Famfs: Get ready for big pools of disaggregated shared memory

John Groves
Technical Director
Co-Chair of the CXL Software and Systems Working Group
Primary developer of famfs
Aug 2025

micron Intelligence Accelerated™

SNIA DEVELOPER CONFERENCE
SDC 25
BY Developers FOR Developers

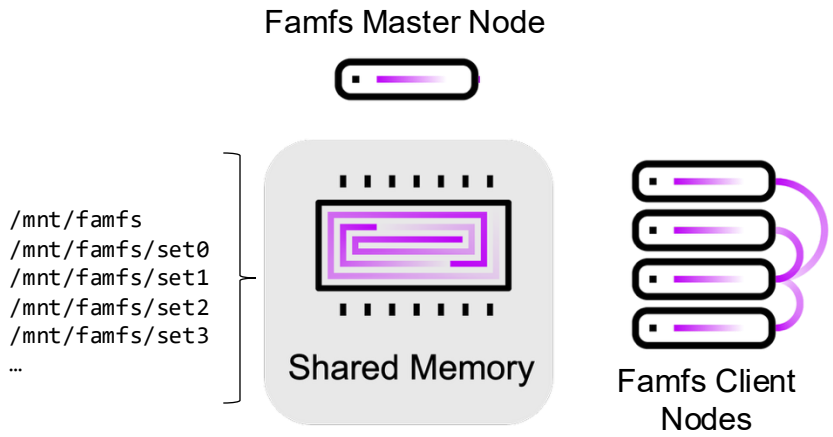
Agenda: Big pools of disaggregated memory are coming

- Overview: Not all problems fit in memory
  - Scaling techniques
  - Disaggregated memory use cases

- Shared memory challenges:
  - Before disaggregation
  - With disaggregation

- What will disaggregated memory look like?

- Famfs: Shared memory needs an access method
  - Famfs history and architecture
  - Files are memory

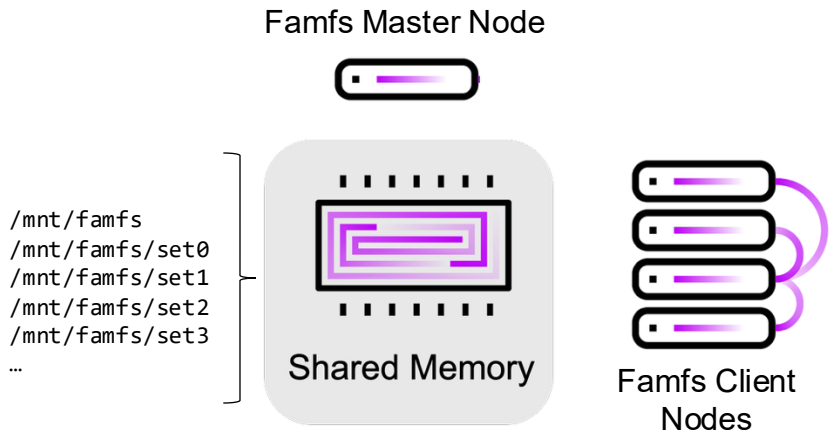



```
mkfs.famfs /dev/dax0.0
famfs mount /dev/dax0.0 /mnt/famfs
famfs cp [-r] <src> <dest>
famfs creat -s <size> <dest>
```

Famfs organizes disaggregated memory as a scale-out file system

Enabling shared JBOM for all apps that can use files

- Famfs is fully open source!
- Memory is accessible as files
  - Write/read become memcpy
  - Mmap provides byte / cache-line access
- “All” apps can access data in files
- Famfs files are memory and not storage
  - Move data into famfs for in-memory access
  - Move data out of famfs to store persistently
- Posix permissions apply, along with strict partitioning of data from separate files
- Orchestration layers such as PNFS can use famfs as a tier – providing memory performance + scale-out sharing




```
mkfs.famfs /dev/dax0.0
famfs mount /dev/dax0.0 /mnt/famfs
famfs cp [-r] <src> <dest>
famfs creat -s <size> <dest>
```

Not All Problems Fit in Memory

Observations from a long career in storage and memory

Not All Problems Fit in Memory!!

Draft

(We sometimes forget this because we work on big problems these days)

Observations from a long career in storage and memory

- Not all problems fit in memory

- The problems (data sets) get bigger, but the available techniques have remained the same
  - **Demand paging (nvme/dma/rdma thrashing)**

Demand paging

- Missing data is read from storage or network (DMA, RDMA, etc.)
- Least recently used (LRU) data is dropped from memory to make room for demand-paged data
- Performance depends on parallelism and access patterns
  - OK for sequential access or highly parallel
  - Terrible for random access

Observations from a long career in storage and memory

- Not all problems fit in memory

- The problems (data sets) get bigger, but the available techniques have remained the same
  - Demand paging (nvme/dma/rdma thrashing)
  - **Scale out (more servers / mem / GPUs)**

Scale Out

- Modern big-data computing feels like “all scale-out all the time”
- Data is sharded for clustered compute
- The tools are good but the workflows are complex
- Some data does not shard well, leading back to thrashing

Observations from a long career in storage and memory

- Not all problems fit in memory

- The problems (data sets) get bigger, but the available techniques have remained the same
  - Demand paging (nvme/dma/rdma thrashing)
  - Scale out (more servers / mem / GPUs)
  - **Scale up (bigger servers / mem / GPUs)**

Scale Up

- Build servers with more
  - Cores
  - I/O
  - Memory
- But single-server memory scaling is gradual
- Current max out at 3TB of memory

Observations from a long career in storage and memory

- Not all problems fit in memory

- The problems (data sets) get bigger, but the available techniques have remained the same
  - Demand paging (nvme/dma/rdma thrashing)
  - Scale out (more servers / mem / GPUs)
  - **Scale up and out (bigger servers / mem / GPUs)**

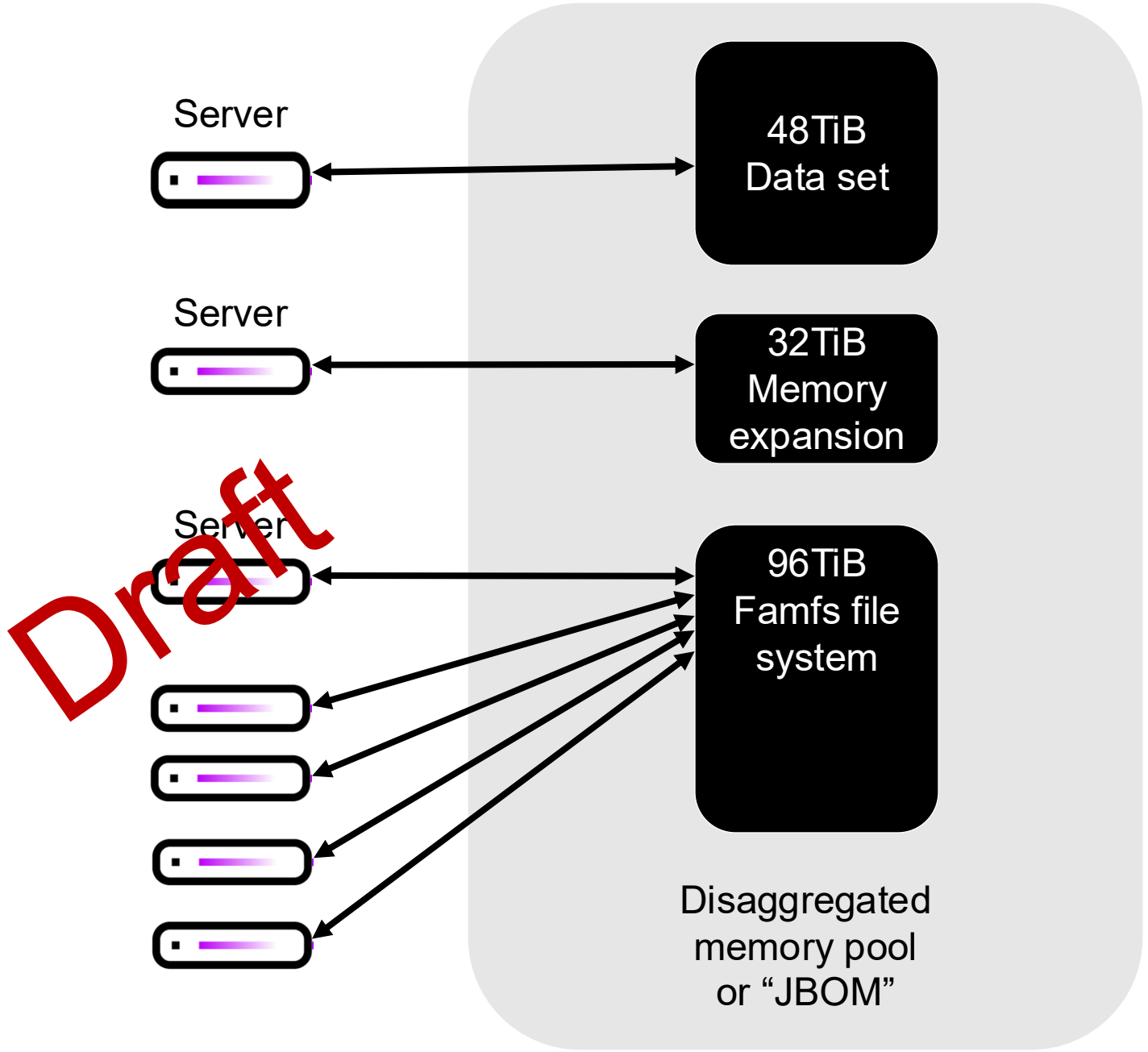
Scale Up And Out!

