6.824 2014 Lecture 4: FDS Case Study

Flat Datacenter Storage
Nightingale, Elson, Fan, Hofmann, Howell, Suzue
OSDI 2012

why are we looking at this paper?
  Lab 2 wants to be like this when it grows up
    though details are all different
  fantastic performance -- world record cluster sort
  good systems paper -- details from apps all the way to network

what is FDS?
  a cluster storage system
  stores giant blobs -- 128-bit ID, multi-megabyte content
  clients and servers connected by network with high bisection bandwidth
  for big-data processing (like MapReduce)
    cluster of 1000s of computers processing data in parallel

high-level design -- a common pattern
  lots of clients
  lots of storage servers ("tractservers")
  partition the data
  master ("metadata server") controls partitioning
  replica groups for reliability

why is this high-level design useful?
  1000s of disks of space
    store giant blobs, or many big blobs
  1000s of servers/disks/arms of parallel throughput
  can expand over time -- reconfiguration
  large pool of storage servers for instant replacement after failure

motivating app: MapReduce-style sort
  a mapper reads its split 1/Mth of the input file (e.g., a tract)
    map emits a <key, record> for each record in split
    map partitions keys among R intermediate files (M*R intermediate files in total)
  a reducer reads 1 of R intermediate files produced by each mapper
    reads M intermediate files (of 1/R size)
    sorts its input
    produces 1/Rth of the final sorted output file (R blobs)
  FDS sort
     FDS sort does not store the intermediate files in FDS
     a client is both a mapper and reducer
     FDS sort is not locality-aware
        in mapreduce, master schedules workers on machine that are close to the data
        e.g., in same cluster
     later versions of FDS sort uses more fine-grained work assignment
       e.g., mapper doesn't get 1/N of the input file but something smaller
       deals better with stragglers

The Abstract's main claims are about performance.
   They set the world-record for disk-to-disk sorting in 2012 for MinuteSort
      1,033 disks and 256 computers (136 tract servers, 120 clients)
      1,401 Gbyte in 59.4s

Q: does the abstract's 2 GByte/sec per client seem impressive?
   how fast can you read a file from Athena AFS? (abt 10 MB/sec)
   how fast can you read a typical hard drive?
   how fast can typical networks move data?

```
Q: abstract claims recover from lost disk (92 GB) in 6.2 seconds
   that's 15 GByte / sec
   impressive?
   how is that even possible? that's 30x the speed of a disk!
   who might care about this metric?

what should we want to know from the paper?
  API?
  layout?
  finding data?
  add a server?
  replication?
  failure handling?
  failure model?
  consistent reads/writes? (i.e. does a read see latest write?)
  config mgr failure handling?
  good performance?
  useful for apps?

* API
  Figure 1
  128-bit blob IDs
  blobs have a length
  only whole-tract read and write -- 8 MB

Q: why are 128-bit blob IDs a nice interface?
   why not file names?

Q: why do 8 MB tracts make sense?
   (Figure 3...)

Q: what kinds of client applications is the API aimed at?
   and not aimed at?

* Layout: how do they spread data over the servers?
  Section 2.2
  break each blob into 8 MB tracts
  TLT maintained by metadata server
    has n entries
    for blob b and tract t, i = (hash(b) + t) mod n
    TLT[i] contains list of tractservers w/ copy of the tract
  clients and servers all have copies of the latest TLT table

Example four-entry TLT with no replication:
  0: S1
  1: S2
  2: S3
  3: S4
  suppose hash(27) = 2
  then the tracts of blob 27 are laid out:
  S1: 2 6
  S2: 3 7
  S3: 0 4 8
  S4: 1 5 ...
  FDS is "striping" blobs over servers at tract granularity

Q: why have tracts at all? why not store each blob on just one server?
   what kinds of apps will benefit from striping?
   what kinds of apps won't?
```

Q: how fast will a client be able to read a single tract?

Q: where does the abstract's single-client 2 GB number come from?

Q: why not the UNIX i-node approach?
store an array per blob, indexed by tract #, yielding tractserver
so you could make per-tract placement decisions
e.g. write new tract to most lightly loaded server

Q: why not hash(b + t)?

