Memory Management for Spark

Ken Salem
Cheriton School of Computer Science
University of Waterloo
Where I’m From
What We’re Doing

Flexible Transactional Persistence

DBMS-Managed Energy Efficiency

Non-Relational Database Design

Non-Relational Database Design
Talk Plan

● Part 1: Spark overview
  ○ What does Spark do?
  ○ Spark system architecture
  ○ Spark programs
  ○ Program execution: sessions, jobs, stages, tasks

● Part 2: Memory and Spark
  ○ How does Spark use memory?

● Part 3: Memory-Oriented Research
  ○ External caches
  ○ Cache sharing
  ○ Cache management
Part 1: Spark Overview
What Is Spark?

- Popular Apache platform for scalable data processing
- Analytical programs in Scala, Java, Python, embedding Spark parallel operators
- Higher level frameworks for ETL (SparkSQL), graph processing (GraphX), streaming (Spark Streaming), machine learning (MLLib)
- Data in HDFS, NoSQL stores, Hive, others.
A Spark Cluster

Driver

One per application. Runs sequential part, farms parallel part out to Executors.

Local secondary storage available for temporary/intermediate results

Application input/output from cluster-wide storage system (e.g., HDFS, Cassandra, Hive)

Cluster Storage
Each application has its own private driver and executors.
Spark Executors

- Executor = JVM
- Workers in an executor share access to a cache.
- Application specifies
  - (memory) size of executors
  - number of workers per executor
  - number of executors
k-Means Clustering
A Spark Program

```python
lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0])
data = lines.map(parseVector).cache()
kPoints = data.takeSample(False, K, 1)
tempDist = 1.0
while tempDist > convergeDist:
    closest = data.map(lambda p: (closestPoint(p, kPoints), (p, 1)))
    pointStats = closest.reduceByKey(lambda p1_c1[0], p2_c2: (p1_c1[0] + p2_c2[0], p1_c1[1] + p2_c2[1]))
    newPoints = pointStats.map(lambda st: (st[0], st[1][0] / st[1][1])).collect()
    tempDist = sum(np.sum((kPoints[iK] - p) ** 2) for (iK, p) in newPoints)
    for (iK, p) in newPoints:
        kPoints[iK] = p

print("Final centers: " + str(kPoints))
```

Define an **RDD** of data points, from a file. Each element of the RDD is a data point.
RDDs

- RDD = Resilient Distributed Dataset
  - Set of elements on which parallel operations can be performed

- RDDs can be defined from:
  - Collections defined in the application program
  - External data sources (e.g., files)
  - Other RDDs

- Spark defines operations that can be performed on RDDs.
  - RDD operations are how Spark apps expose parallelism
lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0])
data = lines.map(parseVector).cache()
kPoints = data.takeSample(False, K, 1)
tempDist = 1.0
while tempDist > convergeDist:
closest = data.map(lambda p: (closestPoint(p, kPoints), (p, 1)))
pointStats = closest.reduceByKey(lambda p1_c1, p2_c2: (p1_c1[0] + p2_c2[0], p1_c1[1] + p2_c2[1]))
newPoints = pointStats.map(lambda st: (st[0], st[1][0] / st[1][1])).collect()

for (iK, p) in newPoints:
kPoints[iK] = p

print("Final centers: " + str(kPoints))

- **takeSample** is an example of a Spark **action**
- Actions generate normal program values (in this case, a Python array) from RDDs
- **kPoints** is the initial set of cluster centers
Spark Program Graph

Input file

lines

data

RDD

takeSample
A Spark Program

```python
lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0])
data = lines.map(parseVector).cache()
kPoints = data.takeSample(False, K, 1)
tempDist = 1.0
while tempDist > convergeDist:
    closest = data.map(lambda p: (closestPoint(p, kPoints), (p, 1)))
    pointStats = closest.reduceByKey(lambda p1_c1, p2_c2: (p1_c1[0] + p2_c2[0], p1_c1[1] + p2_c2[1]))
    newPoints = pointStats.map(lambda st: (st[0], st[1][0] / st[1][1])).collect()
    tempDist = sum(np.sum((kPoints[iK] - p) ** 2) for (iK, p) in newPoints)
    for (iK, p) in newPoints:
        kPoints[iK] = p
print("Final centers: " + str(kPoints))
```

**map and reduceByKey are Spark transformations** - they define a new RDD from an existing RDD. Transformations are evaluated lazily.

