CMS Data Formats and Impact on Federated Access

Brian Bockelman

XRootD Workshop @ UCSD; January 2015

What Affects Performance?

- **Bandwidth**: Sustainable transfer rate between client and server.
- **Latency**: Minimum time for a network round-trip between client and server.
- **Data Rate**: Amount of data processed by application divided by CPU time.
- **Latency sensitivity**: Frequency of the application being blocked on IO.
- While the first two are characteristics of the federated data store, the second two are characteristics of the computing model and software.
    - I'll focus on the latter two for this talk.

Introduction to our Data Formats

- Vocabulary for this talk:
  - **RAW**: Unprocessed detector data.
  - **GEN-SIM**: Simulated physics event data.
  - **RECO**: Basic reconstructed physics objects with some relevant detector performance information.
  - **AOD**: Physics objects useful for analysis.
  - **MiniAOD**: Refined physics useful for the majority of analyses; optimized for size.
  - **User** / ROOT-based data formats: Anything else a user may dream up as output of their jobs.
  - **Other**: Various small non-ROOT files used for input (XML files, generator input).

Broad Strokes

- All “EDM files”: One large event TTree and a small number of auxiliary trees (various metadata, such as run/lumi info).
- **RAW**: A small number (~10) of large branches. Data is mostly binary blobs. No splitting. About 300KB / event. ZLIB-compressed.
- **RECO**: <300 top-level objects. Split level 1. A small number of large branches. About 1MB / event. ZLIB-compressed.
- **AOD**: <300 top-level objects, fully split. Many objects are complex C++ objects: slow to deserialize (same as in RECO). About 300KB / event. LZMA-compressed.
- **MiniAOD**: ~100 top-level objects. Keeps the most popular collections and applies reasonable energy thresholds. Optimized for size and a little for speed; about 30KB / event. ZLIB-compressed.
- **User**: Who knows? Typically, ntuples with 30-100 top level-branches. Most branches are extremely simple. .5-30KB / event.
- Note: I quote Run1 numbers; event sizes inflate proportionally for Run2, but don’t particularly affect performance.

ROOT Wisdom

- ROOT optimizes for single TTree access.
  - Woe be to those with multiple active TTrees.
- It's small, but there is a per-branch & basket memory and performance overhead.
  - Rule of thumb - try to keep # of branches <2k. **Never go above 10k**. Woe be to those with 20k branches!
  - In CMS data formats, we see about 5-10MB of RAM used per 1GB of TTree. This sets a (loose) upper bound on the file sizes we could produce.

Computing Models Matter!

- Regardless of how well-designed the data format is, the ratio of CPU cycles to bytes read is relevant!
    - At some point, you *do* run out of bandwidth.
- Physics-heavy algorithms - reconstructing all or part of the event or simulation - keep the overall data rate down.
    - Event selection (slimming & skimming)
- In CMS, end-stage analyses — where data rates are at the highest - is done at the local user's site.
    - **Data rate for grid jobs is <1MB/s** on average.
    - Reconstruction / signal events are about 40KB/s. Analysis jobs are 250-500KB/s.
- Biggest issue is pileup - background events that must be mixed with simulated signal. If PU is 40, this means we need to read in ~40MB per simulated event.
    - Saving grace is that we can distribute background samples to all sites.

Software Matters Too

- In early 2009, I started working on CMSSW IO for a parochial reason - I wanted our Tier-2 site to have better IO performance.

- At the time, CMSSW IO averaged around 10 reads per event; we're now at 100 events per read.
    - The same amount of data is read, but we now read it in bigger chunks and in vectors of reads.

- By improving data structures and introducing different compression algorithms, we reduce the overall data rate.
    - However, *with the exception of pileup*, data rate is irrelevant in practice.

