Flat Datacenter Storage

Edmund B. Nightingale, Jeremy Elson, et al.

6.S897

Motivation

- Imagine a world with “flat” data storage
    - Simple, Centralized, and easy to program
- Unfortunately, datacenter networks were once **oversubscribed**
    - Shortage of bandwidth ⇒ *“Move Computation to Data”*
        - Programming models like MapReduce, Dryad, etc.
- These locality constraints hindered efficient resource utilization!

Motivation

Data Center Networks are getting faster!

New topologies mean networks can support **Full Bisection Bandwidth**.

Motivation

Idea: Design with full bisection bandwidth in mind.

All compute nodes can access all storage with equal throughput!

Consequence: No need to worry about data locality.

FDS read/write performance exceeds 2 GB/s, can recover 92GB of lost data in 6.2 seconds, and broke a world record in sorting in 2012.

FDS Design

Design: Blobs and Tracts

Data is stored in logical *blobs*

- Byte sequences with a 128-bit Global Unique Identifiers (GUID)
- Divided into constant sized units called *tracts*

Tracts are sized so random and sequential accesses have same throughput

Both tracts and blobs are <u>mutable</u>

Disk is managed by a `tractserver` process

- Read/write to disk directly without filesystem; tract data cached in memory

Design: System API

- Reads and writes not guaranteed to appear in the order they are issued
- Read and writes are atomic
- API is non-blocking
    - Responds to application using a callback

Non-blocking API helps performance: many requests can be issued in parallel, and FDS can pipeline disk reads with network transfers.

Design: Locating a Tract

Tractservers can be found deterministically using a **Tract Locator Table (TLT)**.

TLT is distributed to clients using a centralized *metadata server*.

To read or write tract *i* in blob with GUID *g*:

Tract_Locator = (Hash(***g***) + ***i***) mod **TLT_Length**

Deterministic, and produces uniform disk utilization

Don't hash *i* so a blob uses entries in TLT uniformly.

TLT Example

Row	Version Number	Replica 1	Replica 2	Replica 3
1	234	A	F	B
2	235	B	C	L
3	567	E	D	G
4	13	T	A	H
5	67	F	B	G
6	123	G	E	B
7	86	D	V	C
8	23	H	E	F

Replication

Each TLT entry is **k-way**

Writes go to all k replicas; reads pick a random entry.

Metadata updates are serialized by a primary replica and shared with secondaries using a two-phase commit protocol

Replicated Data Layout

- O(n^2) TLT entries for **two replicas** mean fast recovery for single failure
  - ~*1/n* data on each of the remaining disks, so highly parallel recovery.
  - *Problem: two failures = guaranteed data loss!*
    - Since each pair of disks appears in the TLT, two losses means all disks failed for one TLT entry.
- **Simple solution**: O(n^2) entries (every possible pair), and k-way replication with $k > 2$. $k - 2$ replicas chosen at random.

Design: Metadata

Stored on a special tract for each blob, accessed using the TLT

Blobs are extended using API calls, which access the metadata tract

Appends to metadata are equivalent to “record append” on GFS

Design: Dynamic Work Allocation

Since data and compute no longer need to be co-located, work can be assigned **dynamically** and with **finer granularity**.

With FDS, a cluster can centrally schedule work for each worker as it nears completion of it's previous unit.

Note: Unlike MapReduce, etc., which must take into account where data resides when assigning work!

Significant impact on performance.

Replication and Failure Recovery

Failure Recovery

TLT carries a version number for each row.

On failure:

1. Metadata server **detects failure** after HeartBeat message times out
2. Current **TLT is invalidated** by incrementing version
3. Random tractservers are picked to **fill gaps in TLT** after failure
4. tractservers ACK new assignment, **replicate data**

Clients with stale TLTs request new ones from metadata server

No need to wait for replication to finish; just TLT update.