- Elements of closest: (index, (point,1))
- Elements of pointStats: (index, (pointsum, count))
- Elements of newPoints: (index, point)
Spark Program Graph

takeSample

data

closest

pointStats

newPoints

collect
A Spark Program

lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0])

data = lines.map(parseVector).cache()

kPoints = data.takeSample(False, K, 1)

tempDist = 1.0

while tempDist > convergeDist:
    closest = data.map(lambda p: (closestPoint(p, kPoints), (p, 1)))

    pointStats = closest.reduceByKey(lambda p1_c1, p2_c2:
        (p1_c1[0] + p2_c2[0], p1_c1[1] + p2_c2[1]))

    newPoints = pointStats.map(lambda st: (st[0], st[1][0] / st[1][1])).collect()

    tempDist = sum(np.sum((kPoints[iK] - p) ** 2) for (iK, p) in newPoints)

    for (iK, p) in newPoints:
        kPoints[iK] = p

print("Final centers: " + str(kPoints))
Spark Program Graph

- `takeSample`
- `data`
- `closest`
- `pointStats`
- `newPoints`
- `collect`
- `closest`
- `pointStats`
- `newPoints`
- `collect`
- `closest`
- `pointStats`
- `newPoints`
- `collect`
Another Example Program

tc = spark.sparkContext.parallelize(generateGraph(), partitions).cache()
edges = tc.map(lambda x_y: (x_y[1], x_y[0]))
oldCount = 0
nextCount = tc.count()
while True:
    oldCount = nextCount
    # Perform the join, obtaining an RDD of (y, (z, x)) pairs, then project the result to obtain the new (x, z) paths.
    new_edges = tc.join(edges).map(lambda __a_b: (__a_b[1][1], __a_b[1][0]))
    tc = tc.union(new_edges).distinct().cache()
    nextCount = tc.count()
    if nextCount == oldCount:
        Break
print("TC has %i edges" % tc.count())
Spark Jobs

One Spark job per action.
Spark Jobs

```python
lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0])
data = lines.map(parseVector).cache()
kPoints = data.takeSample(False, K, 1)
tempDist = 1.0
while tempDist > convergeDist:
    closest = data.map(lambda p: (closestPoint(p, kPoints), (p, 1)))
    pointStats = closest.reduceByKey(lambda p1_c1, p2_c2:
        (p1_c1[0] + p2_c2[0], p1_c1[1] + p2_c2[1]))
    newPoints = pointStats.map(lambda st: (st[0], st[1][0] / st[1][1])).collect()
    tempDist = sum(np.sum((kPoints[iK] - p) ** 2) for (iK, p) in newPoints)
    for (iK, p) in newPoints:
        kPoints[iK] = p

print("Final centers: " + str(kPoints))
```

Jobs 2...N
Spark Jobs

- One job per Spark action.
- Spark transformations are evaluated *lazily*.
- Jobs are created sequentially, as actions are encountered during program execution.
- Spark driver does not know in advance how many jobs a program will produce.
RDD Partitioning

