Co-Designing Raft + Thread-Per-Core Model for the Kafka-API

https://github.com/vectorizedio/redpanda

background

alex gallego
@emaxerrno

- developer, founder & CEO of Vectorized, hacking on Redpanda, a modern streaming platform for mission critical workloads.
- previously, principal engineer at Akamai; co-founder & CTO of concord.io, a high performance stream processing engine built in C++ and acquired by Akamai in 2016

agenda

observation 1. hardware is fundamentally different than it was a decade ago

observation 2. new bottleneck is CPU. everything is asynchronous

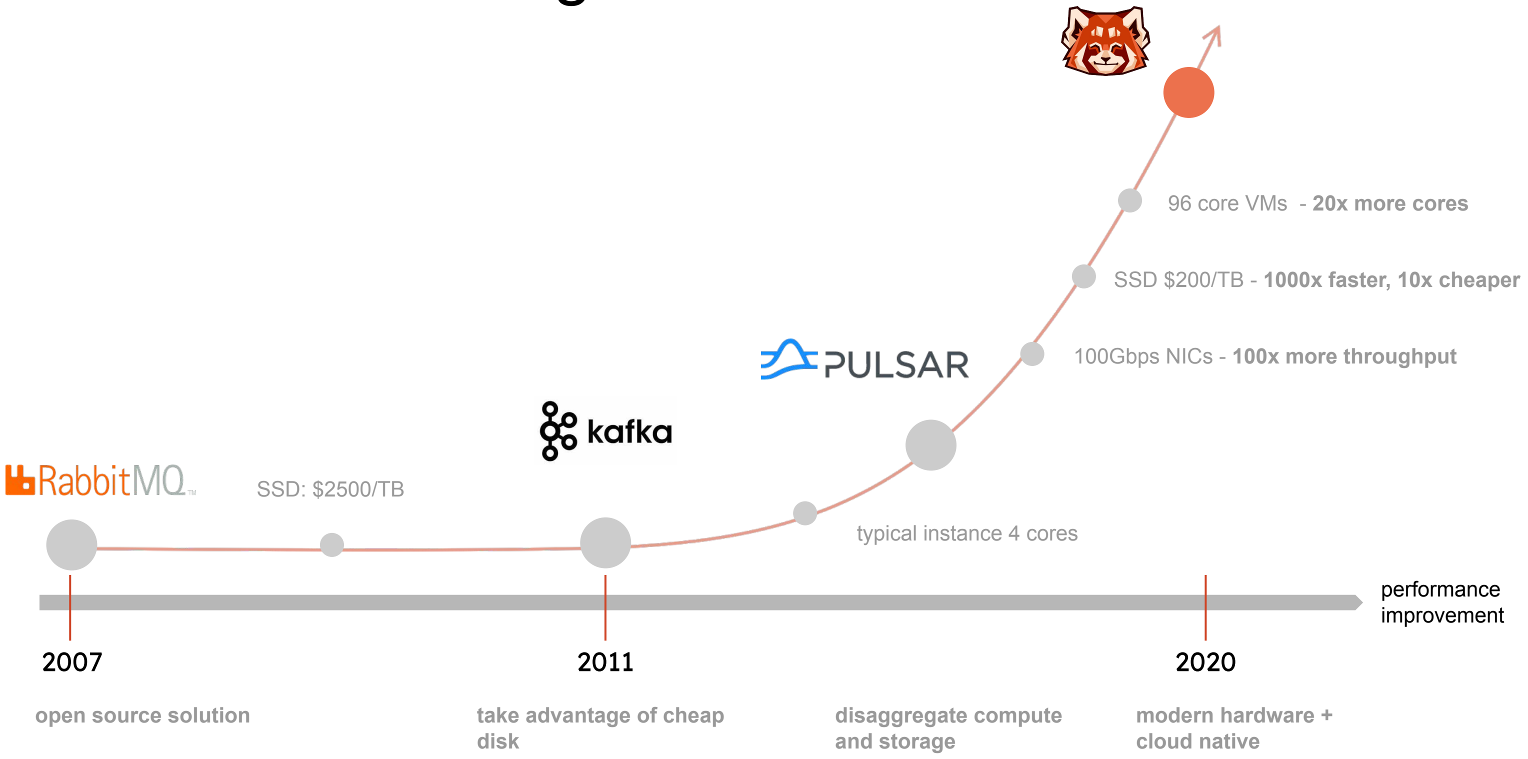
conclusion: need a new way to build software

practical impl: we implemented redpanda - a new storage engine - from scratch with the principles that we'll cover here & achieve 10-100x better tail latencies; src code on

