A Balanced Large-Scale Sorting System

CSE 224 – Jan 20, 2022

Alex Rasmussen, George Porter, Michael Conley, Radhika Niranjan Mysore, Amin Vahdat (UCSD)
Harsha V. Madhyastha (UC Riverside)
Alexander Pucher (Vienna University of Technology)
The Rise of Big Data Workloads

• Very high I/O and storage requirements
  – Large-scale web and social graph mining
  – Business analytics – “you may also like …”
  – Large-scale “data science”
• Recent new approaches to “data deluge”: data intensive scalable computing (DISC) systems
  – MapReduce, Hadoop, Dryad, …
Size of “data-intensive” has grown
Size of “data-intensive” has grown

100MB  1 TB  100 TB

1,000,000x increase
“Data-intensive” commonly means MapReduce

Input: Set<key-value pairs>
1. Apply map() to each pair
2. Group by key; sort each group
3. Apply reduce() to each sorted group

Sorting is the challenge!
Performance via scalability

• 10,000+ node MapReduce clusters deployed
  – With impressive performance
• Example: Yahoo! Hadoop Cluster Sort
  – 3,452 nodes sorting 100TB in less than 3 hours
• But…
  – Less Than 3 MB/sec per node
  – Single disk: ~100 MB/sec
• Not an isolated case
  – See “Efficiency Matters!”, SIGOPS 2010
Overcoming Inefficiency With Brute Force

• Just add more machines!
  – But expensive, power-hungry mega-datacenters!

• What if we could go from 3 MBps per node to 30?
  – 10x fewer machines accomplishing the same task
  – or 10x higher throughput
Sorting!
Evaluating IO efficiency: GraySort

- Sorting contest [Jim Gray et al., 1985]
  - Importance of the IO subsystem
  - 1985: Sort 100MB
  - 1999: Sort 1TB

- 2009: Sort 100 TB

Sorting records

- GraySort:
  - Time to sort 100TB
- MinuteSort:
  - How much in 60s
- JouleSort:
  - How much in 1 Joule
- PennySort:
  - How much for 1¢
- CloudSort:
  - How much $$ to sort 100TB?
Inefficiency of deployed scale-out systems

• Analysis of GraySort contest results*
  – On average: 94% disk IO idle; 33% of CPU idle

• Case study: 2009 Yahoo! Hadoop Cluster
  – Sorted 100TB with 3,452 nodes in ≈3 hours
  – 1% disk efficiency

* Anderson and Tucek, “Efficiency matters!” SIGOPS OSR 44, 1 (March 2010)
3452 nodes at 1% efficiency
35 nodes at 100% efficiency
TritonSort Goals

• Build a highly efficient DISC system that improves per-node efficiency by an order of magnitude vs. existing systems
  – Through balanced hardware and software

• Secondary goals:
  – Completely “off-the-shelf” components
  – Focus on I/O-driven workloads (“Big Data”)
  – Problems that don’t come close to fitting in RAM
  – Initially sorting, but have since generalized
Outline

• Define hardware and software balance
• TritonSort design
  – Highlighting tradeoffs to achieve balance
• Evaluation with sorting as a case study
Building a “Balanced” System

- *Balanced hardware* drives all resources as close to 100% as possible
  - Removing any resource slows us down
  - Limited by commodity configuration choices
- *Balanced software* fully exploits hardware resources
Hardware Selection (in 2010)

• Designed for I/O-heavy workloads
  – Not just sorting

• Static selection of resources:
  – Network/disk balance
    • 10 Gbps / 80 MBps ≈ 16 disks
  – CPU/disk balance
    • 2 disks / core = 8 cores
  – CPU/memory
    • Originally ~1.5GB/core… later 3 GB/core
Resulting Hardware Platform

52 Nodes:

- Xeon E5520, 8 cores (16 with hyperthreading)
- 24 GB RAM
- 16 7200 RPM hard drives
- 10 Gbps NIC
- Cisco Nexus 5020
- 10 Gbps switch
Software Architecture

• Staged, pipeline-oriented dataflow system
• Program expressed as digraph of stages
  – Data stored in buffers that move along edges
  – Stage’s work performed by worker threads
• Platform for experimentation
  – Easily vary:
    • Stage implementation
    • Size and quantity of buffers
    • Worker threads per stage
    • CPU and memory allocation to each stage
Why Sorting?