File

<table>
<thead>
<tr>
<th>data</th>
<th>closest</th>
<th>pointStats</th>
<th>newPoints</th>
</tr>
</thead>
<tbody>
<tr>
<td>[2,3]</td>
<td>(1, ([2,3], 1))</td>
<td>(1, ([2,3], 1))</td>
<td>(1, ([2,3], 1))</td>
</tr>
<tr>
<td>[1,3]</td>
<td>(1, ([1,3], 1))</td>
<td>(2, ([7,6], 1))</td>
<td>(1, ([1,3], 1))</td>
</tr>
<tr>
<td>[7,6]</td>
<td>(2, ([7,6], 1))</td>
<td>(1, ([3,3], 1))</td>
<td>(1, ([3,3], 1))</td>
</tr>
<tr>
<td>[3,3]</td>
<td>(1, ([3,3], 1))</td>
<td>(1, ([3,3], 1))</td>
<td>(1, ([3,3], 1))</td>
</tr>
<tr>
<td>[2,5]</td>
<td>(2, ([2,5], 1))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[4,2]</td>
<td>(2, ([4,2], 1))</td>
<td>(1, ([1,1], 1))</td>
<td>(1, ([1,1], 1))</td>
</tr>
<tr>
<td>[3,2]</td>
<td>(2, ([3,2], 1))</td>
<td>(2, ([7,2], 1))</td>
<td>(2, ([7,2], 1))</td>
</tr>
<tr>
<td>[1,1]</td>
<td>(1, ([1,1], 1))</td>
<td>(2, ([8,2], 1))</td>
<td>(2, ([8,2], 1))</td>
</tr>
<tr>
<td>[7,2]</td>
<td>(2, ([7,2], 1))</td>
<td>(2, ([5,1], 1))</td>
<td>(2, ([5,1], 1))</td>
</tr>
<tr>
<td>[8,2]</td>
<td>(2, ([8,2], 1))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[5,1]</td>
<td>(2, ([5,1], 1))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[6,3]</td>
<td>(2, ([6,3], 1))</td>
<td>(2, ([7,6], 1))</td>
<td>(2, ([46,30], 9))</td>
</tr>
<tr>
<td>[1,2]</td>
<td>(1, ([1,2], 1))</td>
<td>(2, ([2,5], 1))</td>
<td>(2, ([5,11,3,33])</td>
</tr>
<tr>
<td>[2,3]</td>
<td>(1, ([2,3], 1))</td>
<td>(2, ([4,2], 1))</td>
<td></td>
</tr>
<tr>
<td>[4,7]</td>
<td>(1, ([2,3], 1))</td>
<td>(2, ([3,2], 1))</td>
<td></td>
</tr>
<tr>
<td>[6,1]</td>
<td>(1, ([6,1], 1))</td>
<td>(2, ([7,2], 1))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(2, ([4,7], 1))</td>
<td>(2, ([8,2], 1))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(1, ([6,1], 1))</td>
<td>(2, ([5,1], 1))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(2, ([6,3], 1))</td>
<td>(2, ([6,3], 1))</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Lines

data

closest

shuffle

pointStats

newPoints

collect
Stages and Tasks

Job 1
- [2,3]
- [1,3]

Job 2, Stage 1
- (1,([2,3],1))
- (1,([1,3],1))
- (1,([3,3],1))
- (2,([2,5],1))

Job 2, Stage 2
- (1,([2,3],1))
- (1,([1,3],1))
- (1,([3,3],1))
- (1,([1,1],1))
- (2,([7,2],1))
- (2,([8,2],1))
- (2,([5,1],1))
- (2,([6,3],1))
- (2,([4,7],1))

Task:
- collect
- newPoints
- pointStats
- closest
- data
- lines
lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0])
data = lines.map(parseVector).cache()
kPoints = data.takeSample(False, K, 1)
tempDist = 1.0
while tempDist > convergeDist:
    closest = data.map(lambda p: (closestPoint(p, kPoints), (p, 1)))
    pointStats = closest.reduceByKey(lambda p1_c1, p2_c2: (p1_c1[0] + p2_c2[0], p1_c1[1] + p2_c2[1]))
    newPoints = pointStats.map(lambda st: (st[0], st[1][0] / st[1][1])).collect()
    tempDist = sum(np.sum((kPoints[iK] - p) ** 2) for (iK, p) in newPoints)
    for (iK, p) in newPoints:
        kPoints[iK] = p
print("Final centers: " + str(kPoints))

