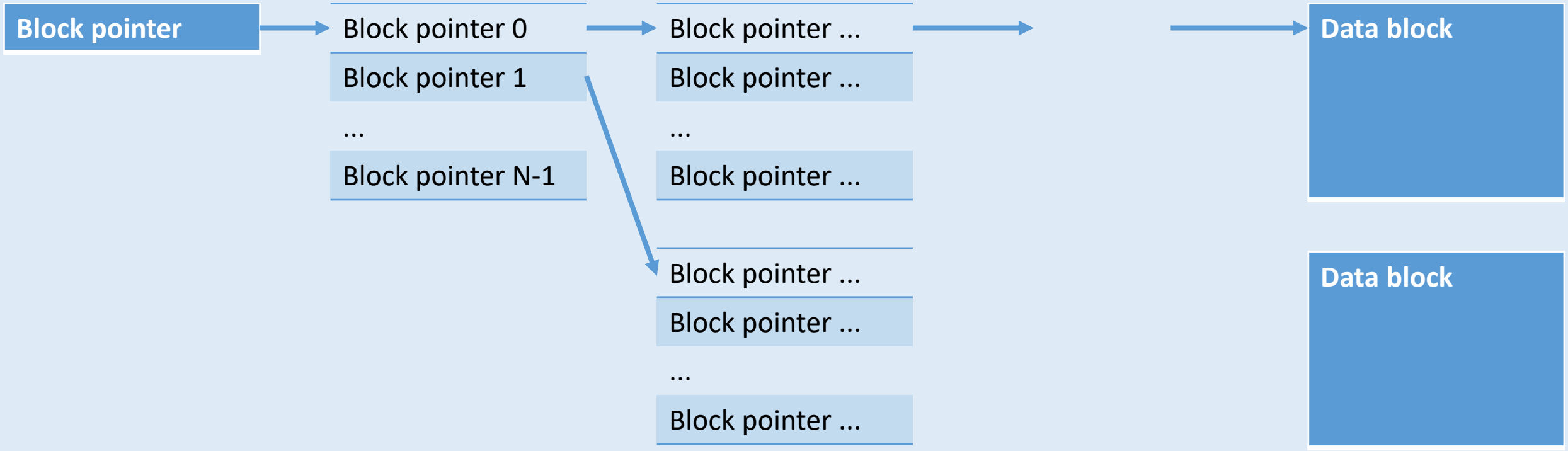
A “file system object”, a dnode for short, is a tree of block pointers:



Remark: unlike an ext2 inode that has direct block pointers, double indirect pointers and triple indirect pointers, a ZFS dnode has only one pointer to locate the file content. If the file is 128Kb or shorter, then the pointer points directly to the data, otherwise it points to a tree.

Data management unit and “file system objects”

A “file system object”, a dnode for short, is a tree of block pointers:



Remark: ZFS has no support for extents:

1. extents are too big to copy-on-write and dedup,
2. extents are too big to copy when rebuilding corrupted data (compare a 128K block to a 128M extent),
3. merging block reads into batches gives most of the performance gains of extents.

Data management unit: files, directories and file systems

A file in ZFS is an unstructured “file system object”.

A directory is a file system object that ZFS parses as a hash table that maps file names to dnodes.

Remark: hash tables in ZFS are handled by a module called ZAP, ZFS Attribute Processor. Because of this you may often find references to “ZAP objects” in the documentation.

Data management unit: files, directories and file systems

A file in ZFS is an unstructured “file system object”.

A directory is a file system object that ZFS parses as a hash table that maps file names to dnodes.