• Easy to describe
• Industrially applicable
• Uses all cluster resources
Current TritonSort Architecture

• External sort – two reads, two writes*
  – Don’t read and write to disk at same time
    • Partition disks into input and output

• Two phases
  – **Phase one**: route tuples to appropriate on-disk partition (called a “logical disk”) on appropriate node
  – **Phase two**: sort all logical disks in parallel

Architecture Phase One

Input Disks --> Reader --> Node Distributor --> Sender
Architecture Phase One

Receiver → LD Distributor → Coalescer → Writer

Linked list per partition

Output Disks

Disk 1
Disk 2
Disk 3
Disk 4
Disk 5
Disk 6
Disk 7
Disk 8
• 100 MBps/disk * 8 disks = 800 MBps
• No computation, entirely I/O and memory operations
  – Expect most time spent in iowait
  – 8 reader workers, one per input disk
    ✓ All reader workers co-scheduled on a single core
NodeDistributor

• Appends tuples onto a buffer per destination node
• Memory scan + hash per tuple
• 300 MBps per worker
  – Need three workers to keep up with readers
Sender & Receiver

- 800 MBps (from Reader) is 6.4 Gbps
  - All-to-all traffic
- Must keep downstream disks busy
  - Don’t let receive buffer get empty
  - Implies strict socket send time bound
- Multiplex all senders on one core (single-threaded tight loop)
  - Visit every socket every 20 µs
  - Didn’t need epoll()/select()
Balancing at Scale

TCP Throughput (in Gbps)

# Nodes

- **Single-threaded**
- **Multi-threaded**
- **Required Throughput (6.4 Gbps)**
Logical Disk Distributor

\[ H(t_0)H(t_1) = N \]

12.8 KB
Logical Disk Distributor

- Data non-uniform and bursty at short timescales
  - Big buffers + burstiness = head-of-line blocking
  - Need to use *all* your memory *all* the time

- **Solution**: Read incoming data into smallest buffer possible, and form chains
Coalescer & Writer

• Copies tuples from LDBuffer chains into a single, sequential block of memory
• Longer chains = larger write before seeking = faster writes
  – Also, more memory needed for LDBuffers
• Buffer size limits maximum chain length
  – How big should this buffer be?
Appending records to partitions

Buffer of k/v pairs

k1 k2 k3 k4 k5 k6 k7 k8 ...

Writer 1 → P/M
Writer 2 → P/M
Writer M → P/M

M output disks
P/M partitions per disk
Approach #1: Delegate to OS

- Low performance due to insufficient batching
Approach #2: Per-partition buffers

Buffer of k/v pairs

| k1 | k2 | k3 | k4 | k5 | k6 | k7 | k8 | … |

- Partition 1 (20GB / P)
- Partition 2 (20GB / P)
- Partition 3 (20GB / P)
- Partition 4 (20GB / P)
- Partition …

(20GB)

Writer 1 → P/M
Writer 2 → P/M
Writer M → P/M

M output disks
P/M partitions per disk
Approach #2: Per-partition buffers

- Non-uniform arrivals result in “hot” buffers
TritonSort: Load-balancing across partitions

Buffer of k/v pairs

k1 k2 k3 k4 k5 k6 k7 k8 …

PartitionAppender

20GB pool
(≈2M buffers)

Fine-grained allocation of small buffers to partitions

Writer 1 → P/M
Writer 2 → P/M
Writer M → P/M

M output disks
P/M partitions per disk

10KB

TritonSort: Load-balancing across partitions
TritonSort: Load-balancing across partitions

Buffer of k/v pairs
k1 k2 k3 k4 k5 k6 k7 k8 ...