- sometimes you get to reinvent the wheel when the road changes
    - hardware is fundamentally different
        - **1000x faster disks**
        - **100x cheaper disks**
        - **20x taler machines** (225 vCPU on GCP)
        - 100x higher throughput NICs (100Gbps is common)

observation 1:

evolution of streaming



observation 2:

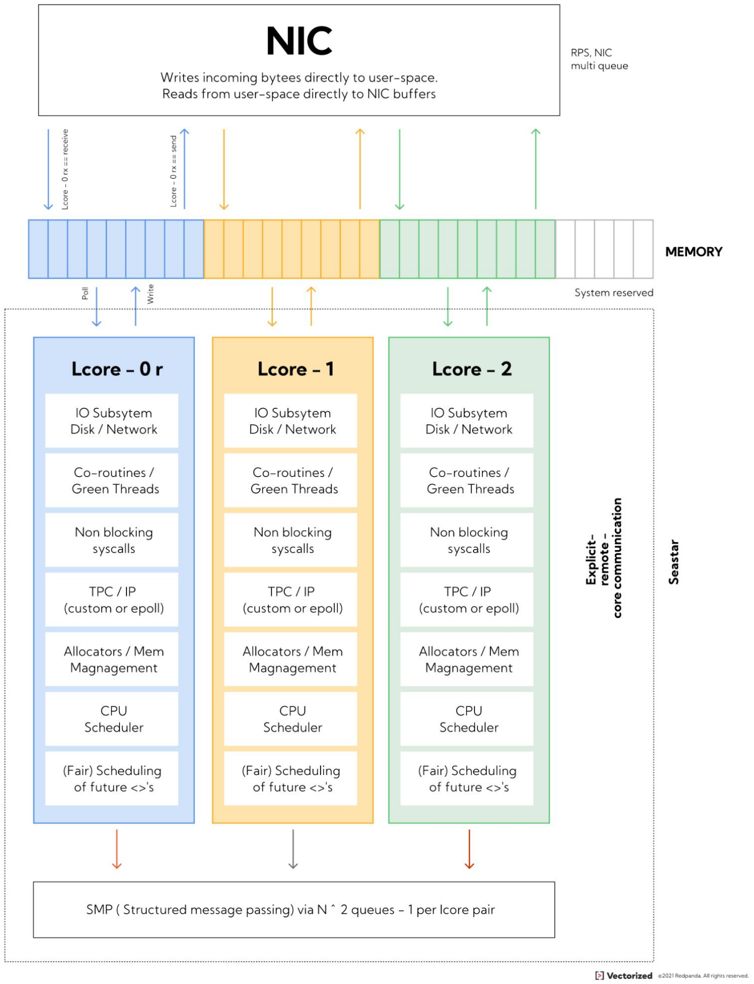
everything is async; cpu the new bottleneck

at 3GHz
(3 billion instructions per second)

1 DMA page write -> 20-140us

1 blocking page write
-> 20-140us (x 3 Million)
-> 60-420M clock cycles wasted

western digital nvme ssd
1.2GB/s writes



thread per core architecture

- explicit scheduling everywhere
  - IO groups
  - x-core groups (smp)
  - memory throttling
- **ONLY supports async interfaces**
  - **requires library re-writes for threading model to work well**

new way to build software:

async-only

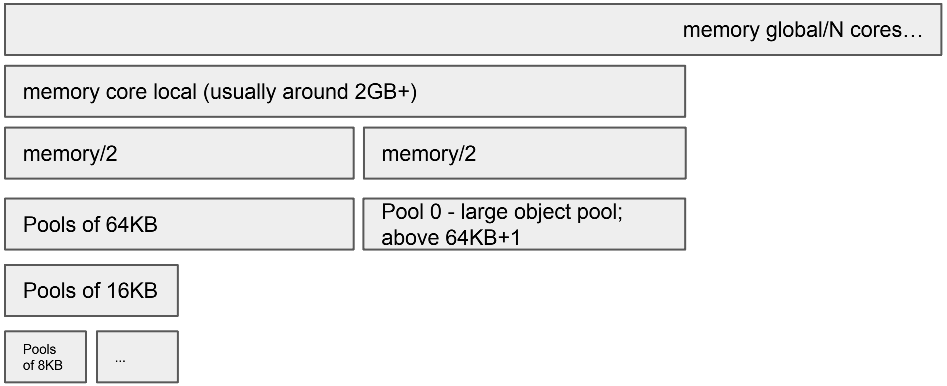
cooperative scheduling framework

future<>

- **viral primitive** (like actors, Orleans, Akka, Pony, etc) - mix, map-reduce, filter, chain, fail, complete, generate, fulfill, sleep, expire futures, etc
- **fundamentally about program structure**. w/ concurrent structure, parallelism is a free variable
- **one pinned thread per core** - must express parallelism and concurrency explicitly
- **no locks on the hotpath - network of SPSC queues**