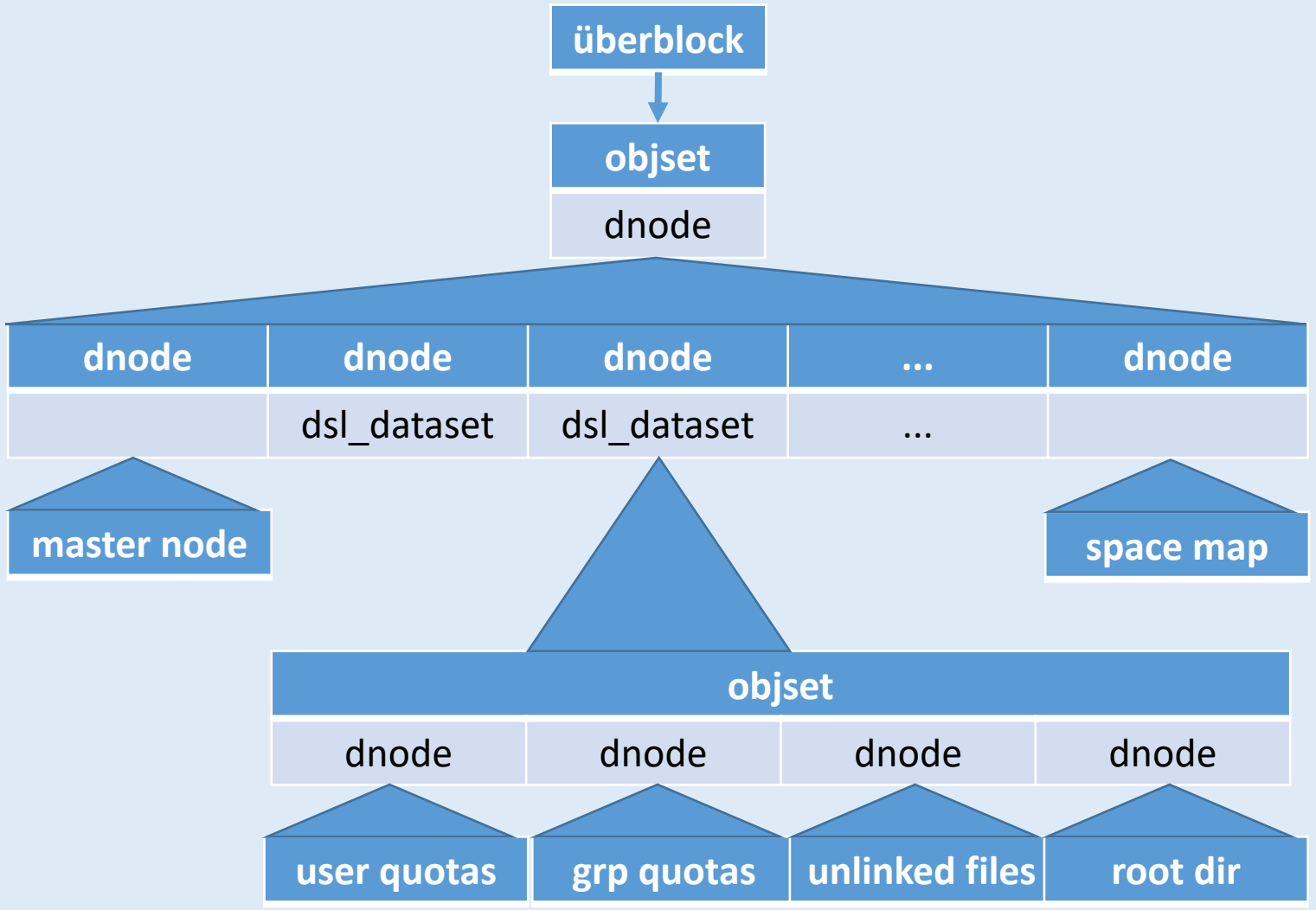
Remark: hash tables in ZFS are handled by a module called ZAP, ZFS Attribute Processor. Because of this you may often find references to “ZAP objects” in the documentation.

A file system in ZFS is also a file system object that contains 4 pointers:

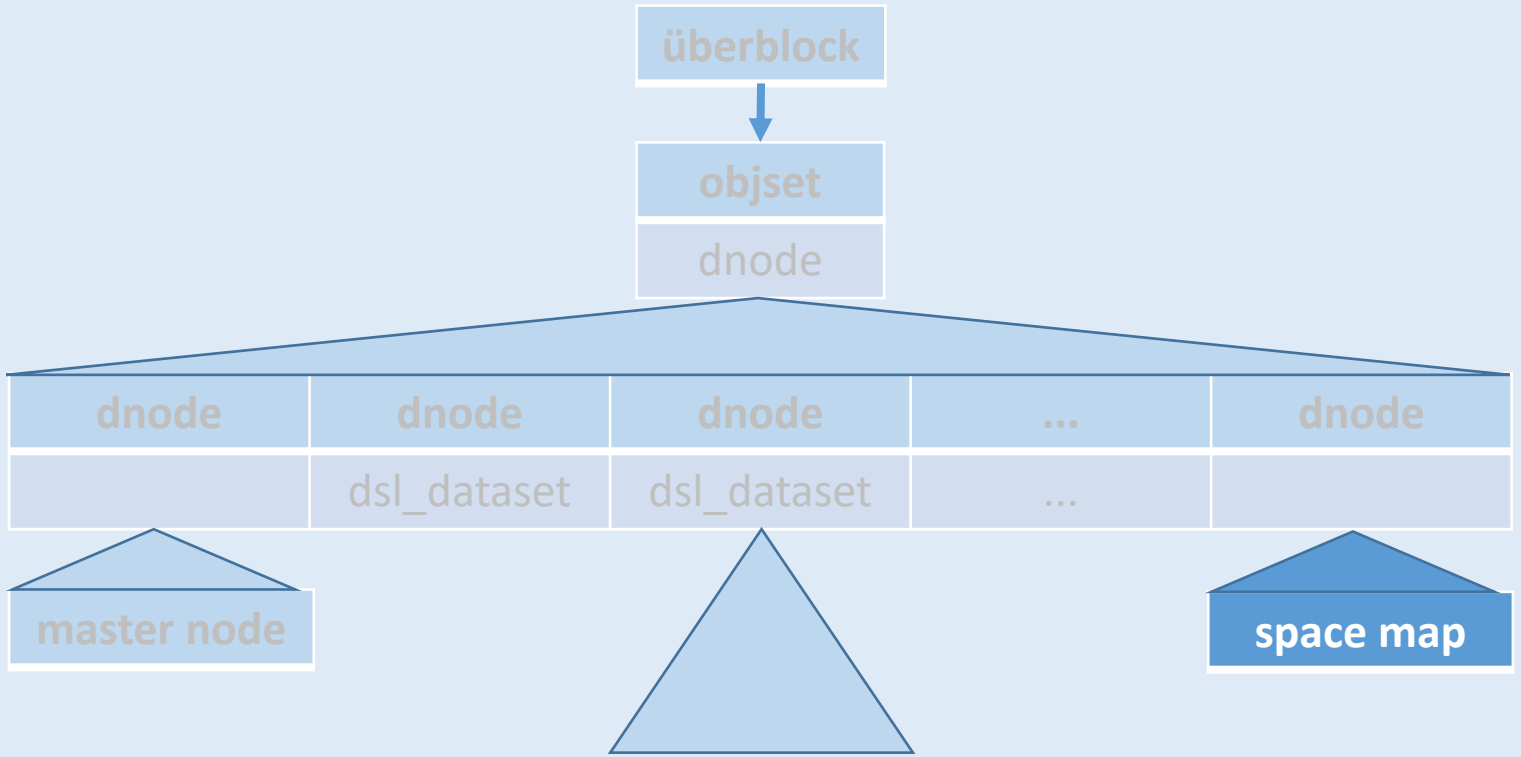
- ZAP object that represents the root directory,
- ZAP object that keeps user quotas,
- ZAP object that keeps group quotas,
- ZAP object that holds pointers to open files without names.

Observation: with ZFS it is trivial to create a new FS that uses resources of a disk pool. It is as simple as creating a file or a directory.

Overview of a ZFS pool



Tracking free space: space maps



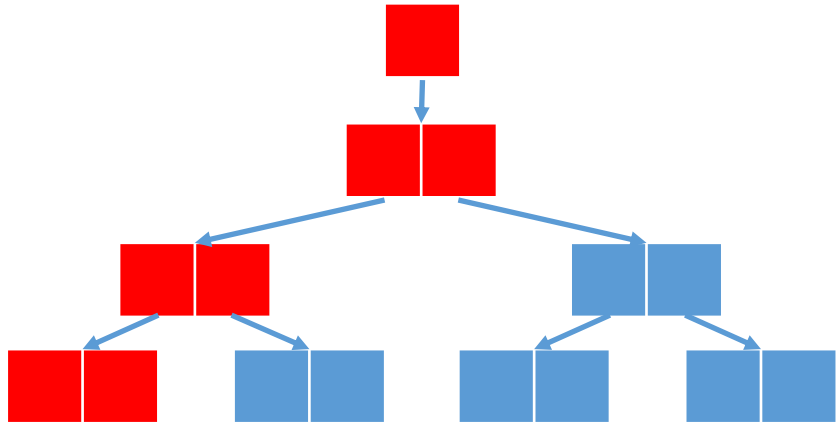
ZFS maintains per-metaslab space maps.