CMSSW IO - Wrapping ROOT Since 2008

- In CMS, no user has access to raw ROOT TTree objects.
    - All requests go through an intermediate layer in CMSSW. This allows to spy on user's logical requests (branches/events) and manage various ROOT caches.
- Similarly, CMS overrides all ROOT IO plugins.
    - Allows us to intercept IO requests and change them or do accounting.
- By controlling both the logical layer and the IO layer for all applications, work done by the experts benefits all users.

CMS Techniques - “Lazy-Download”

- File is divided into 256MB chunks; first time data from a chunk is accessed, it is downloaded to disk and unlinked.
    - All future accesses are done via mmap of the local file.
        - Latency becomes irrelevant - we end up dominated by bandwidth & data rate.
- For unoptimized workflows (merging using TTreeCloner) and poor performing storage.
    - Highly discouraged, however: becomes problematic to process large files and increases bandwidth. Jobs that need 10% of an event now read 100% of files.

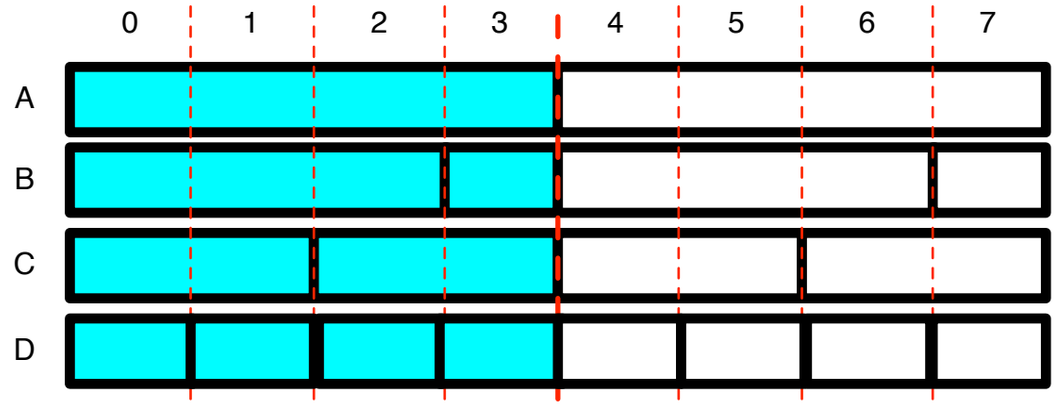
Caching/Prefetching

- By now, most HEP experiments have enabled TTreeCache. This prefetches a certain subset of branches after a “training period”.
    - The prefetch happens in data “clusters” of about 20MB. When the first event of a cluster is read, ROOT will issue reads for all events in the cluster for the relevant branches.
        - ROOT additionally supports overlapped IO - reading cluster A in a separate thread while cluster B is in use. CMS doesn’t use this - never been validated in our setup.
- CMS improves on the TTreeCache design by combining several caches together to tackle startup and cache miss costs.
    - Next few slides are from my 2013 CHEP talk.

Improved TTreeCache startup

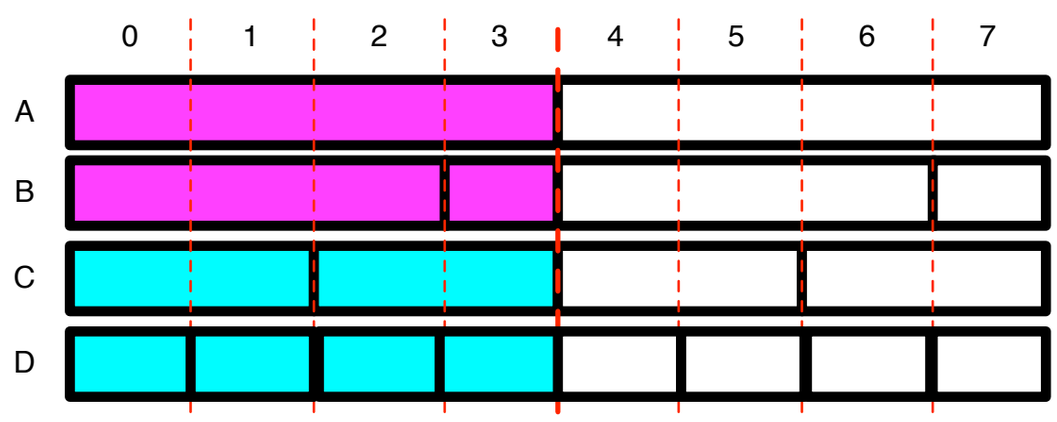
- When training, the TTreeCache reads out one basket at a time.
  - If the user reads 1000 branches on the first event, there are 1000 network round trips; on our test network, this is ~130second delay.
- We create a second TTreeCache which fetches all data for the first 20 events (or 20MB; whatever is smaller) and we separately record all branches used.
  - After the first 20 events, we manually train another TTreeCache which is used throughout the rest of the job run.
- Hence, the first 20 events typically are read with a *single* network round-trip.