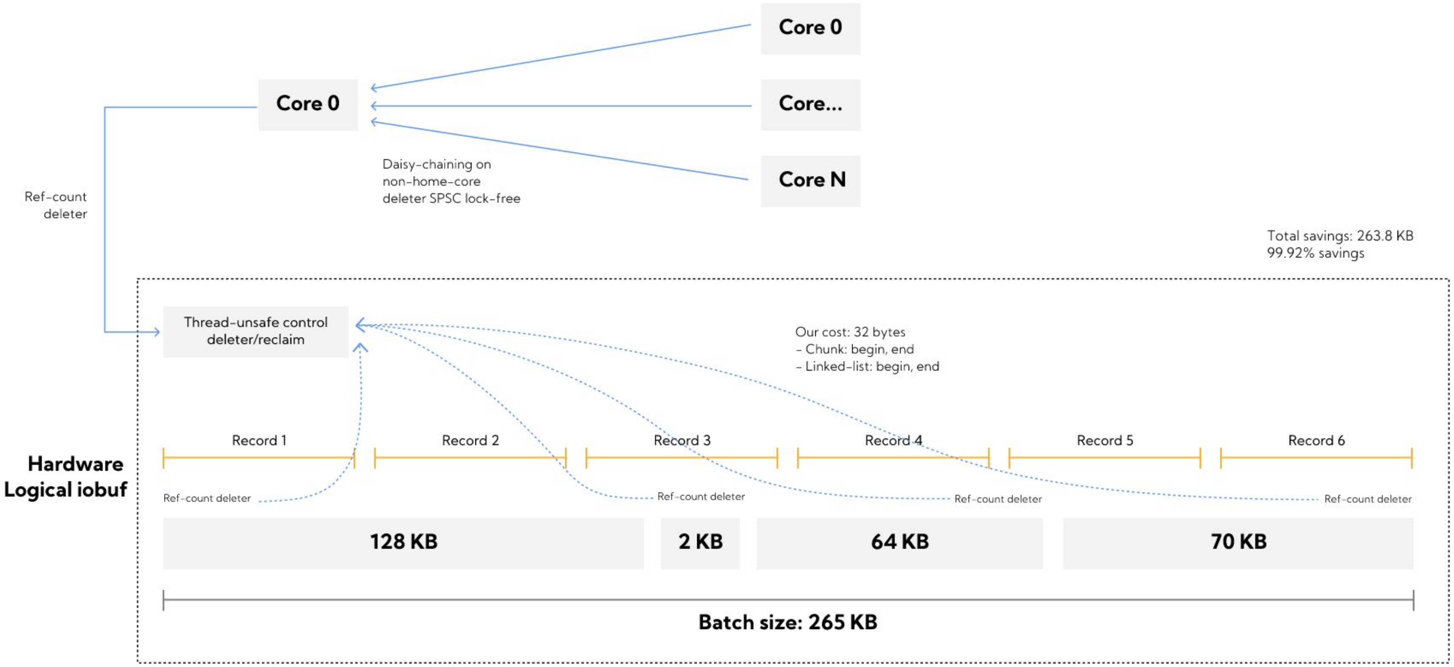
technique 1:
no virtual memory

buddy allocator



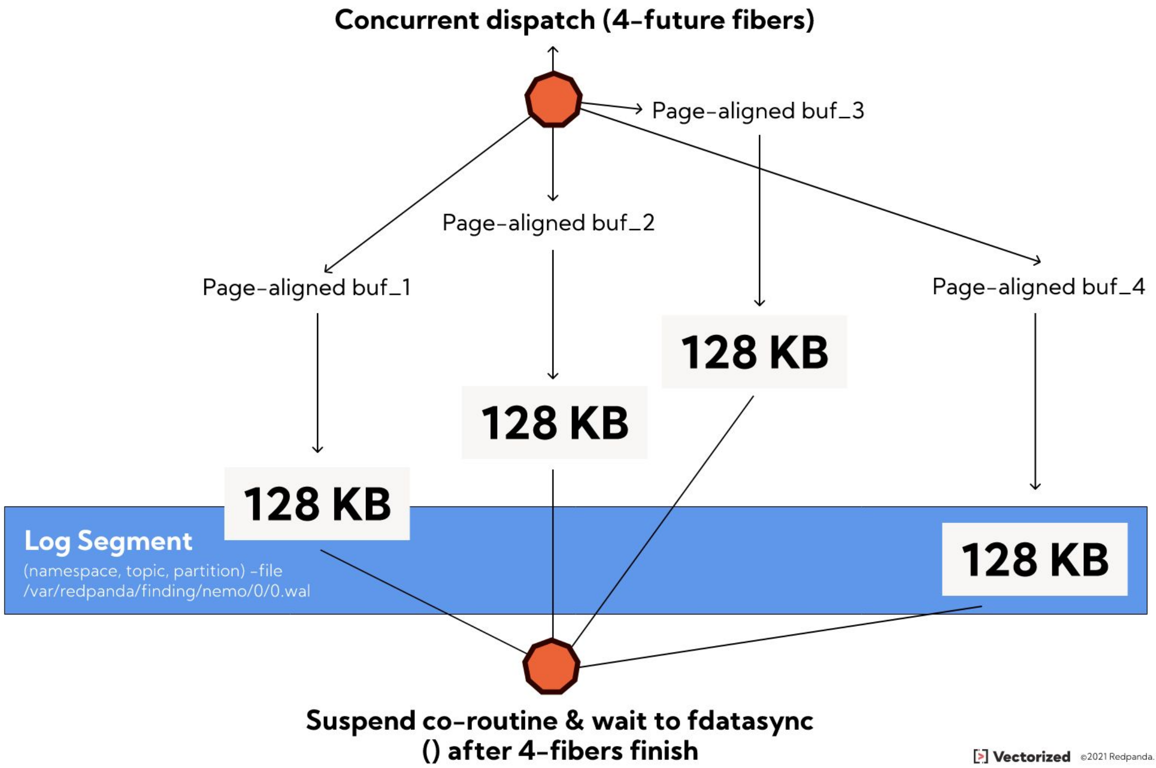
- preallocate 100% of mem; split across N-cores for thread-local allocation/access
- create pools by dividing the memory one layer above/2 and creating a new pool
- large allocations (above 64KB are not pooled)
- buddy allocator pools for all object sizes below 64KB
- full free-lists are recycled
- difficult to use this technique in practice, and requires developer retraining/accounting for every single byte present in the system at all times
  - forces developer to pay additional attention to all hash-maps, allocations, pooling, etc

technique 2:
iobuf - TPC buffer management



src: https://vectorized.io/blog/tpc-buffers/

technique 3:
out of order dma writes



technique 4:

no page cache - embed domain knowledge

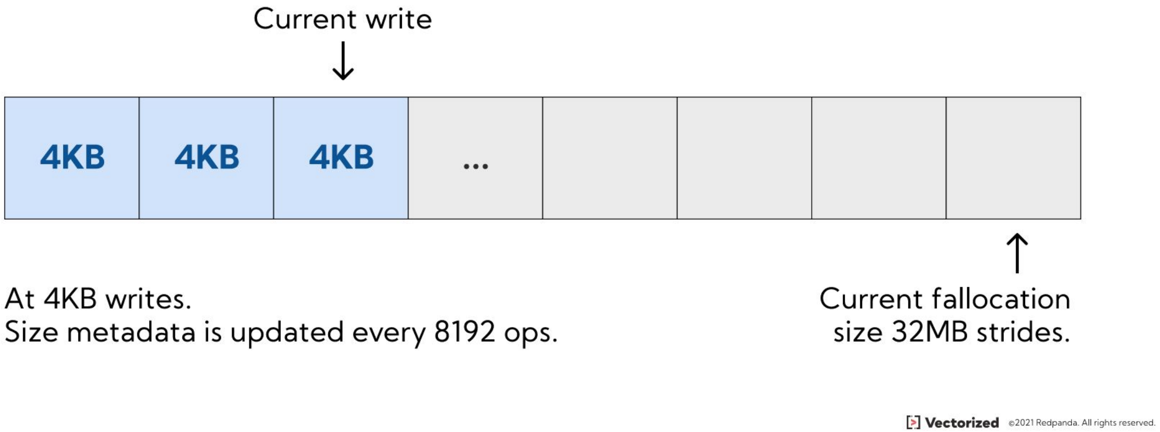
- the linux kernel page cache introduces non-determinism in the IO path
- page cache uses global lock **per file-object**
    - This is a very smart thing to do for generic scheduling of IO - DMA is hard to get right
- page cache is never a bad choice, but not always a good choice.
    - Always a good **middle ground**
    - Introduces generic read-ahead semantics (for our workload specific project)
- hard to understand failure semantics (specific to version) leads to **hard to track correctness bugs** (see pgsql bug)

- **thread-local object cache** instead
- format ready to go onto the wire instead
- no translation necessary
- stats for file write latency influence application level eager backpressure

```
length: varint
attributes: int8
    bit 0~7: unused
timestampDelta: varlong
offsetDelta: varint
keyLength: varint
key: byte[]
valueLen: varint
value: byte[]
Headers => [Header]
```

technique 5:

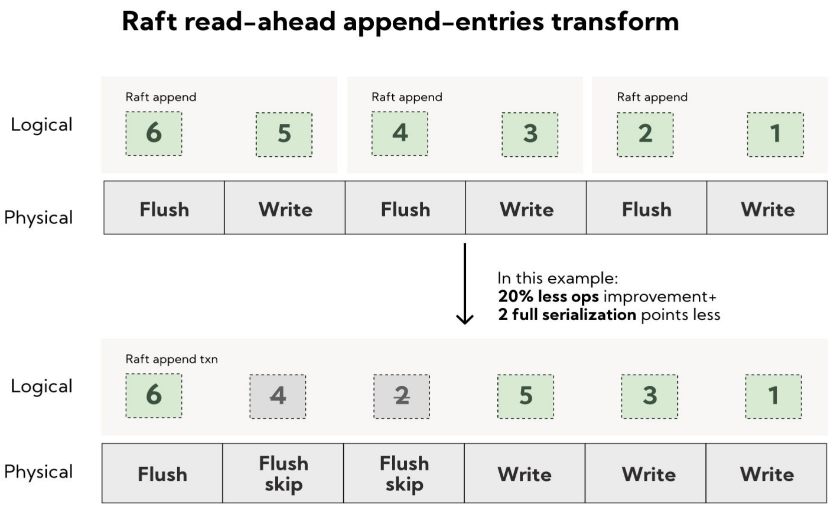
adaptive fallocation



- reduce metadata contention
- use fdatasync vs fsync
- 20% latency improvement
- ahead-of-time metadata update

technique 6:
raft read-ahead op dispatching

- artificially debounce writes by 4ms
- scan the ops & drop flushes
- if any op required a flush, do it at the end
  - higher buffer utilization
  - higher hardware utilization
  - lower latency
  - skips full disk-level barriers (fdatasync)



technique 7:

