PART 1. LARGE SCALE DATA ANALYTICS
IN-MEMORY CLUSTER COMPUTING
Sangmi Lee Pallickara
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs435

FAQs

• Term project: Proposal
  • 5:00PM October 31, 2019

• Additional readings
Today’s topics

• In-Memory cluster computing
  • Apache Spark

Large Scale Data Analytics
In-Memory Cluster Computing: Apache Spark
Large Scale Data Analytics

In-Memory Cluster Computing: Apache Spark

Introduction

This material is built based on


- Spark Overview, https://spark.apache.org/docs/2.3.0/
- Spark programming guide
  - Job Scheduling
    - https://spark.apache.org/docs/2.0.0-preview/job-scheduling.html
Distributed processing with the Spark framework

Inefficiencies for emerging applications:
(1) Data reuse
- Data reuse is common in many iterative machine learning and graph algorithms
  - PageRank, K-means clustering, and logistic regression
Inefficiencies for emerging applications:
(2) Interactive data analytics

- User runs multiple ad-hoc queries on the same subset of the data

Existing/Previous approaches

- Hadoop
  - Writing output to an external stable storage system
    - e.g. HDFS
    - Substantial overheads due to data replication, disk I/O, and serialization

- Pregel
  - Iterative graph computations

- HaLoop
  - Iterative MapReduce interface

- Pregel/HaLoop support specific computation patterns
  - e.g. looping a series of MapReduce steps
Large Scale Data Analytics

In-Memory Cluster Computing: Apache Spark

RDD (Resilient Distributed Dataset)

- **Read-only, memory resident partitioned collection of records**
  - A fault-tolerant collection of elements that can be operated on in parallel

- RDDs are the core unit of data in Spark
  - Most Spark programming involves performing operations on RDDs
Word Count Example

we use a few transformations to build a dataset of (String, Int) pairs called counts and then save it to a file

```java
JavaRDD<String> textFile = sc.textFile("hdfs://...");
JavaPairRDD<String, Integer> counts
    = textFile.flatMap(s -> Arrays.asList(s.split(" ")).iterator())
    .mapToPair(word -> new Tuple2<>(word, 1))
    .reduceByKey((a, b) -> a + b);
counts.saveAsTextFile("hdfs://...");
```

Overview of RDD

- **Lineage**
  - How it was derived from other dataset to compute its partitions from data in stable storage?
    - RDDs do not need to be materialized at all times

- **Persistence**
  - Users can indicate which RDDs they will reuse and the storage strategy

- **Partitioning**
  - Users can specify the partitioning method across machines based on a key in each record
Spark Programming Interface to RDD: Transformation

• “transformations”
  
  • Operations that create RDDs  
    • Return pointers to new RDDs  
    • e.g. map, filter, and join  
  
  • RDDs can only be created through deterministic operations on either  
    • Data in stable storage  
    • Other RDDs

Spark Programming Interface to RDD: Action

• “actions”
  
  • Operations that return a value to the application or export data to a storage system  
    • e.g. count: returns the number of elements in the dataset  
    • e.g. collect: returns the elements themselves  
    • e.g. save: outputs the dataset to a storage system
Spark Programming Interface to RDD: 
**Persist**

• “persist”
  • Indicates which RDDs they want to **reuse in future operations**
  
  • Spark keeps persistent RDDs in memory by default

  • If there is not enough RAM
    • It can spill them to disk

  • Users are allowed to,
    • store the RDD only on disk
    • replicate the RDD across machines
    • specify a persistence priority on each RDD

**Example: Console Log Mining**

• Suppose that a web service is experiencing errors and an operator wants to search terabytes of logs in the Hadoop file system (HDFS) to find the cause

• The user load the **error messages** from the logs into the RAM across a set of nodes and query them interactively

```python
lines = spark.textFile("hdfs://...")
errors=lines.filter(_.startsWith("ERROR"))
errors.persist()
```

No work has been performed
User can use the RDD in actions
Example: Console Log Mining [2/3]

- Users can perform further transformations and actions on the RDD

```scala
//To count number of error messages
errors.count()

//Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")).count()

//Return the time fields of errors mentioning HDFS as an array (assuming time is field number 3 in a tab-separated format)
errors.filter(_.contains("HDFS"))
  .map(_.split('/t')(3))
  .collect()
```