Startup TTreeCache



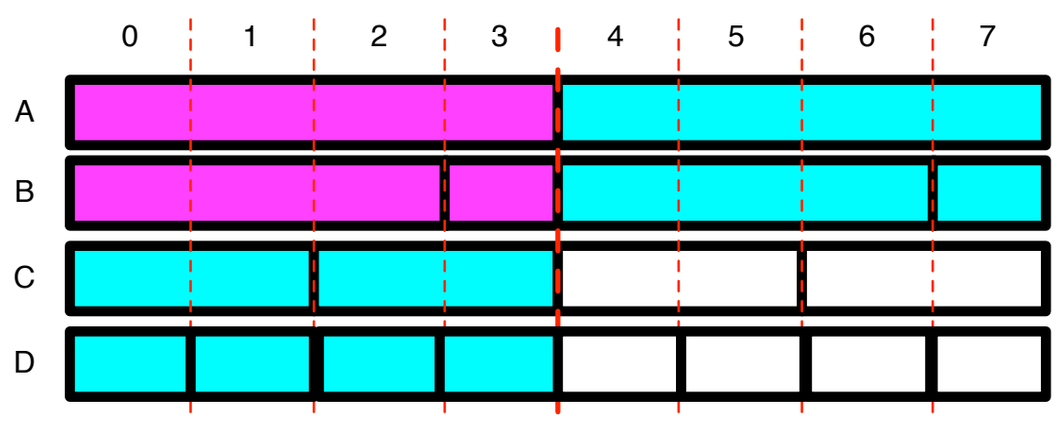
Read all baskets in first cluster using startup TTreeCache

Startup TTreeCache



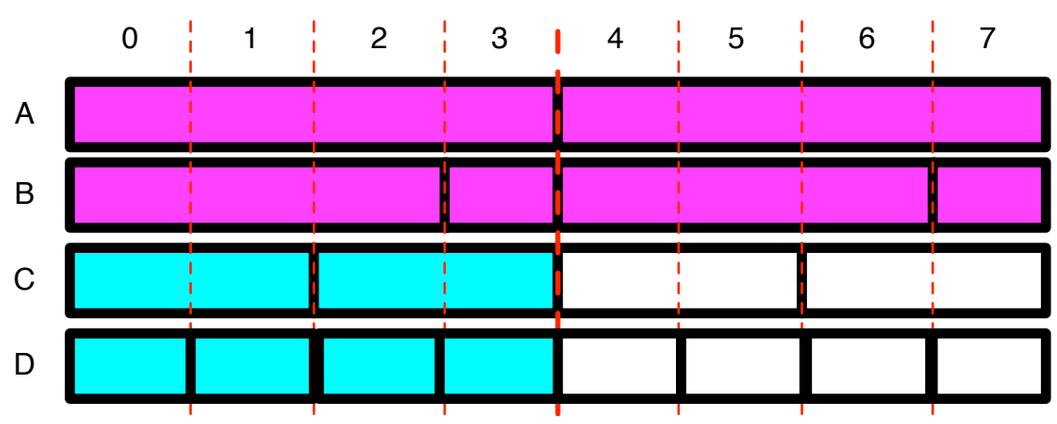
User reads from branches A & B
for first cluster

Startup TTreeCache



Regular TTreeCache prefetches branches A & B for second cluster

Startup TTreeCache



User reads from branches A & B for second cluster.