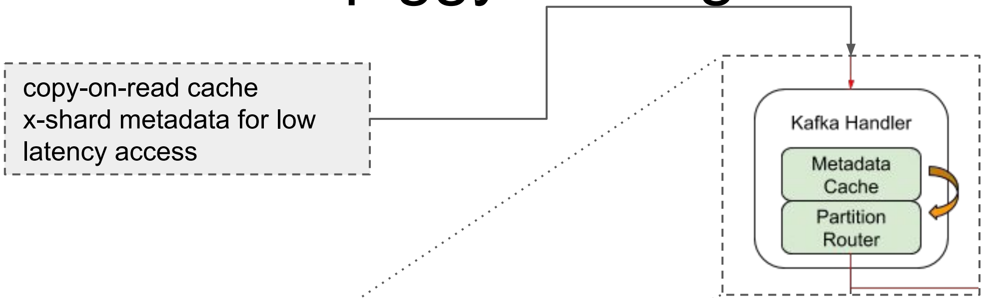
request pipelining per partition

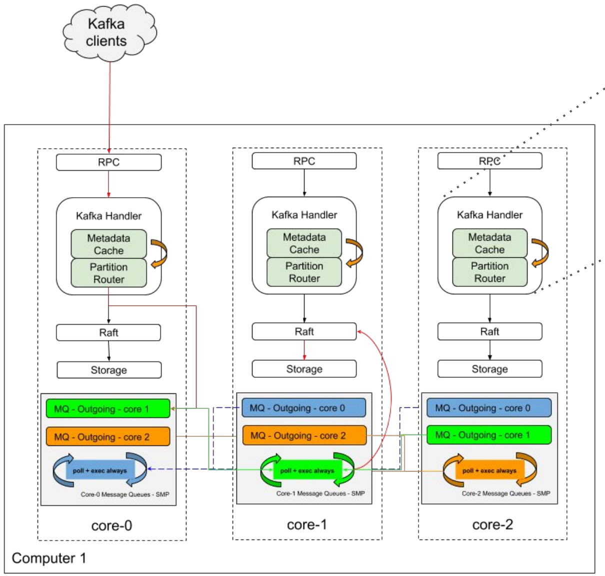
- parallelism model == number of cores/pthreads in the system
- read full request metadata and assign subrequest to physical core
- for all non-overlapping cores, execute in parallel
- for all overlapping cores per *partition* pipeline (enqueue writes in order)

technique 8:

core-local metadata piggybacking

(...pandabacking?)