Failure Recovery

Row	Version	Replica 1	Replica 2	Replica 3
1	8	A	F	B
2	17	B	C	L
3	324	E	D	G
4	3	T	A	H
5	456	F	B	G
6	723	G	E	B
7	235	D	V	C
8	312	H	E	F

Row	Version	Replica 1	Replica 2	Replica 3
1	9	A	F	H
2	18	I	C	L
3	324	E	D	G
4	3	T	A	H
5	457	F	C	G
6	724	G	E	A
7	235	D	V	C
8	312	H	E	F

Figure 2: A Tract Locator Table before (*left*) and after (*right*) Disk B fails. B's appearances in the table are replaced with different disks and the version numbers of affected rows are incremented. All the disks in rows that contained B participate in failure recovery in parallel.

Fault Recovery Guarantees

Weak Consistency

Similar to GFS; trackservers may inconsistent during failure, or if client fails after a write to a subset of replicas

Availability

Clients only need to wait for an updated trackserver list

Partition Tolerance?

One active master at a time to prevent corrupted states, but a partitioned network may mean that clients can’t write to all replicas

Results: Read + Write Performance

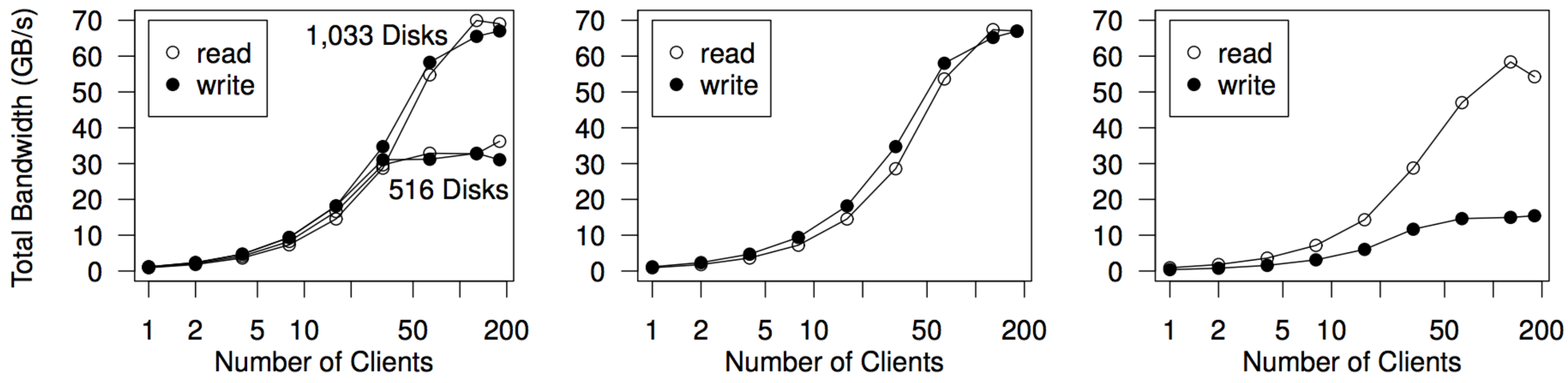


Figure 4: Mean aggregate throughput of 1 to 180 clients reading and writing 8 MB tracts on a 1,033-disk cluster. Standard deviation is less than 1% of the mean of each point. The *x* axes use logarithmic scales. *(a)* Sequential reading and writing in a single-replicated cluster. Results for a 516-disk cluster are also shown. *(b)* Random reading and writing in a single-replicated cluster. *(c)* Sequential reading and writing in a triple-replicated cluster.

Comparison with GFS

Simple metadata server

Only stores TLTs, not information about the tracts themselves. So tracts can be arbitrarily small! (Google says 64MB is too big for their chunksize)

Master in GFS also a potential bottleneck as scale increases?

Single file reads can be issued with very high throughput

Since tracts are stored across many disks, reads can be issued in parallel

Anything Else?

Takeaways

- A storage system that takes advantage of new data center properties

  Namely: **Full Network Bisection Bandwidth**

- Each compute node has equal throughput to each storage node, so don’t need to design for locality
- No need to worry about locality $\Rightarrow$ **simple design and failure recovery**, ability to schedule jobs at fine granularity and without wasting resources

Results show that the system is fast at recovery, and provides efficient I/O

Bonus Slides

Cluster Growth

tractservers can be added at runtime.