1. Local execution at Driver until `takeSample` triggers Job 1.
2. Scheduler determines Job 1 stages and tasks, and assigns tasks to Executors.
Spark Scheduler

Driver

server

Executor

W W W

Cluster Storage

Job 1 Tasks
lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0])
data = lines.map(parseVector).cache()
kPoints = data.takeSample(False, K, 1)

```python
tempDist = 1.0
while tempDist > convergeDist:
    closest = data.map(lambda p: (closestPoint(p, kPoints), (p, 1)))
    pointStats = closest.reduceByKey(lambda p1_c1, p2_c2:
        (p1_c1[0] + p2_c2[0], p1_c1[1] + p2_c2[1]))
    newPoints = pointStats.map(lambda st: (st[0], st[1][0] / st[1][1])).collect()
    tempDist = sum(np.sum((kPoints[iK] - p) ** 2) for (iK, p) in newPoints)
    for (iK, p) in newPoints:
        kPoints[iK] = p
```

print("Final centers: " + str(kPoints))

1. Local execution at Driver until `collect` triggers Job 2.
2. Scheduler determines Job 2 stages and tasks, and assigns tasks to Executors.
Spark Scheduler

Driver

Job 2 Tasks

Cluster Storage

Driver

Executor

W W W

server

Executor

W W W

server

Executor

W W W

server
Part 2: Spark and Memory
How Does Spark Use Memory?

- Memory for task execution
  - Shuffles, joins, sorts, aggregations,....
- Memory for caching
  - Saved RDD partitions
- Indirect Memory use, e.g., file caching
Spark Executor Memory

- Boundary can adjust dynamically
- Execution can evict stored RDDs
Executor Out-of-Memory Failures

Performance Depends on Memory

Shortest Paths for 500K Points

failure @ 512MB
Caching in Spark

Job 1 computes data RDD, takes sample.

Job 2 must re-compute the data RDD, unless it has been cached!

Job 3 also needs the data RDD.
What to Cache?

Spark can cache some or all of the partitions of any RDD.

Q: How does Spark know *which* RDDs to cache?

A: The programmer must tell it!
lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0])
data = lines.map(parseVector).cache()
kPoints = data.takeSample(False, K, 1)
tempDist = 1.0
while tempDist > convergeDist:
    closest = data.map(lambda p: (closestPoint(p, kPoints), (p, 1)))
    pointStats = closest.reduceByKey(lambda p1_c1, p2_c2:
                                        (p1_c1[0] + p2_c2[0], p1_c1[1] + p2_c2[1]))
    newPoints = pointStats.map(lambda st: (st[0], st[1][0] / st[1][1])).collect()
    tempDist = sum(np.sum((kPoints[iK] - p) ** 2) for (iK, p) in newPoints)
    for (iK, p) in newPoints:
        kPoints[iK] = p

print("Final centers: " + str(kPoints))

Dear Spark:

Please cache this RDD.

Thanks,
T. Programmer
Spark programmers use `persist(storage_level)` to identify RDDs that should be cached.

- `cache()` means `persist` at the default storage level.

- **storage_level?**
  - MEMORY_ONLY
  - MEMORY_ONLY_SER
  - DISK_ONLY
  - DISK_AND_MEMORY
  - and more...

- Also: `unpersist()`
Effect of Caching
Spark Caching Advice

Don’t cache all RDDs.

Do cache RDDs that are re-used.

Unless there are too many of them.