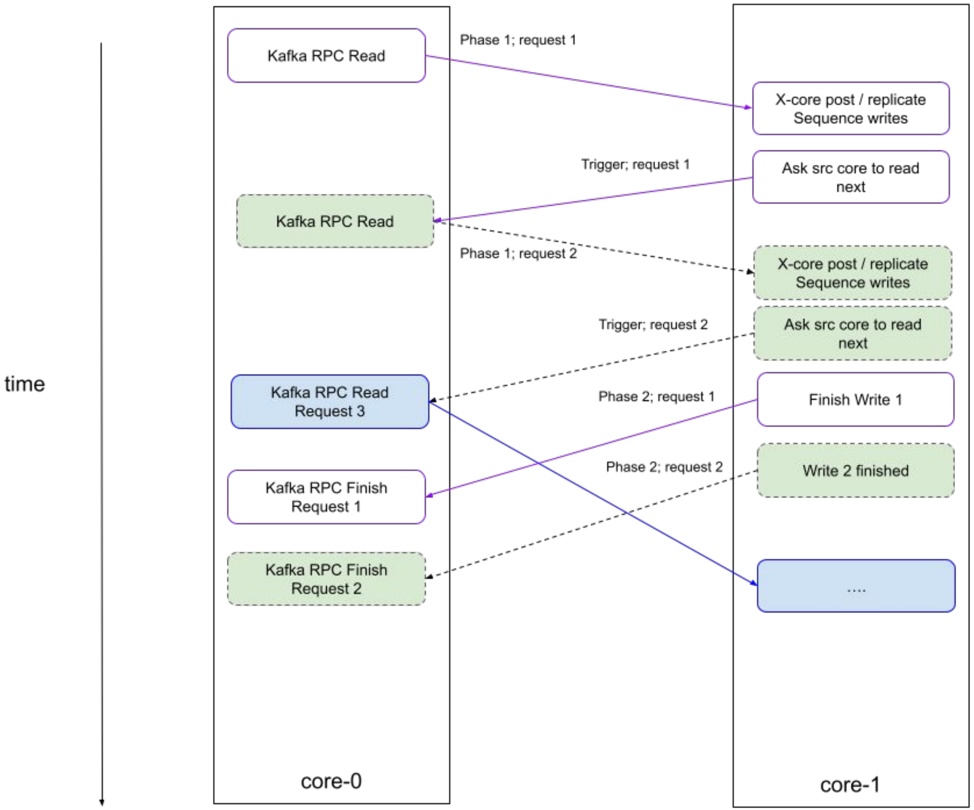


- maintain core-local metadata cache of
  - bytes written per partition (for future readers)
  - latencies from the remote core (could be highly contended and we need TCP backpressure)
  - per TCP-connection read-ahead pointers on disk for O(1) access/assignment

technique 9:

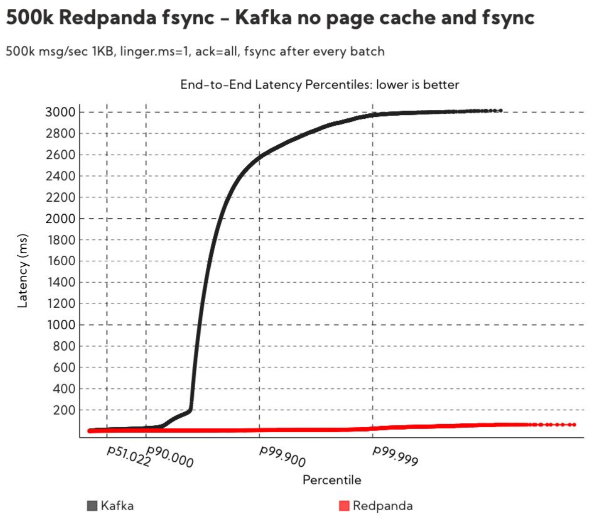
2phase+trigger cross-core write-request splitting

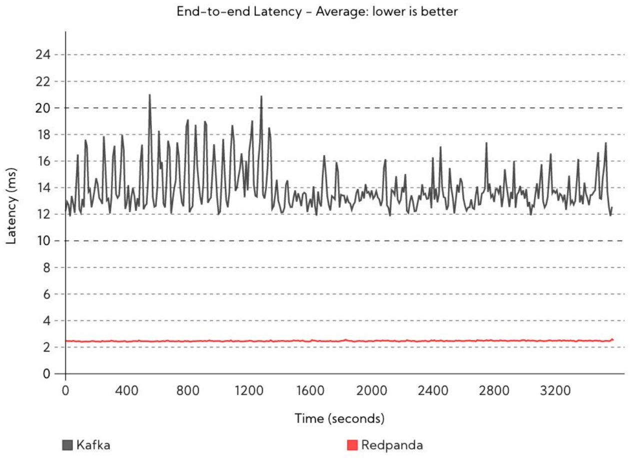
- First Stage
  - on the src core, dispatch write on destination core & return when data is sequenced inside raft/disk which establishes order (but not acknowledged)
- Second Stage
  - Once sequenced on destination core
  - Background x-core message to the src core that signals the src core it can go ahead and send the next produce now
  - Effect
    - cross-core pipelining
    - 10x improvement for contended resources
  - Waiting on acks/flushes etc can happen while the next request is sequenced



check out the code for yourself!

- https://github.com/vectorizedio/redpanda
- ask questions from the maintainers at https://vectorized.io/slack
- say hi on twitter https://twitter.com/vectorizedio





src: https://vectorized.io/blog/fast-and-safe/