Total reads: 2
Default behavior: 4 reads.

Startup Cache Performance

- For local reads, removing the startup cache causes a 1.03x slowdown.
- For remote reads, removing this cache causes a 13.8x slowdown.
- The performance difference is all from the training period. For remote reads, when removing the cache, the first 100 events take 8 minutes. The next 100 events take six seconds.

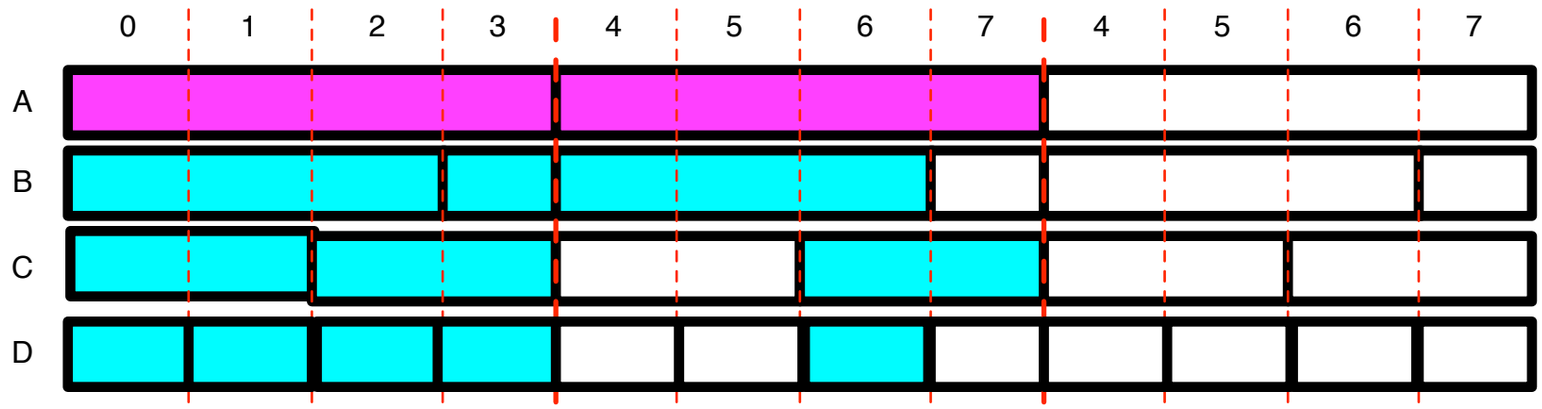
Trigger Pattern Optimization

- It is common for an analysis to read branches X, Y, and Z for each event; then, based on the contents of those branches, read out additional branches.
  - We cannot do prefetching for this case - the code has no way of knowing what will be used!
  - ROOT's default behavior is to do one I/O per basket if the user accesses a branch not in the cache.
- We again use a secondary TTreeCache: whenever we notice a cache miss will happen for the primary TTreeCache, we switch to the other TTreeCache, which reads all the missing baskets for the event.
  - How do we determine the missing baskets? The first time the "trigger" occurs, we prefetch all branches and record which ones were used.