Q: how many TLT entries should there be?
how about n = number of tractservers?
why do they claim this works badly? Section 2.2

The system needs to choose server pairs (or triplets &c) to put in TLT entries
For replication
Section 3.3

Q: how about
0: S1 S2
1: S2 S1
2: S3 S4
3: S4 S3
...
Why is this a bad idea?
How long will repair take?
What are the risks if two servers fail?

Q: why is the paper's n^2 scheme better?
TLT with n^2 entries, with every server pair occuring once
0: S1 S2
1: S1 S3
2: S1 S4
3: S2 S1
4: S2 S3
5: S2 S4
...
How long will repair take?
What are the risks if two servers fail?

Q: why do they actually use a minimum replication level of 3?
same n^2 table as before, third server is randomly chosen
What effect on repair time?
What effect on two servers failing?
What if three disks fail?

* Adding a tractserver
To increase the amount of disk space / parallel throughput
Metadata server picks some random TLT entries
Substitutes new server for an existing server in those TLT entries

* How do they maintain n^2 plus one arrangement as servers leave join?
Unclear.

Q: how long will adding a tractserver take?

Q: what about client writes while tracts are being transferred?
receiving tractserver may have copies from client(s) and from old srvr
how does it know which is newest?

Q: what if a client reads/writes but has an old tract table?

* Replication
  A writing client sends a copy to each tractserver in the TLT.
  A reading client asks one tractserver.

Q: why don't they send writes through a primary?

Q: what problems are they likely to have because of lack of primary?
   why weren't these problems show-stoppers?

* What happens after a tractserver fails?
  Metadata server stops getting heartbeat RPCs
  Picks random replacement for each TLT entry failed server was in
  New TLT gets a new version number
  Replacement servers fetch copies

Example of the tracts each server holds:
  S1: 0 4 8 ...
  S2: 0 1 ...
  S3: 4 3 ...
  S4: 8 2 ...

Q: why not just pick one replacement server?

Q: how long will it take to copy all the tracts?

Q: if a tractserver's net breaks and is then repaired, might srvr serve old data?

Q: if a server crashes and reboots with disk intact, can contents be used?
   e.g. if it only missed a few writes?
   3.2.1's "partial failure recovery"
   but won't it have already been replaced?
   how to know what writes it missed?

Q: when is it better to use 3.2.1's partial failure recovery?

* What happens when the metadata server crashes?

Q: while metadata server is down, can the system proceed?

Q: is there a backup metadata server?

Q: how does rebooted metadata server get a copy of the TLT?

Q: does their scheme seem correct?
   how does the metadata server know it has heard from all tractservers?
   how does it know all tractservers were up to date?

* Random issues

Q: is the metadata server likely to be a bottleneck?

Q: why do they need the scrubber application mentioned in 2.3?
   why don't they delete the tracts when the blob is deleted?
   can a blob be written after it is deleted?

* Performance

Q: how do we know we're seeing "good" performance?
   what's the best you can expect?

Q: limiting resource for 2 GB / second single-client?

Q: Figure 4a: why starts low? why goes up? why levels off?
   why does it level off at that particular performance?

Q: Figure 4b shows random r/w as fast as sequential (Figure 4a).
   is this what you'd expect?

Q: why are writes slower than reads with replication in Figure 4c?

Q: where does the 92 GB in 6.2 seconds come from?
   Table 1, 4th column
   that's 15 GB / second, both read and written
   1000 disks, triple replicated, 128 servers?
   what's the limiting resource? disk? cpu? net?

How big is each sort bucket?
  i.e. is the sort of each bucket in-memory?
  1400 GB total
  128 compute servers
  between 12 and 96 GB of RAM each
  hmm, say 50 on average, so total RAM may be 6400 GB
  thus sort of each bucket is in memory, does not write passes to FDS
  thus total time is just four transfers of 1400 GB
    client limit: 128 * 2 GB/s = 256 GB / sec
    disk limit: 1000 * 50 MB/s = 50 GB / sec
  thus bottleneck is likely to be disk throughput