- Build servers with more
  - Cores
  - I/O
  - Memory
- But single-server memory scaling is gradual
- Current max out at 3TB of memory

➢ **Build and share huge pools of composable, shared memory!**

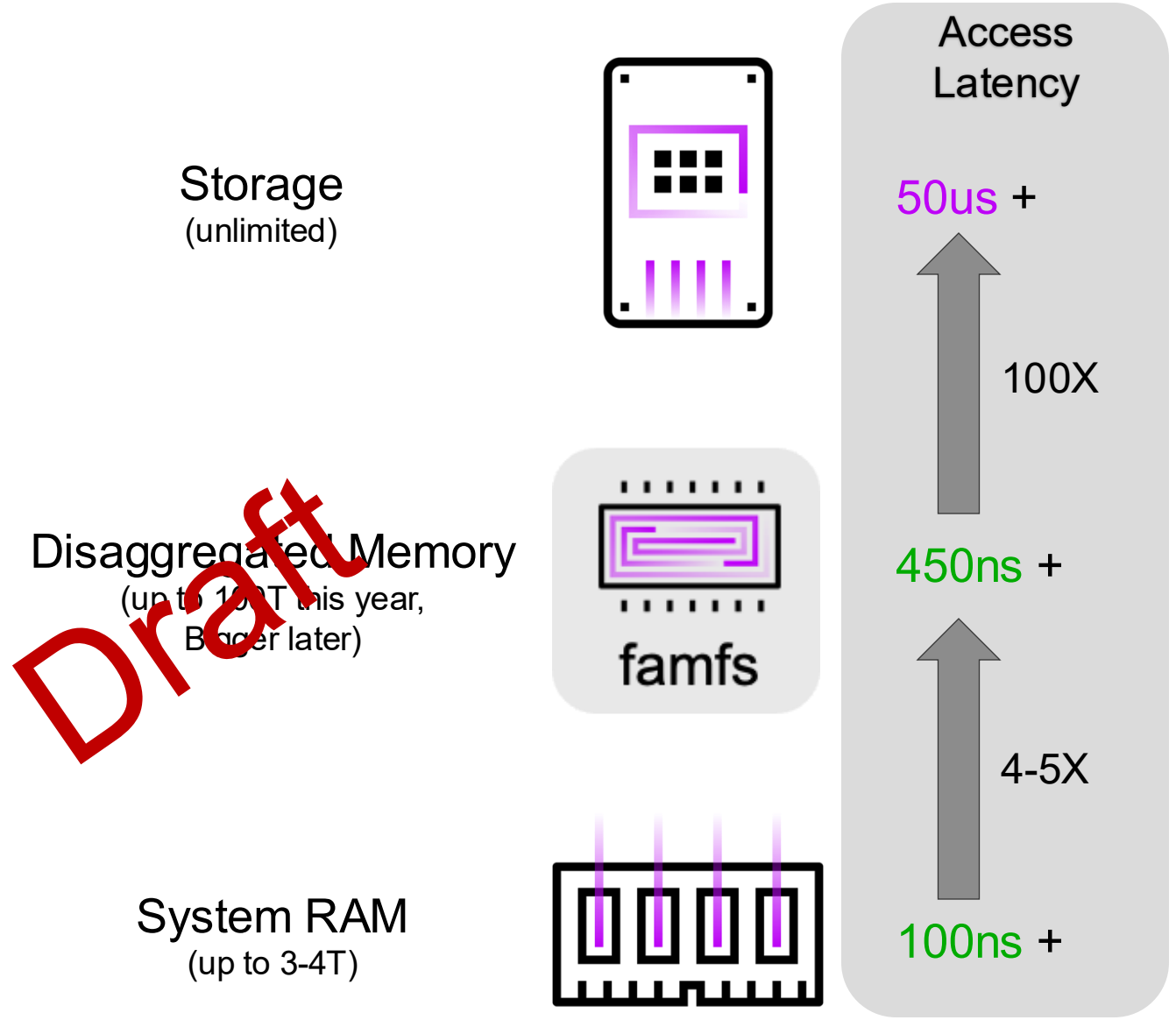
Composable Memory Let's Us Scale Up and Out

- Not all problems fit in memory
- The problems (data sets) get bigger, but the available techniques remain the same
  - Scale up (bigger servers / mem / GPUs)
  - Scale out (more servers / mem / GPUs)



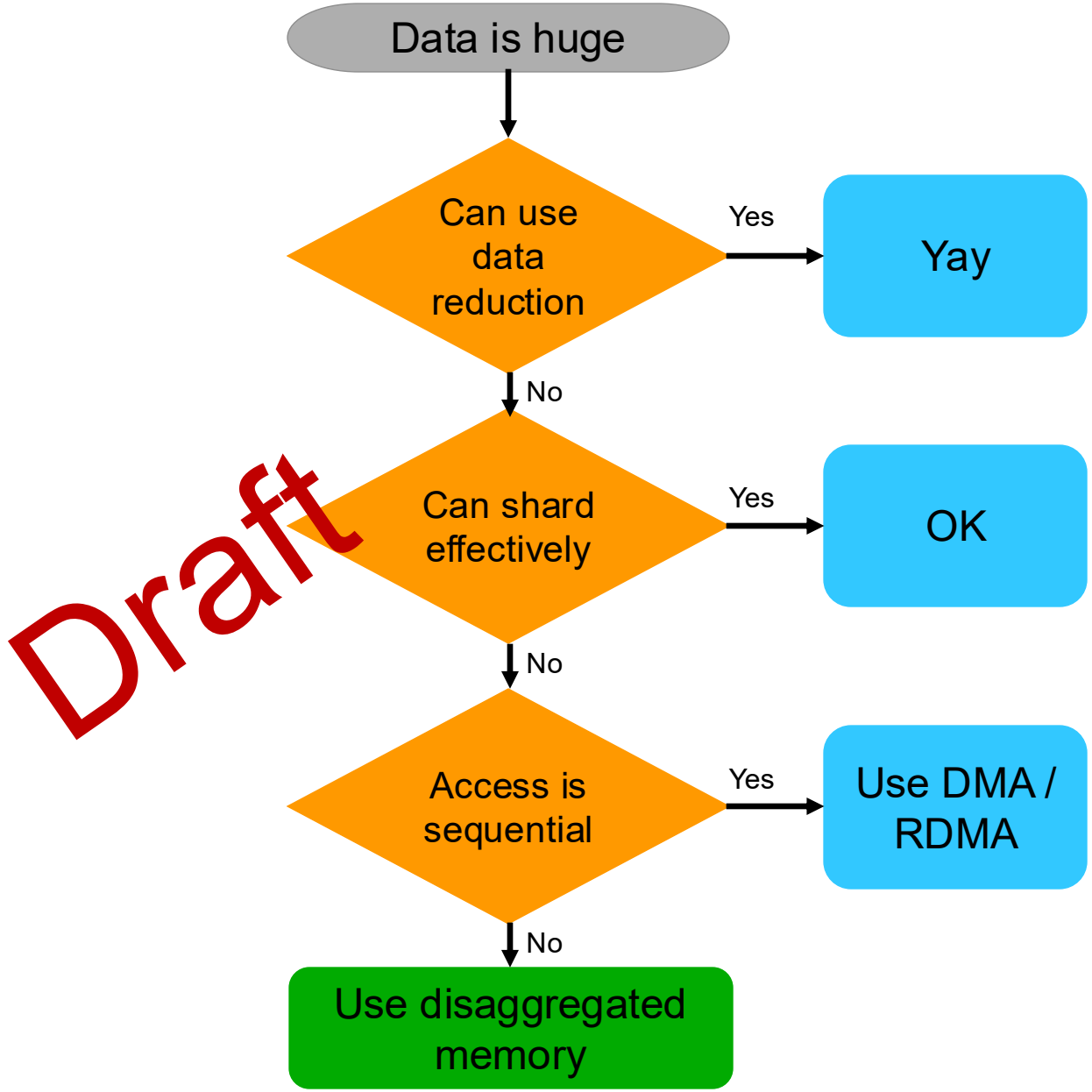
The superpower of memory is low-latency random access