Trigger pattern optimization

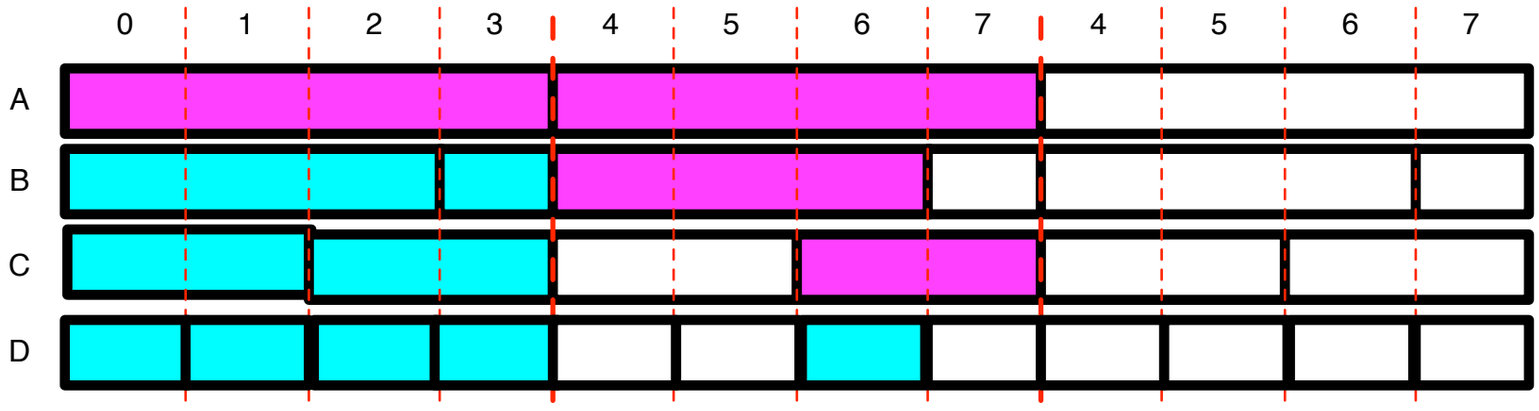
Suppose branch A is our physics trigger;
CMSSW notices object 6A is requested

Trigger pattern optimization



This is our first cache-miss.
Trigger cache prefetches all branches for event 6.

Trigger pattern optimization

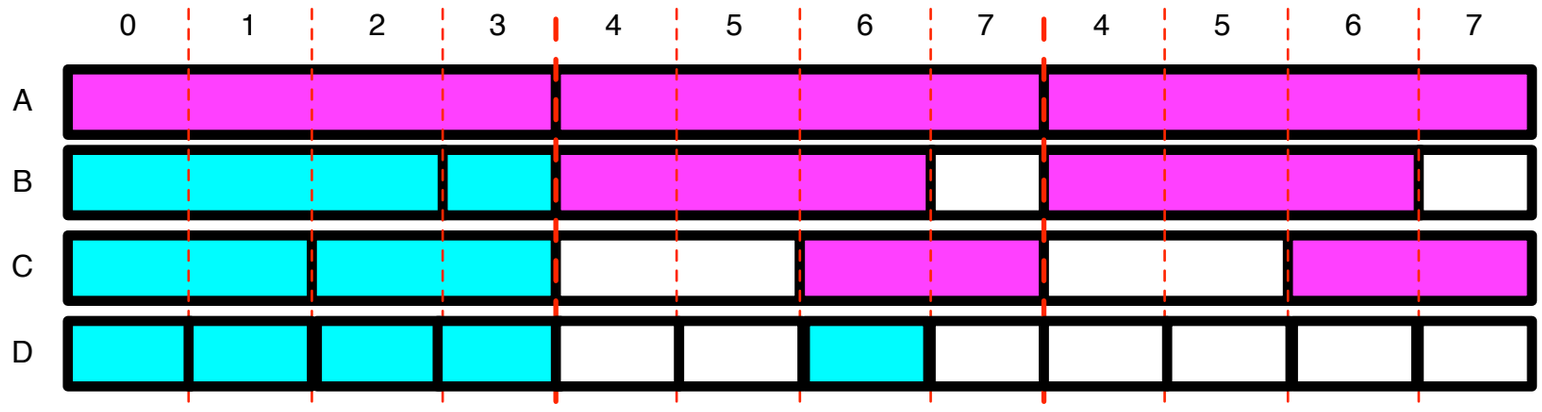


User reads branches B and C of event 6.