Lazy Evaluation

- Transformations on RDDs are **lazily evaluated**
  - Spark will **NOT** begin to execute until it sees an action
  - Spark internally records metadata to indicate that this operation has been requested

- **Loading data from files into an RDD** is lazily evaluated

- Reduces the number of passes it has to take over our data by grouping operations together
Example: Console Log Mining

Spark code:
```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
errors.filter(_.contains("HDFS"))
  .map(_.split('/t')(3))
  .collect()
```

Benefits of RDDs as a distributed memory abstraction [1/7]

- RDD vs. Distributed Shared Memory (DSM)?
  - How does RDD work differently compared to DSM?
  - Write/Consistency/Fault-Recovery mechanism/Straggler mitigation

- RDDs can only be created ("written") through **coarse-grained transformations**
  - Coarse-grained transformations are applied over an entire dataset
  - Reads on RDDs can still be fine-grained
    - A large read-only lookup table
    - Applications perform bulk writes

- **More efficient fault tolerance**
  - Lineage based bulk recovery
Benefits of RDDs as a distributed memory abstraction [2/7]

RDD vs. DSM - Read operation

- RDD
  - Either coarse grained or fine grained

- DSM
  - The read operation in Distributed shared memory is fine-grained

Benefits of RDDs as a distributed memory abstraction [3/7]

RDD vs. DSM - Write operation

- RDD
  - The write operation in RDD is coarse grained

- DSM
  - The write operation in Distributed shared memory is fine-grained
Benefits of RDDs as a distributed memory abstraction [4/7]

RDD vs. DSM- Consistency

• RDD
  • RDD is immutable in nature
    • Any changes on RDD is permanent
    • The level of consistency is high

• DSM
  • If the programmer follows the rules, the memory will be consistent and the results of memory operations will be predictable

Benefits of RDDs as a distributed memory abstraction [5/7]

RDD vs. DSM- Fault-Recovery Mechanism

• RDD
  • The lost data can be easily recovered in Spark RDD using lineage graph at any moment
  • For each transformation, new RDD is formed
  • RDDs are immutable

• DSM
  • Fault tolerance is achieved by a checkpointing technique which allows applications to roll back to a recent checkpoint rather than restarting
Benefits of RDDs as a distributed memory abstraction [6/7]

RDD vs. DSM - Straggler Mitigation

- Stragglers
  - Nodes taking more time to complete than their peers
  - Due to load imbalance, I/O blocks, garbage collections, etc.

- RDD
  - Creates backup copies of slow tasks
    - without accessing the same memory

- DSM
  - It is quite difficult to achieve straggler mitigation

---

Benefits of RDDs as a distributed memory abstraction [7/7]

- Runtime can schedule tasks based on data locality
  - To improve performance

- RDDs degrade gracefully when there is insufficient memory
  - Partitions that do not fit in the RAM are stored on disk
Applications not suitable for RDDs

• RDDs are best suited for batch applications that apply the same operations to all elements of a dataset
  • Steps are managed by lineage graph efficiently
  • Recovery is managed effectively

• RDDs would not be suitable for applications
  • Making asynchronous fine-grained updates to shared state
  • e.g. a storage system for a web application or an incremental web crawler

Large Scale Data Analytics
In-Memory Cluster Computing: Apache Spark
RDD in Spark
RDDS in Spark: The Runtime

User’s driver program launches multiple workers, which read data blocks from a distributed file system and can persist computed RDD partitions in memory.

Representing RDDs

- A set of partitions
  - Atomic pieces of the dataset

- A set of dependencies on parent RDDs

- A function for computing the dataset based on its parents

- Metadata about its partitioning scheme

- Data placement
Large Scale Data Analytics
In-Memory Cluster Computing: Apache Spark
RDD Dependency in Spark

Dependency between RDDs [1/4]

- **Narrow** dependency
- **Wide** dependency
Dependency between RDDs [2/4]

- **Narrow** dependency
  - Each partition of the parent RDD is used by **at most one partition** of the child RDD

Dependency between RDDs [3/4]

- **Wide** dependency
  - Multiple child partitions may depend on a single partition of parent RDD
Dependency between RDDs [4/4]

• Narrow dependency
  • Pipelined execution on one cluster node
  • e.g. a map followed by a filter
  • Failure recovery is more straightforward

• Wide dependency
  • Requires data from all parent partitions to be available and to be shuffled across the nodes
  • Failure recovery could involve a large number of RDDs
    • Complete re-execution may be required