In which case, just cache some of them

Or maybe try a DISK or SER level, depending on I/O capacity, recomputation cost, and serialization effectiveness.
Part III: Research

Topics

Exploit Re-Use of Inputs
Cache Sharing Across Programs
Better Cache Management
Research: Quartet

● Input skew: 80% accesses to 10% of files
● Reuse happens quickly: 90% of re-use within one hour
● Key idea: run tasks with cached file inputs first
  ○ Tasks in each stage are independent
  ○ Maximize “memory local” task input
● Exploits Duet: in-kernel framework for exposing contents of file cache to applications

In-Kernel File Cache
Quartet Example

```
lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0])
data = lines.map(parseVector).cache()
kPoints = data.takeSample(False, K, 1)
tempDist = 1.0
while tempDist > convergeDist:
    closest = data.map(lambda p: (closestPoint(p, kPoints), (p, 1)))
    pointStats = closest.reduceByKey(lambda p1_c1, p2_c2: (p1_c1[0] + p2_c2[0], p1_c1[1] + p2_c2[1]))
    newPoints = pointStats.map(lambda st: (st[0], st[1][0] / st[1][1])).collect()
    tempDist = sum(np.sum((kPoints[iK] - p) ** 2) for (iK, p) in newPoints)
    for (iK, p) in newPoints:
        kPoints[iK] = p
print("Final centers: " + str(kPoints))
```

This job runs faster if the input is already (partially) in the file cache.

This job does not benefit.
Research: Cache Sharing

- Quartet can exploit data sharing across programs, since file system cache is external to Spark executors.

- Other ways to achieve this:
  - External caches, e.g., Tachyon/Alluxio
  - Concurrent Spark applications with shared Spark contexts
External Caches

- Distributed, shared in-memory file system
- Caching, or tiered with underlying persistent file system
Sharing Spark Contexts

- Concurrent Spark app
- Each thread can run Spark jobs
- All jobs share the same Executors (and executor caches)
Research: Robus

- **Goal:** Manage a *shared cache* to improve *performance* of a multi-tenant workload, while maintaining *fairness*.
- **Approach:** (1) Batch incoming jobs, (2) make caching decision for batch, (3) load cache, (4) run batch, (5) repeat.
- Enables sharing within batch, not across batches.

Robus

job queue, with weight (relative importance)
Research: What to Cache?

- Spark applications identify RDDs to be cached
- Executors attempt to cache what the application identifies.
- LRU eviction
- Cache granularity = RDD partition
Application-Managed Caching: Best Practice

What to cache?
- RDDs that will be re-used

How to cache it?
- MEMORY_ONLY if everything fits
- MEMORY_ONLY_SER if it helps things fit
- Avoid disk unless computation is expensive or selective

Difficult Advice to Follow!

Can we automate this?
Spark Caching Example

Application says to cache C, E, and J

Is this the best choice? Can Spark make it automatically?
Caching Issue #1: Partial Info

Amount of re-use of C,E is unknown after Job 1

Unpersist after Job 2?
Caching Issue #2: Job Structure

Cost of recalculating E depends on whether its ancestors are cached.
Caching Issue #3: Unhinted Candidates

Is D smaller than E?
Is D -> E expensive?
A Larger Example

9 jobs
16 stages
55 RDDs
Research: Neutrino

- **Goal**: Eliminate programmer-specified storage levels.

- **Approach**:
  - Pilot run(s) to learn complete program graph
  - Schedule cache/convert/discard on RDD partitions at each stage to achieve min-cost execution.

Our Take on the Caching Problem

**Goal:** Application-managed caching ⇒ Application-assisted caching

**Approach:**
Application identifies re-use
- avoid pilot runs, program analysis
Spark maximizes effectiveness of available space
- Key challenge: size and cost estimation!
- Approach: on-the-fly estimation, conditional plans.
Summary

- Spark applications expose parallelism using RDD operations
- Spark provides distributed, fault-tolerant, in-memory execution
- Memory is a critical resource for Spark
  - For program execution, for result caching
- Related Research
  - Exploiting re-use of input data and query results across programs
  - What, where, and how to cache results
Thanks!

ken.salem@uwaterloo.ca