1. A space map is a journal of `alloc()` and `free()` requests to the metaslab.
2. The journal is regularly replaced with a condensed version of it. It contains only entries to mark allocated extents as allocated.
3. Space maps have a useful property: they grow smaller as metaslabs become full. A full metaslab needs only 1 space map entry.

Checkpointing

Recall that modifying even a single block forces the FS to modify all parent blocks.

For this to be performant, both ZFS and WAFL try to accumulate as many modifications as possible before forcing a checkpoint. It is typical to accumulate several gigabytes of dirty data.



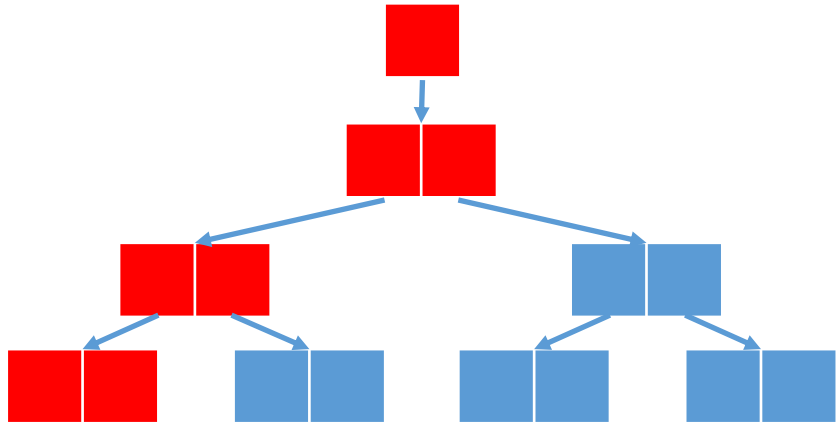
Checkpointing

Recall that modifying even a single block forces the FS to modify all parent blocks.

For this to be performant, both ZFS and WAFL try to accumulate as many modifications as possible before forcing a checkpoint. It is typical to accumulate several gigabytes of dirty data.

There is trouble, though: how does one handle `open(O_SYNC)` and frequent `fsync()`s?

- WAFL relies on the HW of NetApp's appliances and confirms write once they are stored in the NVRAM.
- ZFS needs an SSD journal in the pool. It confirms writes once they are persisted in the journal, and delays checkpoints until there is enough dirty data to write. This log is called ZFS Intent Log, or ZIL for short.



How does copy-on-write implement various requirements that we've added

A FS must always be consistent.	The superblock always points to a consistent tree.
FSCK must be fast. Better yet, there should be no FSCK.	FSCK is not needed.
Writes to files must be fast. Metadata-heavy workloads must be fast, too.	Copied blocks may be allocated sequentially. Updates to different files may also be written sequentially.
It must be easy to use multiple disks for redundancy and speed, and it must be possible to add and replace disks online.	Storage Pool Allocator + Data Management Unit which makes a FS another file-like object.
A FS must implement fast snapshots and clones, and must be able to roll back to a previous snapshot.	Creating a snapshot is free. Just do not remove the previous version of the superblock and the tree that it points to. Rolling back to a snapshot is trivial, too.
A FS must protect itself from the bit rot.	1. Data is stored in mirrored or raidz vdevs, 2. The whole FS is a Merkle tree, 3. The most critical metadata is replicated.
When doing RAID, it must avoid RAID write holes.	RAID write hole may be created when blocks are overwritten. A copy-on-write FS never overwrites blocks, hence it cannot create write holes.

Downsides of COW

- FS operations cause copying of trees. They have a good amortised cost, but `fsync()`s that can't wait for enough dirty data are very expensive.
- Implementation complexity:
  - We still need journaling to have decent performance for synchronous writes.
  - There are no extents so one relies on very good IO scheduling to have decent read performance.
  - Checkpointing takes a long time, so it is important for it not to block concurrent modifications to the FS*. A file system like ext4 that does frequent checkpoints can lock buffers or copy them while flushing them. ZFS can't do that because that would block writers for seconds.
  - Etc.
- Every modification to a file system needs free space:
  - garbage collection,
  - deleting a file.

** See also: stable pages, https://lwn.net/Articles/442355/*