- Memory access latency is much lower than storage latency
- Compare disaggregated memory to storage, not system-ram
- Data that doesn't fit in System-RAM can be random-accessed in disaggregated memory 100x faster than storage



What if data is [much] bigger than memory?

- Some data can be reduced in size effectively
- Some data can be sharded (split across hosts) effectively
- Some data is accessed sequentially, and can be staged via DMA / RDMA
- Random access in disaggregated memory is 2 orders of magnitude lower latency than NVME (100x Improvement)

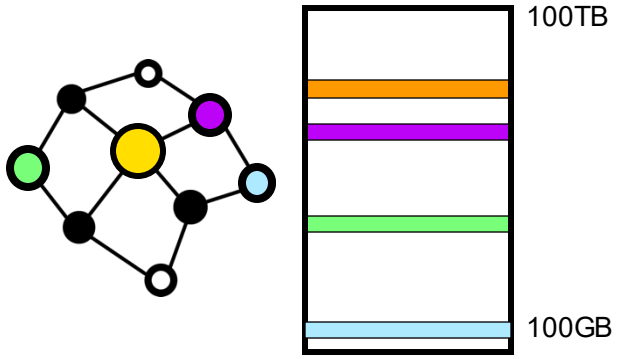


The superpower of memory is low-latency random access

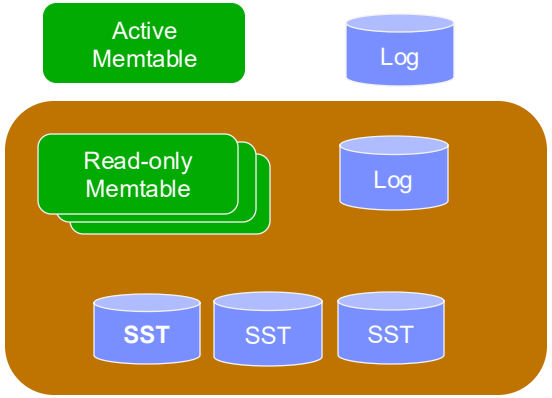
- Famfs with big memory breaks scaling barriers for
  - Graph analytics
  - Rag pipelines
  - In-memory databases and indexes

- Graph databases, RAG/LLM pipelines and indexes can scale to 100T and beyond without sharding or demand-paging

Graph analytics and neural networks

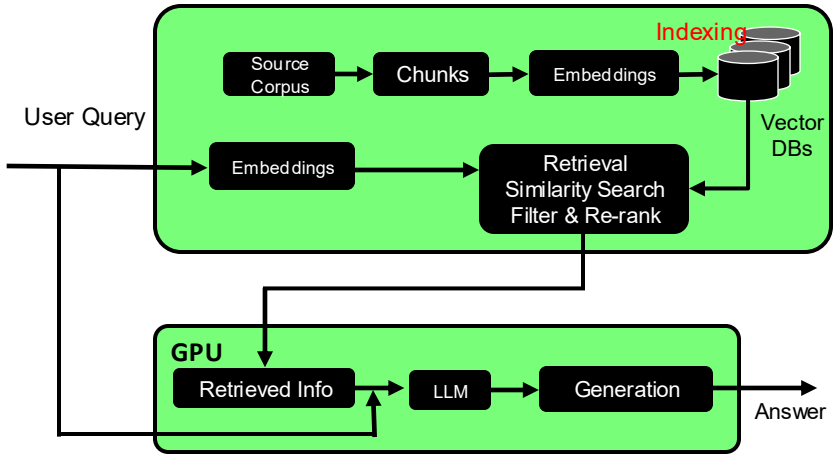


Big Data Index



RocksDB Architecture with 100GB-1PB Datasets using key-value store

RAG/LLM Pipeline



Famfs use cases

Great leverage for “data frames” space (analytics + AI)
and in-memory database applications

- All apps and tools can access data frames in files
- Data analytics and AI applications share a lot of infrastructure in the “data frames” and data lake space
- Data frames ecosystem already uses shared data sets
  (Already accesses data frames as memory-mapped files)
- Many of these use cases are read-only during the data-sharing portion of the work-flow
- Putting shared data frames in famfs enables CXL memory without requiring app modifications
  - (workflows may need to be modified, but this class of apps can easily do that)
- Data lake / data file / in-memory database formats

RocksDB

DuckDB

Shared Memory Challenges

Before Memory Disaggregation

Shared Memory Was Already Hard

Even before disaggregation

- Need synchronization primitives (locks / atomics / barriers)
- Must control write ordering to avoid “nonsense contents
- Correct synchronization comes with significant performance penalties
  - The finer grained, the more the penalty
  - Lock contention, atomic contention, cache line contention, etc.
- Bugs in synchronization are likely
  - Deadlocks, livelocks, stale data, etc.
- Of course when system software developers think of shared memory, we can’t help but think of fine-grained multi-writer cases…

TODO: diagram

Shared memory in a single host

Trend: Less Synchronization

A lock avoided is never contended

- Redis (in-memory KV-store) has a single-threaded put/get path
  - No locks needed
  - Scale out with a Redis instance per core, and consistent hashing to direct keys to instances

- Most sharded big data jobs don't mutate data in place
  - Think of it as "Publish and Share"
  - Scaled-out computing can use non-mutating data published in shared memory with no SW mods

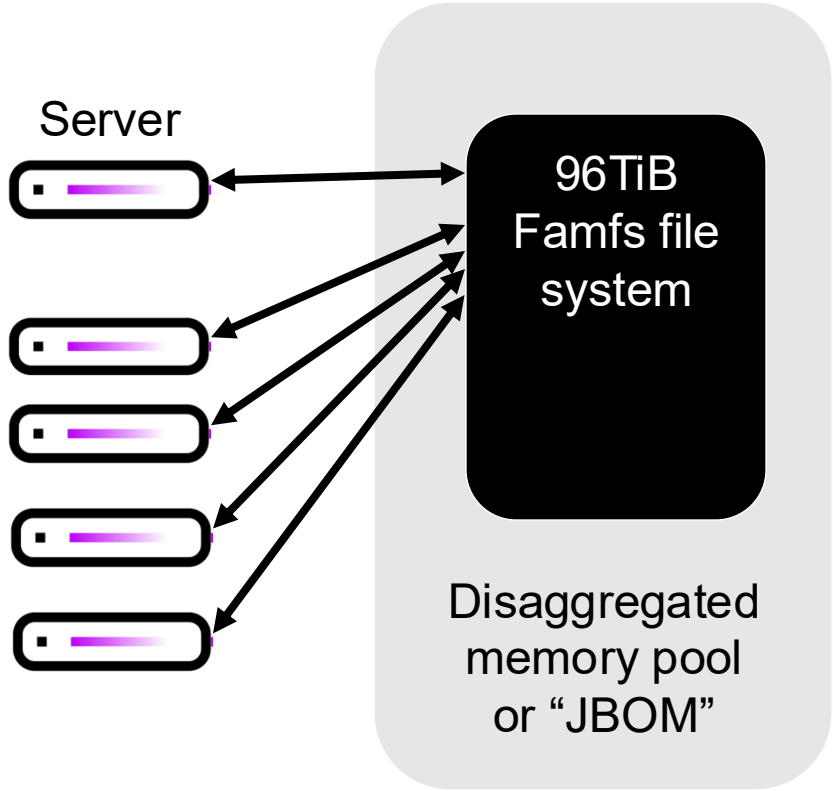
TODO: diagram.