Trigger pattern optimization

Trigger fired for event 6;
Last time, only branches B and C were used.
We only pre-fetch those.

Trigger Pattern Optimization



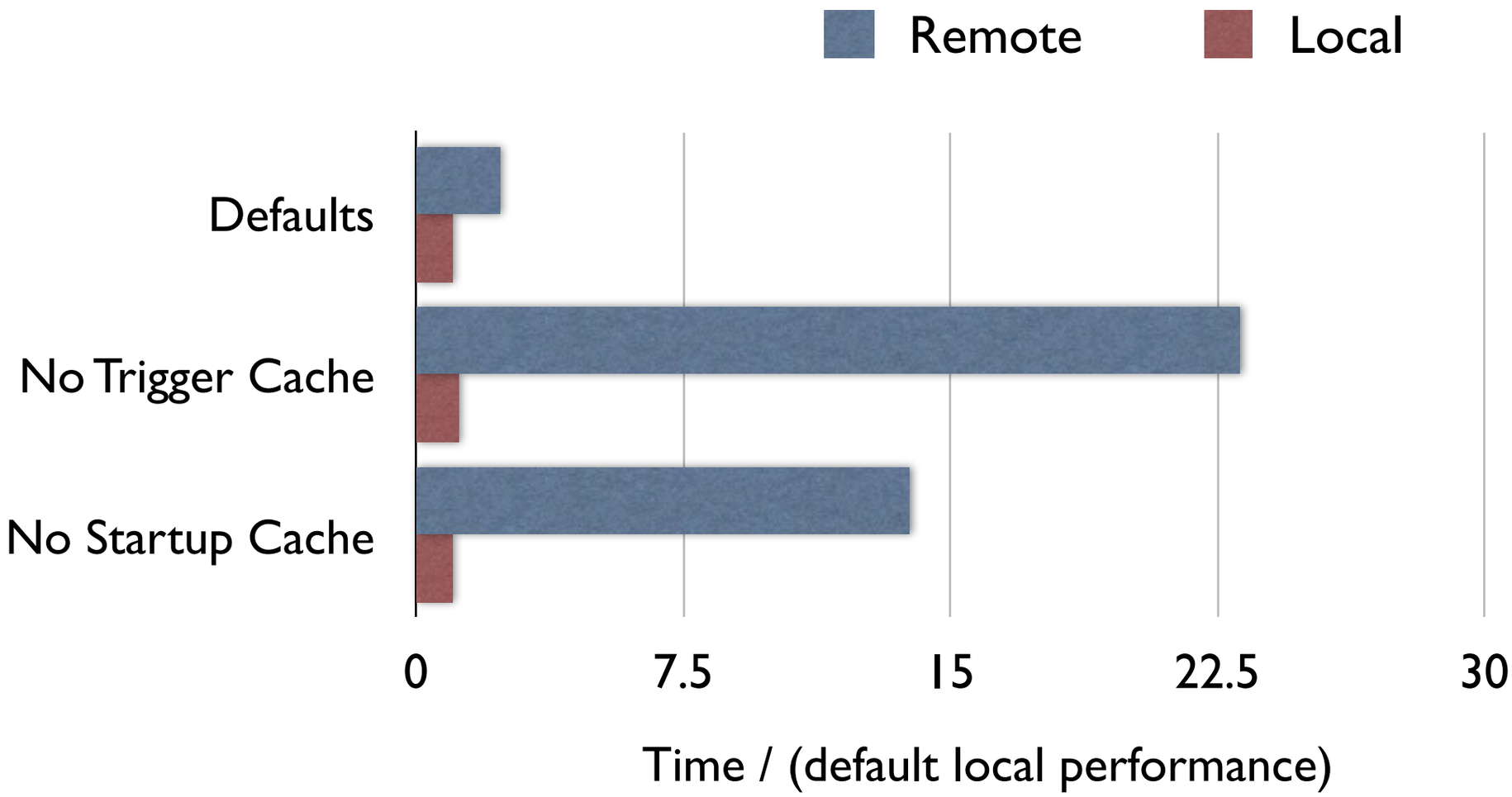
Total reads: 5

ROOT default reads: 7

Trigger Cache

- Disabling the trigger cache incurs a 1.2x slowdown on the local network (compared to the normal CMSSW).
- Disabling the trigger cache incurs a 23x slowdown on the remote network.

