Performance Optimization Case Study: Shattering Hadoop's Sort Record with Spark and Scala

Reynold Xin (@rxin)
March 17, 2015
Who am I?

Reynold Xin (@rxin)

Co-founder, Databricks

Apache Spark committer & maintainer on core API, shuffle, network, block manager, SQL, GraphX

On leave from PhD @ UC Berkeley AMPLab
Spark is In-Memory and Fast

K-Means Clustering
- Hadoop MR: 155 seconds
- Spark: 4.1 seconds

Logistic Regression
- Hadoop MR: 110 seconds
- Spark: 0.96 seconds
“Spark is in-memory. It doesn’t work with BIG DATA.”

“It is too expensive to buy a cluster with enough memory to fit our data.”
Common Misconception About Spark

“Spark is in-memory. It doesn’t work with BIG DATA.”

“It is too expensive to buy a cluster with enough memory to fit our data.”
Spark Project Goals

Works well with GBs, TBs, or PBs or data

Works well with different storage media (RAM, HDDs, SSDs)

Works well from low-latency streaming jobs to long batch jobs
Spark Project Goals

Works well with KBs, MBs, … or PBs or data

Works well with different storage media (RAM, HDDs, SSDs)

Works well from low-latency streaming jobs to long batch jobs
Sort Benchmark

Originally sponsored by Jim Gray to measure advancements in software and hardware in 1987

Participants often used purpose-built hardware/software to compete

- Large companies: IBM, Microsoft, Yahoo, …
- Academia: UC Berkeley, UCSD, MIT, …
Sort Benchmark

DIAPRISM Hardware Sorter
— Sort a Million Records Within a Second —

Shinsuke Azuma, Takao Sakuma, Tetsuya Takeo, Takaaki Ando, Kenji Shirai
Mitsubishi Electric Corp.

Mitsubishi Electric Corp. has released the DIAPRISM hardware sorter for OLAP (On-Line Analytical Processing) and data warehouse applications. It enables high-speed sorting on a commercial PC server. The sorter employs a pipeline merge sort algorithm. It consists of multiple sort processors connected linearly and necessary amount of memory for each processor. The architecture of the sort processor is capable of sorting 4GB of data at one time. The sorter has achieved Datamation benchmark record, sorting a million 100-byte records in 0.998 second.

1MB -> 100MB -> 1TB (1998) -> 100TB (2009)
Sorting 100 TB and 1 PB

Participated in the most challenging category
  – Daytona GraySort
  – Sort 100TB in a fault-tolerant manner
  – 1 trillion records generated by a 3rd-party harness (10B key + 90B payload)

Set a new world record (tied with UCSD)
  – Saturated throughput of 8 SSDs and 10Gbps network
  – 1st time public cloud + open source won

Sorted 1PB on our own to push scalability
On-Disk Sort Record:
Time to sort 100TB

2013 Record: Hadoop
2100 machines
72 minutes

2014 Record: Spark
207 machines
23 minutes

Also sorted 1PB in 4 hours

Source: Daytona GraySort benchmark, sortbenchmark.org
Why Sorting?

Sorting stresses “shuffle”, which underpins everything from SQL to machine learning (group by, join, sort, ALS, etc).

Sorting is challenging because there is no reduction in data.

Sort 100TB = 500TB disk I/O + 200TB network
What made this possible?

Power of the Cloud

Engineering Investment in Spark
  – Sort-based shuffle (SPARK-2045)
  – Netty native network transport (SPARK-2468)
  – External shuffle service (SPARK-3796)

Clever Application Level Techniques
  – Avoid Scala/JVM gotchas
  – Cache-friendly & GC-friendly sorting
  – Pipelining
What made this possible?

Power of the Cloud

Engineering Investment in Spark
  – Sort-based shuffle (SPARK-2045)
  – Netty native network transport (SPARK-2468)
  – External shuffle service (SPARK-3796)

Clever Application Level Techniques
  – Avoid Scala/JVM gotchas
  – Cache-friendly & GC-friendly sorting
  – Pipelining
Sort-based Shuffle

Old hash-based shuffle: require $R$ (#reduce tasks) concurrent streams with buffers, i.e. limits $R$.

New sort-based shuffle: sort records by partition first, and then write them out. One active stream at a time.