Shared Memory Challenges

With Memory Disaggregation

CXL Supports Sharable Memory

- In CXL 2, it can be done via static configurations
  - SW managed cache coherency
- CXL 3 & beyond support shared memory with proper access control via Dynamic Capacity Devices (DCDs)
  - Optional HW cache coherency (but more limited than in-host)
  - HW coherent memory will probably be limited to small fractions of total memory
- Most apps that share CXL memory will probably be in the “lock avoidance” category
  - Debate me if you like



CXL Coherency Tools

- CXL CPUs must support barriers/fences, even in in non-coherent memory
  - Write barriers (aka sfences) must wait for any pending writebacks from “clflushes”
  - Read barriers (aka lfences) must wait for any pending cache line invalidation from “clflushes”
  - Comment: having separate instructions for write-back and invalidate would be quite helpful; we frequently know which we need
- CXL does not support atomics
  - No in-memory locks
  - But lockless structures such as logs and ring buffers can work
- With SW coherency, SW must detect or avoid stale cache lines

TODO:
Diagram: ring buffer example

Lock Avoidance is Great for CXL

- Synchronization to disaggregated memory will be more costly than with local memory
  - And synchronization is already costly…
- ”Publish and Share” patterns work great
  - And a lot of workflows can adapt to this pattern

Draft

RocksDB

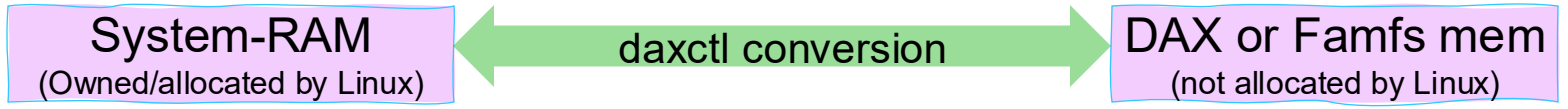
DuckDB

What Will CXL Disaggregated Memory Look Like?

DCD Intro

Background: CXL memory usage models

Pooling (composable system-ram)



- Memory is added as System RAM (managed by Linux)
- Tiering and migration are viable (`migrate_pages()`, TPP, DAMON, etc.)
- Incompatible with multi-host sharing (memory gets zeroed when Linux “onlines” it)
- It’s possible to provision very large amounts of memory for jobs that can’t run in 3-4T

Shared memory as system-ram is nonsense

Sharing (composable ‘not system ram’)

- The hardware supports this (CXL3, DCD, etc.)
- These cases include
  - Both concurrent and sequential sharing
  - Other use cases that use Linux memory-mgmt
- Software usage DAX is too complicated

- Famfs is the missing link
  - “All” apps can use data in files
  - Files already map to memory
  - Many apps use big data in files
  - RAS “blast radius” is limited to apps that access the memory

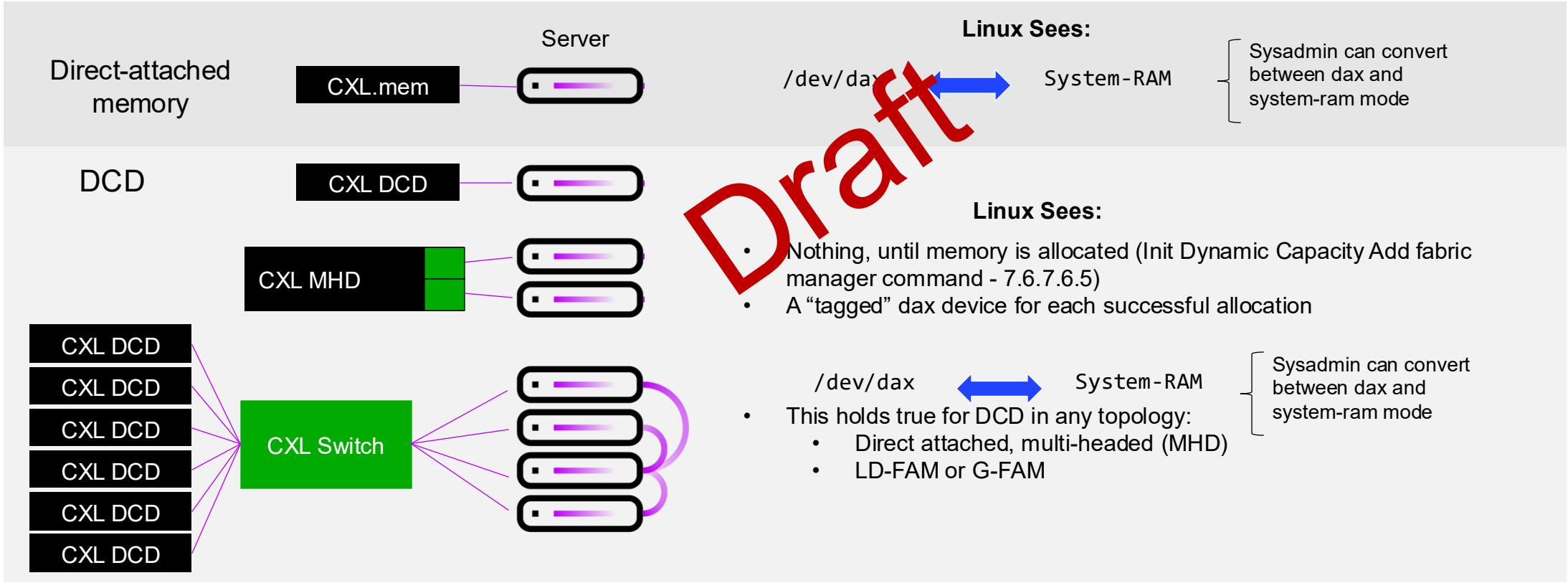
What is DAX?

- Dax is a character driver for a memory device or object
- Memory mapping a dax device brings that memory into the process virtual address space
  - provides load/store access to the memory
- A direct-attached CXL device will surface natively as a dax device
- Dax devices can be “onlined” as system-ram
  - A new numa node will appear
  - dax device will disappear