PartitionAppender

P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 P11 P12 P13 ...

One “chain” of buffers per partition

20GB pool

10KB 10KB 10KB 10KB

10KB pool

Writer 1 → P/M
Writer 2 → P/M
Writer M → P/M

M output disks
P/M partitions per disk

TritonSort: Load-balancing across partitions
Handling “hot” partitions

Buffer of k/v pairs

k1 k2 k3 k4 k5 k6 k7 k8 ...

PartitionAppender

P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 P11 P12 P13 ...

Bu
ff
er of k/v pairof k/v pairs

20GB pool

10KB10KB

PartitionAppender

Large
Chain = Largest possible write

Writer 1

Writer 2

Writer M

M output disks

P/M partitions per disk

Largest chain = Largest possible write
Handling slow disks

Slow disks accept writes less often, leading to larger writes.
Architecture Phase Two

Reader -> Sorter -> Writer

Input Disks

Output Disks
Evaluation

100TB

GravSort

2009

100TB

JouleSort

2010

2,200 nodes

0.578 TB/min

(2.8 MB/sec/node)

52 nodes

0.725 TB/min

(232.4 MB/sec/node)

3,452 nodes

1.42 TB/min

(11.3 MB/sec/node)

Created in 2010
Going after CloudSort

• Our team, with lead student Mike Conley, ported Themis to Amazon’s Cloud Infrastructure

• Goal:
  – Learn how to migrate a system designed for dedicated resources to an on-demand service
  – Break the record using cloud computing
The project in a nutshell...
Key challenges

• “The cloud” often doesn’t have any rain
  – We frequently couldn’t get enough nodes 😞
• Performance(N-node cluster) != N * Performance(1 node)
• Network bandwidth is not good
Characterizing each type of node
What a 100TB sort *should* cost

<table>
<thead>
<tr>
<th>Instance type</th>
<th>Num nodes to sort 100 TB</th>
<th>Sort cost ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>c3.large</td>
<td>9,375</td>
<td>28</td>
</tr>
<tr>
<td>m3.large</td>
<td>9,375</td>
<td>65</td>
</tr>
<tr>
<td>m3.medium</td>
<td>75,000</td>
<td>66</td>
</tr>
<tr>
<td>m1.xlarge</td>
<td>179</td>
<td>155</td>
</tr>
<tr>
<td>i2.4xlarge</td>
<td>94</td>
<td>211</td>
</tr>
<tr>
<td>i2.8xlarge</td>
<td>47</td>
<td>218</td>
</tr>
<tr>
<td>hs1.8xlarge</td>
<td>7</td>
<td>248</td>
</tr>
<tr>
<td>cr1.8xlarge</td>
<td>1,250</td>
<td>2,966</td>
</tr>
</tbody>
</table>
But then... the network...
Factoring in the network

Cost ($) vs VM Configuration

- Observed Network Scalability
- Ideal Network Scalability
- Infinitely Fast Network
<table>
<thead>
<tr>
<th>Category</th>
<th>Previous record</th>
<th>UCSD 2014 Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indy GraySort</td>
<td>1.42 TB/min (2,100 nodes)</td>
<td>6.76 TB/min (178 nodes)</td>
</tr>
<tr>
<td>Daytona GraySort</td>
<td>1.42 TB/min (2,100 nodes)</td>
<td>4.35 TB/min (186 nodes)</td>
</tr>
<tr>
<td>Indy MinuteSort</td>
<td>1401 GB (256 nodes)</td>
<td>4094 GB (178 nodes)</td>
</tr>
<tr>
<td>Indy CloudSort</td>
<td>N/A</td>
<td>$449.53 (330 nodes)</td>
</tr>
<tr>
<td>Daytona CloudSort</td>
<td>N/A</td>
<td>$449.53 (330 nodes)</td>
</tr>
</tbody>
</table>
For more information