Went up to 250,000 tasks in PB sort
Network Transport

High performance event-driven architecture using Netty (SEDA-like)

Explicitly manages pools of memory outside JVM garbage collection

Zero-copy (avoid user space buffer through FileChannel.transferTo)
Network Transport

Sustaining ~1.1GB/node/s
What made this possible?

Power of the Cloud

Engineering Investment in Spark
- Sort-based shuffle (SPARK-2045)
- Netty native network transport (SPARK-2468)
- External shuffle service (SPARK-3796)

Clever Application Level Techniques
- Avoid Scala/JVM gotchas
- Cache-friendly & GC-friendly sorting
- Pipelining
Warning

The guidelines here apply only to hyper performance optimizations

- Majority of the code should not be perf sensitive

Measure before you optimize

- It is ridiculously hard to write good benchmarks
- Use jmh if you want to do that

http://github.com/databricks/style-style-guide
Prefer while loops

array.zipWithIndex.foreach { case (x, i) =>
  ...
}

vs

var i = 0
while (i < array.length)
{
  ...
  i += 1
}
Avoid Collection Library

Avoid Scala collection library
  – Generally slower than java.util.*

Avoid Java collection library
  – Not specialized (too many objects)
  – Bad cache locality
Prefer private[this]

A.scala:

class A {
    private val field = 0
    def accessor = field
}

scalac –print A.scala

Class A extends Object {
    private[this] val field: Int = _;
    <stable> <accessor> private def field(): Int = A.this.field;
    def accessor(): Int = A.this.field();
    def <init>(): A = {
        A.super.<init>();
        A.this.field = 0;
    }
}

JIT might not be able to inline this
Prefer private[this]

B.scala:
  class B {
    private[this] val field = 0
    def accessor = field
  }

scalac -print B.scala

Class B extends Object {
  private[this] val field: Int = _;
  def accessor(): Int = B.this.field;
  def <init>(): B = {
    B.super.<init>();
    B.this.field = 0;
    ()
  }
};
What made this possible?

Power of the Cloud

Engineering Investment in Spark
- Sort-based shuffle (SPARK-2045)
- Netty native network transport (SPARK-2468)
- External shuffle service (SPARK-3796)

Clever Application Level Techniques
- Avoid Scala/JVM gotchas
- Cache-friendly & GC-friendly sorting
- Pipelining
Garbage Collection

Hard to tune GC when memory utilization > 60%

Do away entirely with GC: off-heap allocation
GC Take 1: DirectByteBuffer

java.nio.ByteBuffer.allocateDirect(long size);

Limited size: 2GB only

Hard to recycle
  – only claimed by GC
  – possible to control with reflection, but there is a static synchronize!
GC Take 2: sun.misc.Unsafe

Available on most JVM implementations (frequently exploited by trading platforms)

Provide C-like explicit memory allocation

```java
class Unsafe {

    public native long allocateMemory(long bytes);
    public native void freeMemory(long address);

    ...
}
```
class Unsafe {
  ...

  public native copyMemory(
      long srcAddress, long destAddress,
      long bytes);

  public native long getLong(long address);

  public native int getInt(long address);

  ...
}
Caveats when using Unsafe

Maintain a thread local pool of memory blocks to avoid allocation overhead and fragmentation (Netty uses jemalloc)

-ea during debugging, -da in benchmark mode
Cache Friendly Sorting

Naïve Scala/Java collection sort is extremely slow
  – object dereferences (random memory access)
  – polymorphic comparators (cannot be inlined)

Spark’s TimSort allows operating on custom data memory layout
  – Be careful with inlining again
Data Layout Take 1

Record format: 10B key + 90B payload

Consecutive Memory Block (size N * 100B)

Comparison: uses the 10B key to compare.

Swap: requires swapping 100B.

Too much memcpy!
Data Layout Take 2

Array[Array[Byte]]

Swap: cheap (8B or 4B)

Comparison: requires dereference 2 pointers (random access)

Bad cache locality!
Data Layout Take 3

Consecutive Memory Block (size N * 14B)

Sort comparison: using the 10B prefix inline

Swap: only swap 14B

Good cache locality & low memcpy overhead!
AlphaSort: A Cache-Sensitive Parallel External Sort

Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet

Received September 8, 1994; revised version received, March 28, 1995; accepted March 28, 1995.