TODO:
Dax diagram

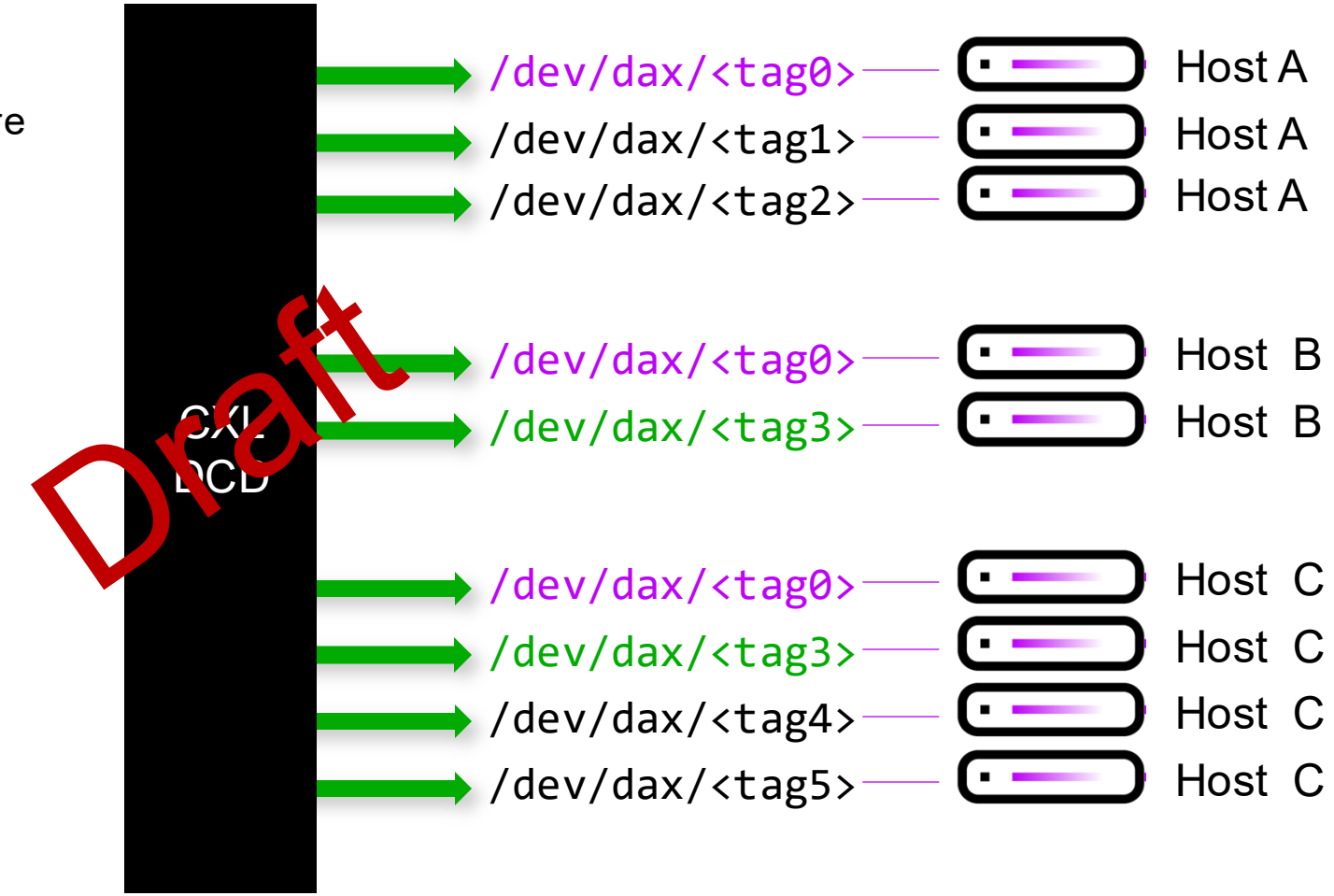
CXL Dynamic Capacity Devices (DCDs)

- Think of a **Dynamic Capacity Device** (**DCD**) as a memory device with built-in allocator and access control
- The allocator is necessary for multi-host environments
- DCD (via fabric manager) can give additional hosts access to a sharable allocation, writable or read-only, etc.



CXL tagged capacity name space

- DCD is not usable until memory is allocated
- Allocation (Init DC Add)(sharable allocations are ”tagged”, and appear as “virtual” dax devices)
- Tagged dax memory can be “onlined” as system-ram (non-shared memory)
- Sharable memory can surface simultaneously or not
- A famfs instance lives on one or more tagged dax memory instances
- Famfs can also interleave files across an arbitrary number of Tags
- CXL interleave can be programmed across multiple tagged allocations*



A CXL JBOM Will Have Many DCDs

- Each “Tagged Capacity” allocation will surface as a dax device

- Interleaving is critical to memory performance, so most use cases will need aggregate across many tagged capacity instances
  - Note hardware interleaving requires the same DPA range on every DCD – which will be difficult or impossible if DCD address spaces fragment
  - Famfs can interleave across many DCDs independent of DPA-space fragmentation

- Each famfs file system should span across many Tagged Capacity instances (each from a separate DCD)
  - Interleaved for performance

TODO:
- Many DCDs
- ->many tagged capacity allocations
- ->interleaved famfs file system

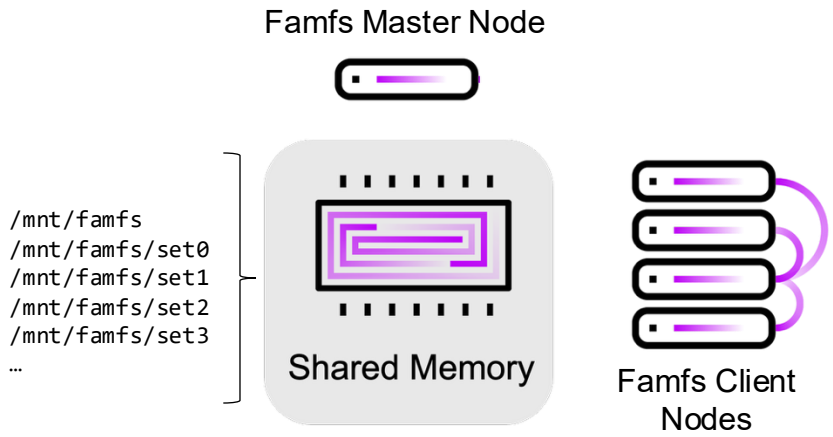
Introduction to famfs

The file system abstraction is a natural fit for shared memory

Famfs organizes disaggregated memory as a scale-out file system

Enabling shared JBOM for all apps that can use files

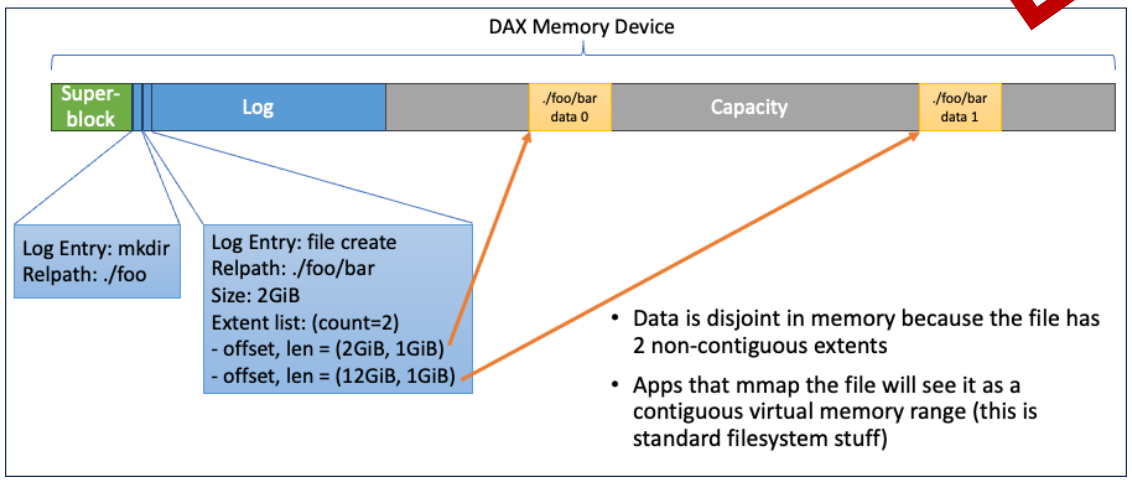
- Memory is accessible as files
  - Write/read become memcpy
  - Mmap provides byte / cache-line access
- “All” apps can access data in files
- Famfs files are memory and not storage
  - Move data into famfs for in-memory access
  - Move data out of famfs to store persistently
- Posix permissions apply, along with strict partitioning of data from separate files
- Orchestration layers such as PNFS can use famfs as a tier – providing memory performance + scale-out sharing

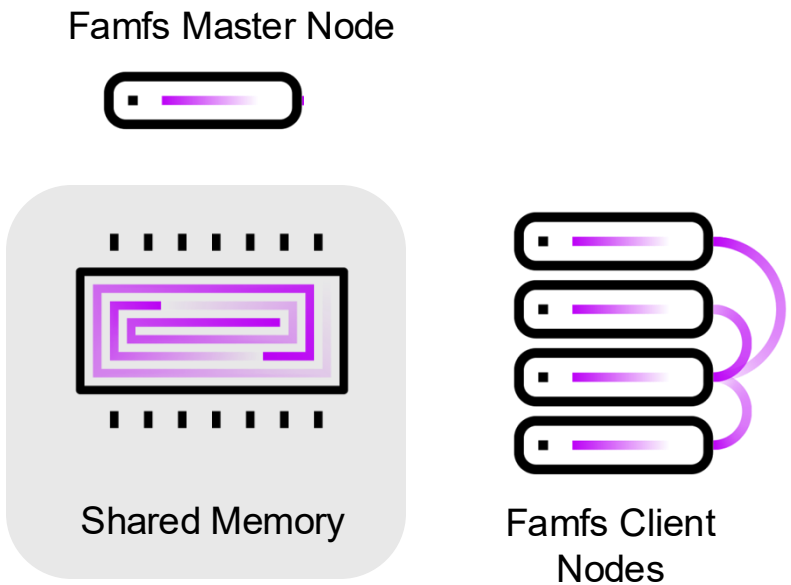



```
mkfs.famfs /dev/dax0.0
famfs mount /dev/dax0.0 /mnt/famfs
famfs cp [-r] <src> <dest>
famfs creat -s <size> <dest>
```

