CSE 444: Database Internals

Lecture 23
Spark

References

- Spark is an open source system from Berkeley
- Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Matei Zaharia et. al. NSDI'12.

Motivation

- Goal: Better use **distributed memory** in a cluster
- Observation:
  - Modern data analytics involves **iterations**
  - Users also want to do **interactive** data mining
  - In both cases, want to keep intermediate data in memory and reuse it
  - MapReduce does not support this scenario well
    - Requires writing data to disk between jobs

Approach

- New abstraction: Resilient Distributed Datasets
- RDD properties
  - Parallel data structure
  - Can be persisted in memory
  - Fault-tolerant
  - Users can manipulate RDDs with rich set of operators

RDD Details

- An RDD is a partitioned collection of records
  - RDD's are typed: RDD[Int] is an RDD of integers
- An RDD is read only
  - This means no updates to individual records
  - This is to contrast with in-memory key-value stores
- To create an RDD
  - Execute a deterministic operation on another RDD
  - Or on data in stable storage
  - Example operations: map, filter, and join

RDD Materialization

- Users control persistence and partitioning
- Persistence
  - Should we materialize this RDD in memory?
- Partitioning
  - Users can specify key for partitioning an RDD

Let's think about it…

- So RDD is a lot like a view in a parallel engine
- A view that can be materialized in memory
- A materialized view that can be physically tuned
  - Tuning: How to partition for maximum performance

Spark Programming Interface

- RDDs implemented in new Spark system
- Spark exposes RDDs though a language-integrated API similar to DryadLINQ but in Scala
- Later Spark was extended with SQL

Why Scala?

From Matei Zaharia (Spark lead author): "When we started Spark, we wanted it to have a concise API for users, which Scala did well. At the same time, we wanted it to be fast (to work on large datasets), so many scripting languages didn't fit the bill. Scala can be quite fast because it's statically typed and it compiles in a known way to the JVM. Finally, running on the JVM also let us call into other Java-based big data systems, such as Cassandra, HDFS and HBase.

Since we started, we've also added APIs in Java (which became much nicer with Java 8) and Python"

https://www.quora.com/Why-is-Apache-Spark-implemented-in-Scala

Querying/Processing RDDs

- Programmer first defines RDDs through **transformations** on data in stable storage
  - Map
  - Filter
  - …
- Then, can use RDDs in **actions**
  - Action returns a value to app or exports to storage
  - Count (counts elements in dataset)
  - Collect (returns elements themselves)
  - Save (output to stable storage)

Example (from paper)

Search logs stored in HDFS

```
lines = spark.textFile("hdfs://…")
errors = lines.filter(_.startsWith("Error"))
errors.persist()
errors.collect()
errors.filter(_.contains("MySQL")).count()
```


More on Programming Interface

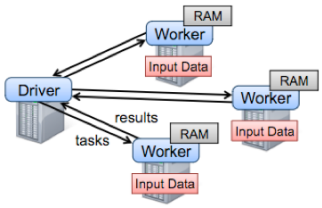
- Large set of pre-defined transformations:
  - Map, filter, flatMap, sample, groupByKey, reduceByKey, union, join, cogroup, crossProduct, …
- Small set of pre-defined actions:
  - Count, collect, reduce, lookup, and save
- Programming Interface includes **iterations**

More Complex Example

```
val points = spark.textFile(...)
                  .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
  val gradient = points.map{ p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a,b) => a+b)
  w -= gradient
}
```

[From Zaharia12]

Spark Runtime



1) Input data in HDFS Or other Hadoop input source

2) User writes driver program

3) System ships code to workers

Figure 2: Spark runtime. The user's driver program launches multiple workers, which read data blocks from a distributed file system and can persist computed RDD partitions in memory.

[From Zaharia12]

Query Execution Details

- Lazy evaluation
  - RDDs are not evaluated until an action is called
- In memory caching
  - Spark workers are long-lived processes
  - RDDs can be materialized in memory in workers
  - Base data is not cached in memory

Key Challenge

- How to provide fault-tolerance efficiently?

Fault-Tolerance Through Lineage

Represent RDD with 5 pieces of information

- A set of partitions
- A set of dependencies on parent partitions
  - Distinguishes between **narrow** (one-to-one)
  - And **wide** dependencies (one-to-many)
- Function to compute dataset based on parent
- Metadata about partitioning scheme and data placement

RDD = Distributed relation + lineage

More Details on Execution

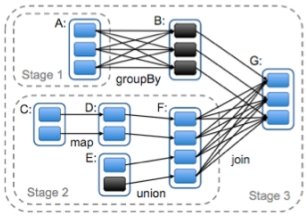


Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case, stage 1's output RDD is already in RAM, so we run stage 2 and then 3.

[From Zaharia12]

Scheduler builds a DAG of stages based on lineage graph of desired RDD.

Pipelined execution within stages

Synchronization barrier with materialization before shuffles

If a task fails, re-run it
Can checkpoint RDDs to disk

Latest Advances

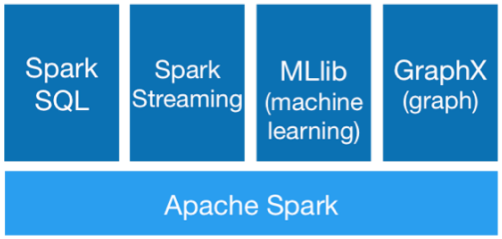


Image from: http://spark.apache.org/

Where to Go From Here

- Read about the latest Hadoop developments
  - YARN
- Read more about Spark
- Learn about GraphLab/Dato
- Learn about Impala, Flink, Myria, etc.
- … many other big data systems and tools...

- Also good to know latest cloud offering: Google, Microsoft, and Amazon