1. Increment version number of TLT, begin copying data to new tractserver
    a. “pending” phase
2. When finished, TLT entry version incremented again, “committing” the new disk

While in pending state, new writes are added to the new tractserver as well. Failure of new server ⇒ expunge it and increment version. Failure of existing tractserver ⇒ run recovery protocol.

Network

Uses a **full bisection bandwidth network**, with ECMP to load balance flows (stochastically guarantees bisection bandwidth).

Storage nodes given bandwidth equal to the **disk bandwidth**

Compute nodes given bandwidth equal to the **I/O bandwidth**

- RSS, zero-copy used to saturate 10, 20 Gbps links respectively
- TCP alone not enough!

Experiments and Results

Testbed Setup

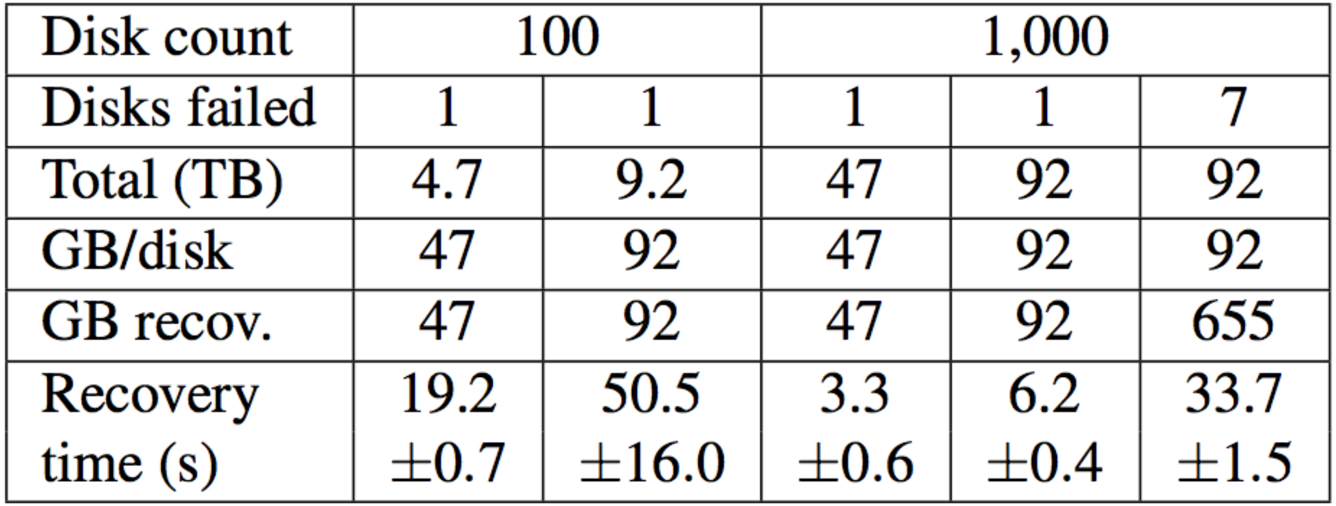
14 Racks, Full bisection bandwidth network with 5.5 Tbps.

- BGP for route selection, IP subnets for each TOR

Operating cost: $250,000

Heterogeneous environment with up to 256 Servers, 2 to 24 cores, 12 to 96 GB RAM, 300GB 10,000RPM SAS Drives, and 500GB, 1TB 7200 RPM SATA Drives

Results: Recovery



Disk count	100		1,000		
Disks failed	1	1	1	1	7
Total (TB)	4.7	9.2	47	92	92
GB/disk	47	92	47	92	92
GB recov.	47	92	47	92	655
Recovery time (s)	19.2 ±0.7	50.5 ±16.0	3.3 ±0.6	6.2 ±0.4	33.7 ±1.5

Table 1: Mean and standard deviation of recovery time after disk failure in a triple-replicated cluster. The high variance in one experiment is due to a single 80 s run.