Famfs is a Scale-Out Shared-Memory File System

- To mount it, you just need access to the memory
  - All metadata is stored in an append-only log
  - Log is written by Master and ”played” by Clients
- V1 handles clients with stale metadata by not supporting truncate or delete
- Metadata handled in user space (library, cli, currently no daemons)
- Read / write / mmap / vma faults handled in kernel
- Memory mapping from famfs == cache-line level access to shared mem
- Many of the limitations can be addressed in future versions






```
mkfs.famfs /dev/dax0.0
famfs mount /dev/dax0.0 /mnt/famfs
famfs cp [-r] <src> <dest>
famfs creat –s <size> <dest>
```

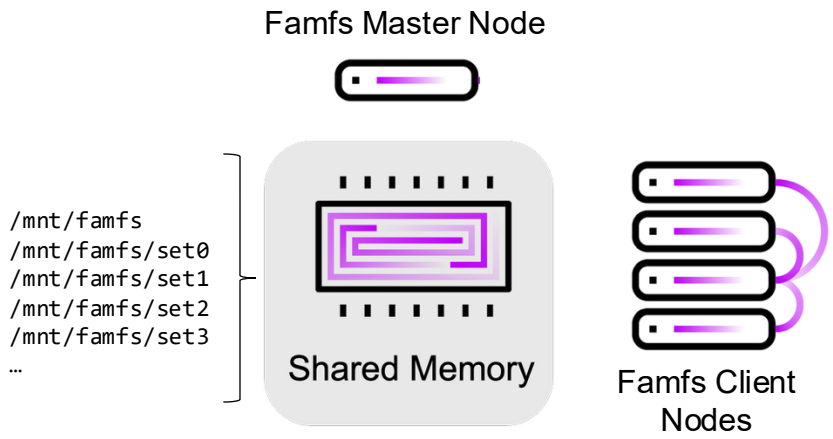
Interleaving is critical for memory performance

- CXL supports hardware interleaving but…
  - The device physical address (DPA) range must be identical on all memory devices in an interleaved set
  - But “memory devices” are virtual – based on DCD (dynamic capacity device) allocations
  - The normal fragmentation of alloc / release will make it difficult or impossible to allocate the same DPA range on, say, 16 allocations from different CXL memories

- Each famfs file can be interleaved across many CXL memory devices
  - Famfs has no constraints about DPA ranges

Famfs status: on track for Linux upstream in late 2025 / early 2026

- Nov 2023 – Introduced famfs at the Linux Plumbers Conference
- Spring 2025 – Famfs Linux patch sets released (v1, v2)
- May 2024 – Famfs session at LSFMM
  (Linux Storage, File System and Memory Management summit)
  - Conclusion: Famfs merging into fuse
- Aug 2024 – Famfs adds interleaved file support
- Nov 2024 – Famfs covered in Storage Newsletter piece on SC24
- 2024 – Famfs in pilot use at CERN, Alibaba, Intel, Universities, etc
- Sep 2024 – Famfs session at Linux Plumbers Conference
- Feb 2025 – Famfs poster at Usenix FAST Conference
- Mar 2025 – Famfs session at LSFMM (LWN Article)
- Spring 2025 – Famfs fuse-based patch sets released (v1, v2)
- Famfs documentation:
  https://github.com/cxl-micron-reskit/famfs/blob/master/README.md



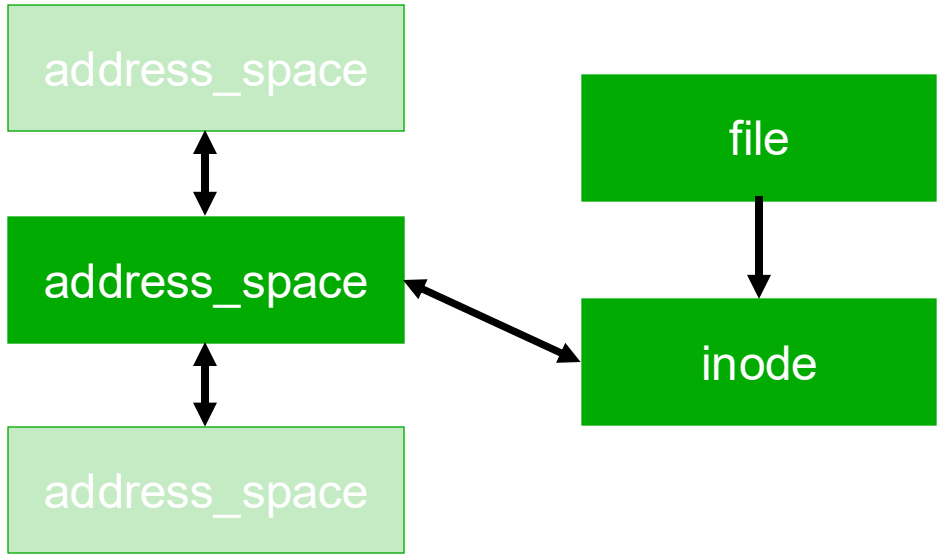

```
mkfs.famfs /dev/dax0.0
famfs mount /dev/dax0.0 /mnt/famfs
famfs cp [-r] <src> <dest>
famfs creat –s <size> <dest>
```

File system layer

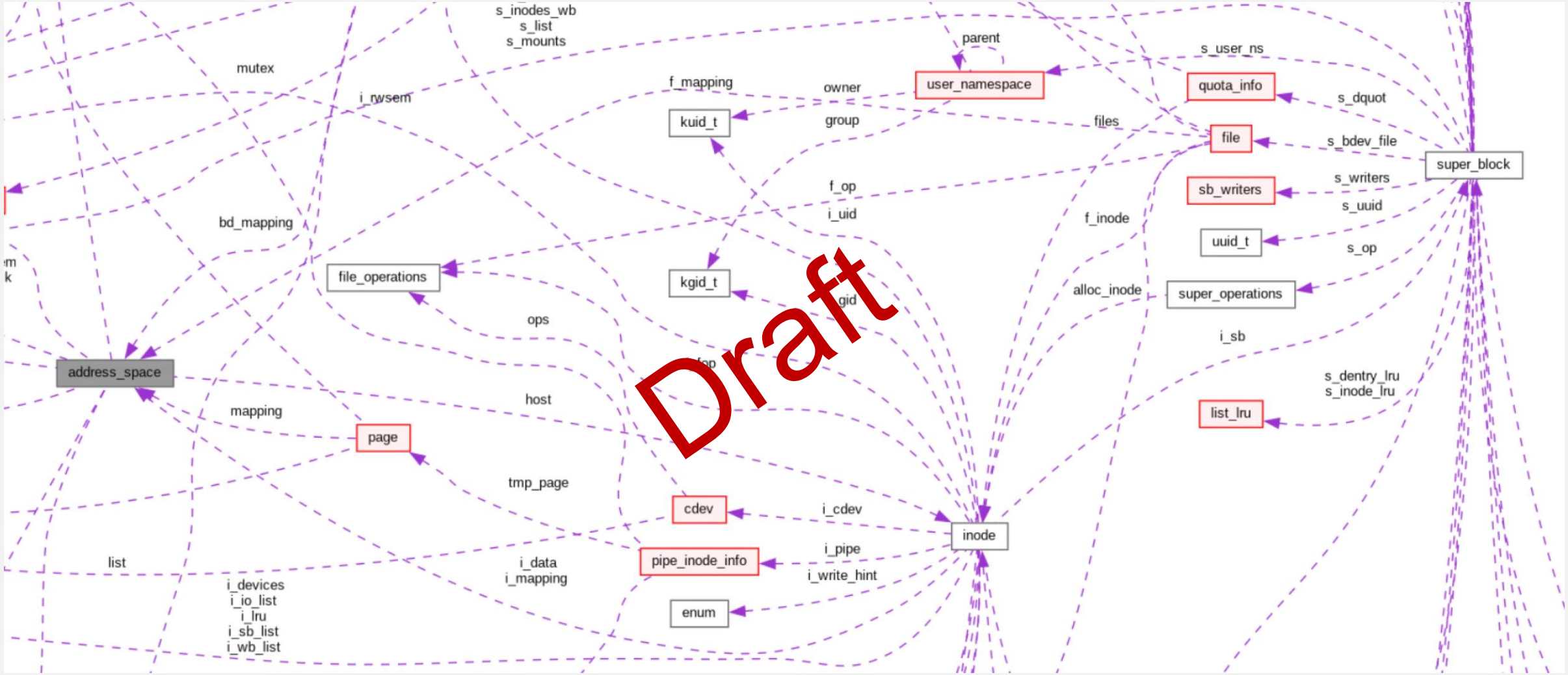
Technical details

Structure of Linux Process Virtual Address Space

- Each process has a chain of address_space structs
  - You can display them with the pmap command
  - Each maps a range of process virtual address space to actual memory
- An address space with no Inode is *anonymous* memory
  - Can't be found by other processes
  - Therefore not sharable
- The dax infrastructure introduced enabled mapping address_spaces to dax memory