Summary - Avoiding Network Round Trips



Not shown: ROOT defaults (no TTreeCache) reading remotely is 177x slower than CMSSW's defaults reading locally!

Run 2 - Multisource Client

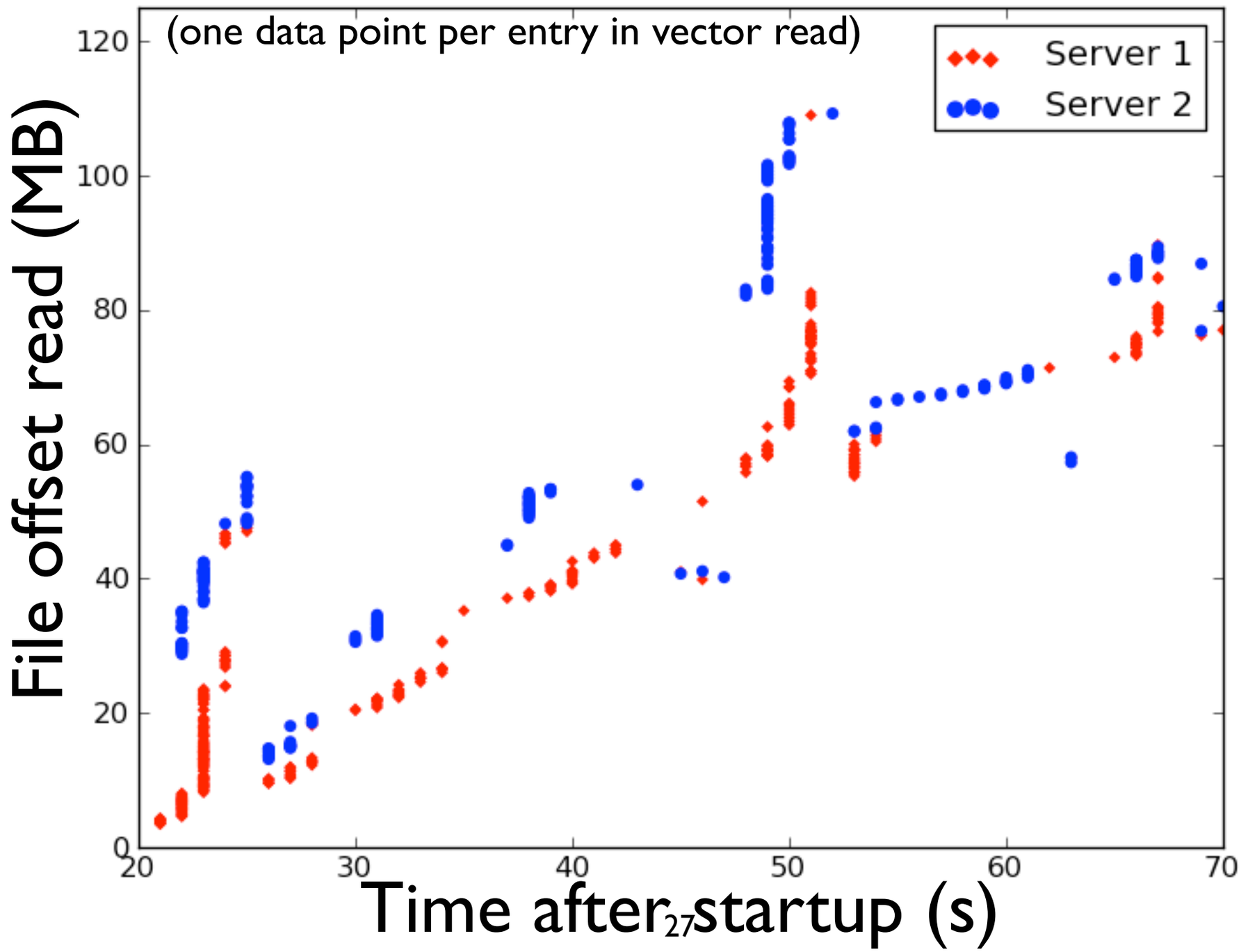
- Long in development, but recently merged for Run2 is the multi-source client.
- Built on top of the new Xrootd client (XrdCl), this client maintains three list of sources for an open file:
    - **Active sources**: Up to 2 servers we use for every read request.
    - **Inactive sources**: Additional sources that are functioning but not used due to worse performance issues.
    - **Disabled sources**: Sources that previously had an error.
- The client will randomly probe - at a very small rate - for additional sources and performance changes in the inactive sources.
    - Will be more aggressive about locating sources at startup under the theory that the cmsd location cache is “hot”.