Abstract. A new sort algorithm, called AlphaSort, demonstrates that commodity processors and disks can handle commercial batch workloads. Using commodity processors, memory, and arrays of SCSI disks, AlphaSort runs the industry-standard sort benchmark in seven seconds. This beats the best published record on a 32-CPU 32-disk Hypercube by 8:1. On another benchmark, AlphaSort sorted more than a gigabyte in one minute. AlphaSort is a cache-sensitive, memory-intensive sort algorithm. We argue that modern architectures require algorithm designers to re-examine their use of the memory hierarchy. AlphaSort uses clustered data structures to get good cache locality, file striping to get high disk bandwidth, QuickSort to generate runs, and replacement-selection to merge the runs. It uses shared memory multiprocessors to break the sort into subsort chores. Because startup times are becoming a significant part of the total time, we propose two new benchmarks: (1) MinuteSort: how much can you sort in one minute, and (2) PennySort: how much can you sort for one penny.
What made this possible?

Power of the Cloud

Engineering Investment in Spark
  – Sort-based shuffle (SPARK-2045)
  – Netty native network transport (SPARK-2468)
  – External shuffle service (SPARK-3796)

Clever Application Level Techniques
  – Avoid Scala/JVM gotchas
  – Cache-friendly & GC-friendly sorting
  – Pipelining
Pipelining

time

core 1
- task 1 read
- task 1 sort
- task 1 write

core 2
- task 2 read
- task 2 sort
- task 2 write

disk I/O bound
- CPU bound
- disk I/O bound
Add a semaphore to throttle concurrent “reads”
What did we do exactly?

Improve single-thread sorting speed
  – Better sort algorithm
  – Better data layout for cache locality

Reduce garbage collection overhead
  – Explicitly manage memory using Unsafe

Reduce resource contention via pipelining

Increase network transfer throughput
  – Netty (zero-copy, jemalloc)
Startup Crunches 100 Terabytes of Data in a Record 23 Minutes

BY KLINT FINLEY 10.13.14 | 2:36 PM | PERMALINK

GigaOM EVENTS RESEARCH

Google launches Contributor, a crowdfunding tool for publishers
Net neutrality looks doomed in Europe before it even gets started
Five tech products that designers have fallen in love with

Databricks demolishes big data benchmark to prove Spark is fast on disk, too

by Derrick Harris Oct. 10, 2014 - 1:49 PM PST

1 Comment
Extra Readings

Databricks Scala Guide (Performance Section)
https://github.com/databricks/scala-style-guide

Blog post about this benchmark entry
http://tinyurl.com/spark-sort
Thanks!
<table>
<thead>
<tr>
<th></th>
<th>Hadoop MR Record</th>
<th>Spark Record</th>
<th>Spark 1 PB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Size</td>
<td>102.5 TB</td>
<td>100 TB</td>
<td>1000 TB</td>
</tr>
<tr>
<td>Elapsed Time</td>
<td>72 mins</td>
<td>23 mins</td>
<td>234 mins</td>
</tr>
<tr>
<td># Nodes</td>
<td>2100</td>
<td>206</td>
<td>190</td>
</tr>
<tr>
<td># Cores</td>
<td>50400 physical</td>
<td>6592 virtualized</td>
<td>6080 virtualized</td>
</tr>
<tr>
<td>Cluster disk throughput</td>
<td>3150 GB/s (est.)</td>
<td>618 GB/s</td>
<td>570 GB/s</td>
</tr>
<tr>
<td>Sort Benchmark Daytona Rules</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Network</td>
<td>dedicated data center, 10Gbps</td>
<td>virtualized (EC2) 10Gbps network</td>
<td>virtualized (EC2) 10Gbps network</td>
</tr>
<tr>
<td>Sort rate</td>
<td>1.42 TB/min</td>
<td>4.27 TB/min</td>
<td>4.27 TB/min</td>
</tr>
<tr>
<td>Sort rate/node</td>
<td>0.67 GB/min</td>
<td>20.7 GB/min</td>
<td>22.5 GB/min</td>
</tr>
</tbody>
</table>