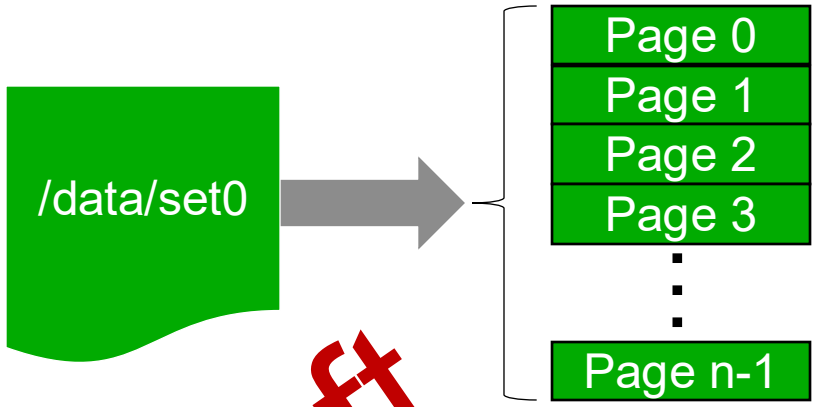


The Full Picture is Much More Complicated…



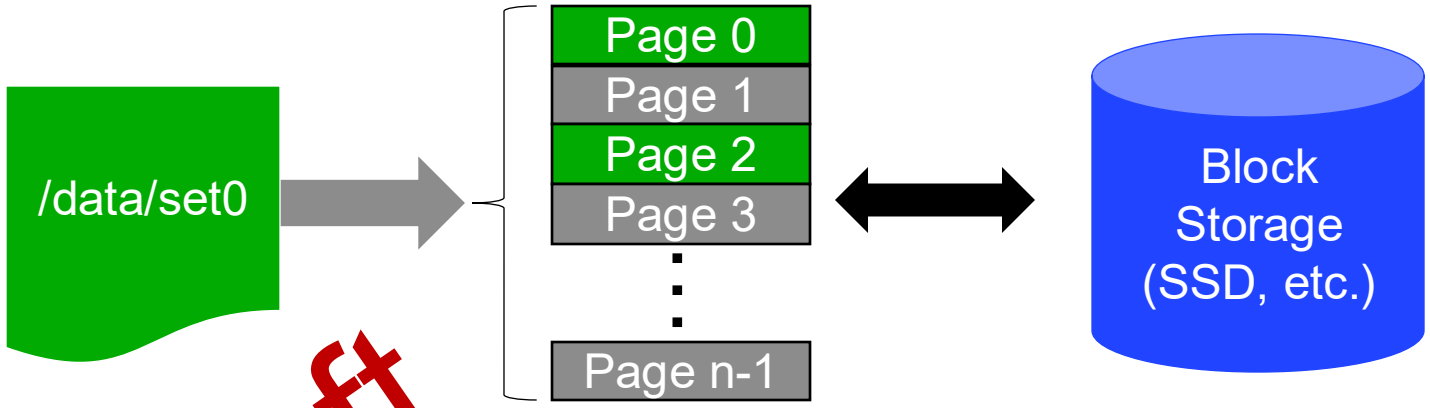
Conventional files as memory

- Files [already] map to memory
  …if the data is in memory
- When the data is in memory:
  - Read/Write are just memcpy() variants
  - Memory mapping assembles the pages into a virtual address range that is directly accessed as memory
- Many are aware of TLBs and page tables, which resolve a virtual address to memory
  - A TLB + page-table miss results in a `fault()` call to the file system to resolve the file offset to a page



Conventional files as memory

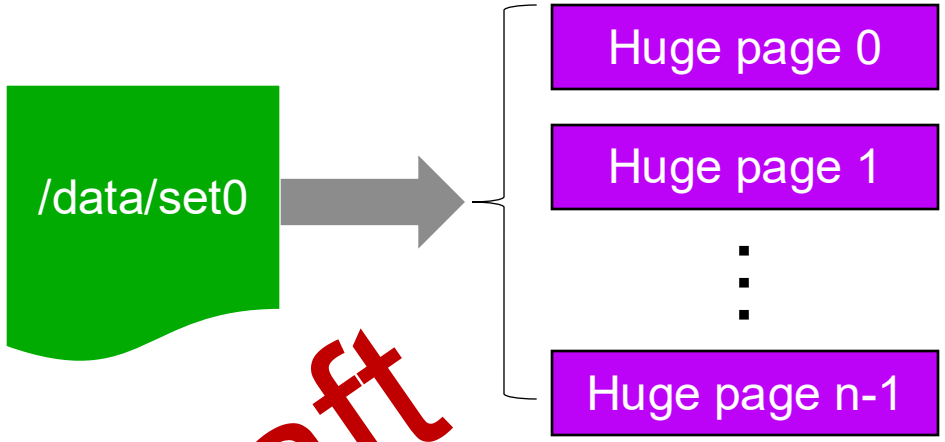
- Files [already] map to memory …if the data is in memory
- When the data is in memory:
  - Read/Write are just memcpy() variants
  - Memory mapping assembles the pages into a virtual address range that is directly accessed as memory
- Many are aware of TLBs and page tables, which resolve a virtual address to memory
  - A TLB + page-table miss results in a `fault()` call to the file system to resolve the file offset to a page



- Conventional file systems sparsely cache pages from a files backing store
  - Meaning a `fault()` call might have to allocate memory and retrieve data from backing storage
- Pages that are cached (green) are accessed as memory
- Pages that are not in cache (gray) must be faulted in from backing store if accessed

Famfs files as memory

- Files [already] map to memory
  …if the data is in memory
- When the data is in memory:
  - Read/Write are just memcpy() variants
  - Memory mapping assembles the pages into a virtual address range that is directly accessed as memory
- Many are aware of TLBs and page tables, which resolve a virtual address to memory
  - A TLB + page-table miss results in a `fault()` call to the file system to resolve the file offset to a page



- Famfs is not sparse; files are always fully mapped to memory
- Famfs data lives in [sharable] dax memory devices
- Huge page mapping reduces virtual memory mapping overhead by 512x
- Since the backing memory is not sparse, there is no downside to huge page mapping

File system / VFS functionality

Storage	Memory caching	Local memory allocation	Memory sharing (Single host)	Direct/DAX memory allocation	Memory sharing (Multi-host FAM)
Conventional file systems					
• Storage is block device • Storage is allocate-on-write or delayed allocation • Preallocation supported (fallocate, etc.) • Free on last unlink (delete) • Mutated pages written-back to storage	• Data is demand-paged from storage into page cache • Mmap accesses data in page cache • Read/write copies to/from page cache • O_DIRECT I/O bypasses the page cache				