Multisource Client Basics

- If there are two active sources, when a read / readv is issued:
  - Request is split in two. Care is taken to make both blocks are as contiguous as possible.
    - Relative size of requests is a function of the recent performance of the servers. Idea is to balance total time.
- Both requests are issued simultaneously. We block on a std::future object until both callbacks occur.
- Control is returned to CMSSW/ROOT once both callbacks finish.
  - CMSSW/ROOT sees none of the parallelism.

Multisource Illustration

Read offset versus time, per source



Multisource Client - Error Recovery

- On client errors with a file open:
  - We retry the open up to 5 times, each time banning the previous bad source. Helps to avoid broken data servers with functioning cmsd.
    - **NOTE: EOS does not obey the blacklist of fileservers**. :(
  - We do not retry if the error came from the redirector itself.
- On file read errors: We set the active source as disabled and, if we are out of active sources, try a reopen immediately.

Improving the Experience

- The multisource client is more aggressive about retries and mitigating problems. Robustness has its downsides!
    - Analogous to TCP: TCP streams will always work, even if it means it degrades your bandwidth by 99.9% without telling you! There's no haptic feedback when something goes wrong.
- In the single source client, it's hard/impossible to figure out reason for errors.
    - Operators always blame AAA (even when it's clearly site-specific). AAA always blames the sites ;)
- Accordingly, we inform the user (via stdout) whenever the multi-source client switches sites.
- We also record a reasonable message to stdout whenever a recoverable error occurs.
- All fatal failures are turned into CMS Exceptions. All information about the current connections, past connections, recovery attempts, error messages, etc, are recorded in the exception.
    - Idea is that users just need to send us the exception message and we can immediately determine the source of the error.

TODO: Monitoring

- The multisource client has made a mess of our client-side monitoring.
    - Before, we sent a single UDP packet to a collector at file close. This packet contained an IO summary (server, client, source site, bytes read, % of file read, readv bytes, etc).
    - Now, we need a *packet per file per server* host.
    - Not necessarily a hard task, but illustrates another place where a more robust client makes the system difficult to understand.

Future Work

- There is little CMS-specific — or Xrootd protocol specific — in our multisource client. Porting is not trivial, however:
    - Implementation depends heavily on an asynchronous / callback-based client API. The DAVIX HTTP client would be the right one to build upon, but it is a synchronous API.
    - Implementation depends strongly on C++11 - atomics, lambdas, std::futures. No pthreads. Neither ROOT nor Xrootd have these.
    - Everything is possible, but effort is running out on AAA grant. Perhaps the best bet for porting would be with one of the **art**-based neutrino experiments (**art** started as a fork of CMSSW).
- Enabling overlapped IO in ROOT is the most likely next target.