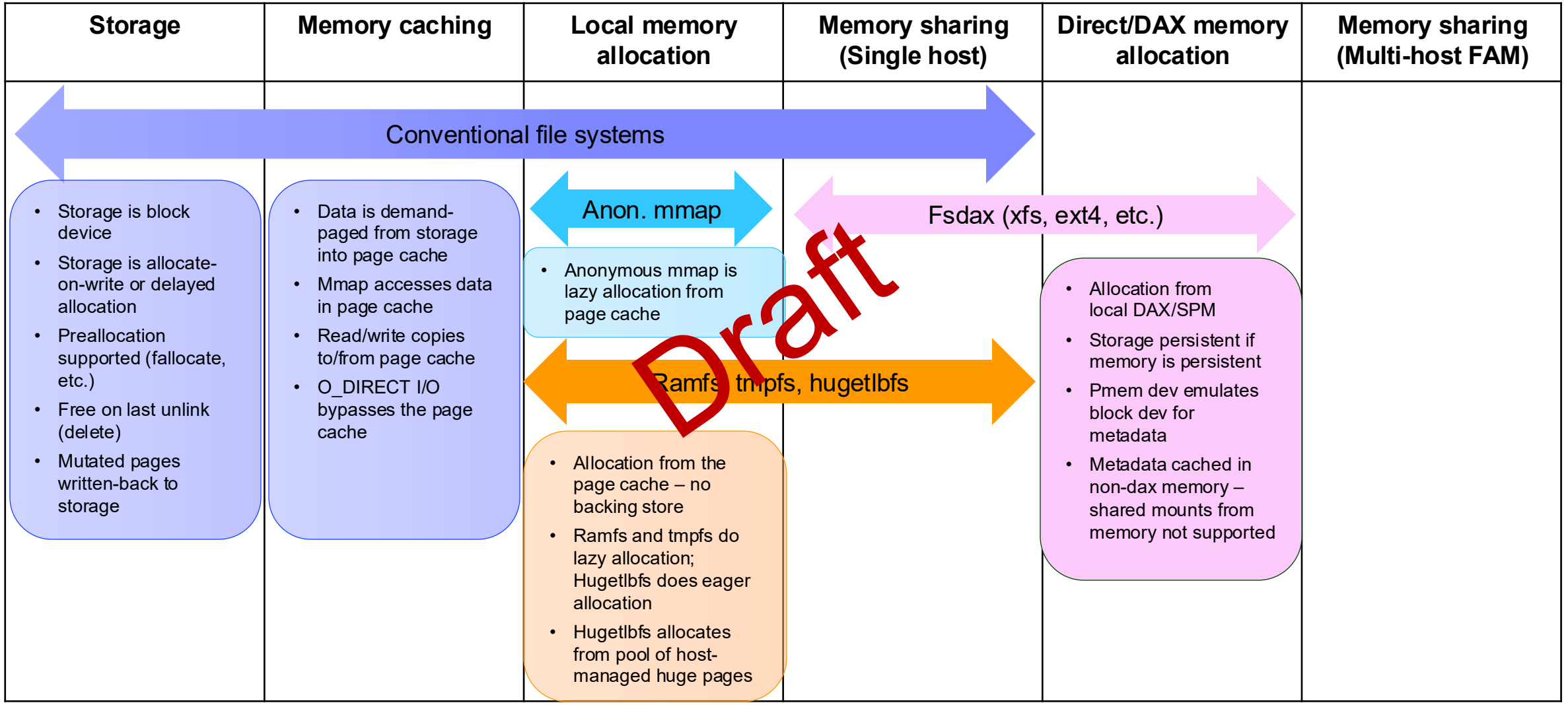
File system / VFS functionality

Storage	Memory caching	Local memory allocation	Memory sharing (Single host)	Direct/DAX memory allocation	Memory sharing (Multi-host FAM)
Conventional file systems					
• Storage is block device • Storage is allocate-on-write or delayed allocation • Preallocation supported (fallocate, etc.) • Free on last unlink (delete) • Mutated pages written-back to storage	• Data is demand-paged from storage into page cache • Mmap accesses data in page cache • Read/write copies to/from page cache • O_DIRECT I/O bypasses the page cache	Anon. mmap • Anonymous mmap is lazy allocation from page cache			

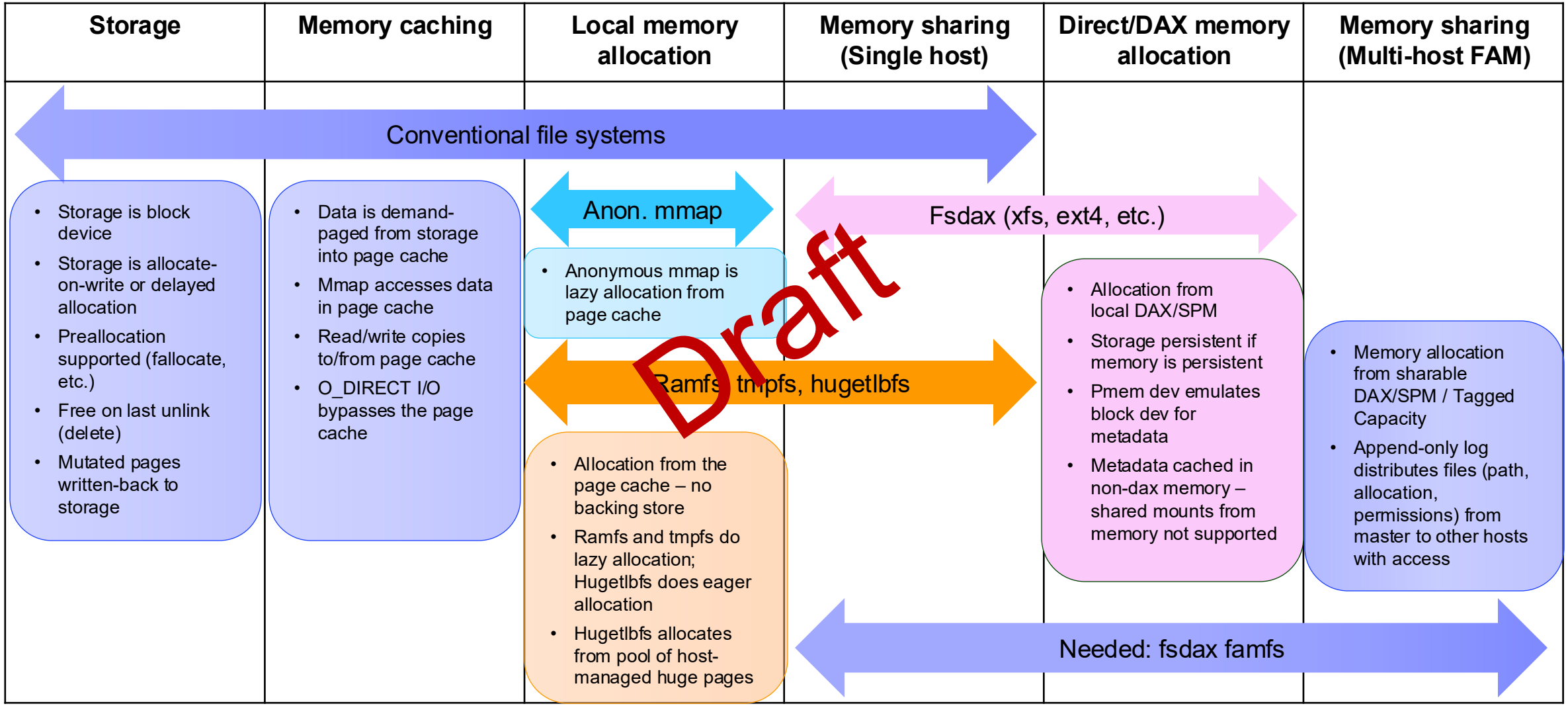
File system / VFS functionality

Storage	Memory caching	Local memory allocation	Memory sharing (Single host)	Direct/DAX memory allocation	Memory sharing (Multi-host FAM)
Conventional file systems					
• Storage is block device • Storage is allocate-on-write or delayed allocation • Preallocation supported (fallocate, etc.) • Free on last unlink (delete) • Mutated pages written-back to storage	• Data is demand-paged from storage into page cache • Mmap accesses data in page cache • Read/write copies to/from page cache • O_DIRECT I/O bypasses the page cache	Anon. mmap • Anonymous mmap is lazy allocation from page cache			
		Ramfs, tmpfs, hugetlbfs • Allocation from the page cache – no backing store • Ramfs and tmpfs do lazy allocation; Hugetlbfs does eager allocation • Hugetlbfs allocates from pool of host-managed huge pages			

File system / VFS functionality



File system / VFS functionality



Draft

micron Intelligence Accelerated™