Apache Spark

Prajakta Kalmegh
Duke University
What is Spark?

Not a modified version of Hadoop

Separate, fast, MapReduce-like engine
- In-memory data storage for very fast iterative queries
- General execution graphs and powerful optimizations
- Up to 40x faster than Hadoop
- Up to 100x faster (2-10x on disk)

Compatible with Hadoop’s storage APIs
- Can read/write to any Hadoop-supported system, including HDFS, HBase, SequenceFiles, etc
Applications

In-memory analytics & anomaly detection (Conviva)
Interactive queries on data streams (Quantifind)
Exploratory log analysis (Foursquare)
Traffic estimation w/ GPS data (Mobile Millennium)
Twitter spam classification (Monarch)

...
Why a New Programming Model?

MapReduce greatly simplified big data analysis

But as soon as it got popular, users wanted more:
» More complex, multi-stage applications (graph algorithms, machine learning)
» More interactive ad-hoc queries
» More real-time online processing

All three of these apps require fast data sharing across parallel jobs

NOTE: What were the workarounds in MR world?
Ysmart [1], Stubby[2], PTF[3], Haloop [4], Twister [5]
Interactive speed

Most queries complete under 10 sec

Monthly query workload of one 3000-node Dremel instance

Dremel: Interactive Analysis of Web-Scale Datasets. VLDB'10
Data Sharing in MapReduce

Slow due to replication, serialization, and disk IO
Data Sharing in Spark

10-100× faster than network and disk
RDD: Spark Programming Model

Key idea: *resilient distributed datasets (RDDs)*
» Distributed collections of objects that can be cached in memory or disk across cluster nodes
» Manipulated through various parallel operators
» Automatically rebuilt on failure

Interface
» Clean language-integrated API in Scala
» Can be used *interactively* from Scala console
More on RDDs

• Immutable: RDD is a read-only, partitioned collection of records
  ‣ Checkpoint RDDs with long lineage chains can be done in the background.
  ‣ Using stragglers: We can use backup tasks to recompute transformations on RDDs

• Transformations: Created through deterministic operations on either
  ‣ data in stable storage or
  ‣ other RDDs

• Lineage: RDD has enough information about how it was derived from other datasets

• Persistence level: Users can choose a re-use storage strategy (caching in memory, storing the RDD only on disk or replicating it across machines; also chose a persistence priority for data spills)

• Partitioning: Users can ask that an RDD’s elements be partitioned across machines based on a key in each record
RDD Transformations and Actions

Note: Lazy Evaluation: A very important concept
DAG of RDDs

**RDD Objects**
- build operator DAG
  - rdd1.join(rdd2)
  - .groupBy(...)
  - .filter(...)

**DAGScheduler**
- split graph into stages of tasks
- submit each stage as ready
- agnostic to operators!

**TaskScheduler**
- launch tasks via cluster manager
- retry failed or straggling tasks
- doesn’t know about stages

**Worker**
- execute tasks
- store and serve blocks
- Threads
- Block manager

Fault Tolerance

RDDs track the series of transformations used to build them (their lineage) to recompute lost data

E.g: messages = textFile(...).filter(_.contains("error")) .map(_.split('t')(2))
Representing RDDs

- Graph-based representation. Five components:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>partitions()</td>
<td>Return a list of Partition objects</td>
</tr>
<tr>
<td>preferredLocations(p)</td>
<td>List nodes where partition $p$ can be accessed faster due to data locality</td>
</tr>
<tr>
<td>dependencies()</td>
<td>Return a list of dependencies</td>
</tr>
<tr>
<td>iterator($p$, parentIters)</td>
<td>Compute the elements of partition $p$ given iterators for its parent partitions</td>
</tr>
<tr>
<td>partitioner()</td>
<td>Return metadata specifying whether the RDD is hash/range partitioned</td>
</tr>
</tbody>
</table>

Table 3: Interface used to represent RDDs in Spark.
Representing RDDs (Dependencies)

Figure 4: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.
Representing RDDs (An example)

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.
Advantages of the RDD model

<table>
<thead>
<tr>
<th>Aspect</th>
<th>RDDs</th>
<th>Distr. Shared Mem.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reads</td>
<td>Coarse- or fine-grained</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Writes</td>
<td>Coarse-grained</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Consistency</td>
<td>Trivial (immutable)</td>
<td>Up to app / runtime</td>
</tr>
<tr>
<td>Fault recovery</td>
<td>Fine-grained and low-overhead using lineage</td>
<td>Requires checkpoints and program rollback</td>
</tr>
<tr>
<td>Straggler mitigation</td>
<td>Possible using backup tasks</td>
<td>Difficult</td>
</tr>
<tr>
<td>Work placement</td>
<td>Automatic based on data locality</td>
<td>Up to app (runtimes aim for transparency)</td>
</tr>
<tr>
<td>Behavior if not enough RAM</td>
<td>Similar to existing data flow systems</td>
<td>Poor performance (swapping?)</td>
</tr>
</tbody>
</table>

Table 1: Comparison of RDDs with distributed shared memory.
Other Engine Features: Implementation

- Not covered in details

Some **Summary**:
- Spark local vs Spark Standalone vs Spark cluster (Resource sharing handled by Yarn/Mesos)
- *Job Scheduling*: DAGScheduler vs TaskScheduler
- *Interpreter Integration*: Ship external instances of variables referenced in a closure along with the closure class to worker nodes in order to give them access to these variables
- *Memory Management*: serialized in-memory (fastest) VS deserialized in-memory VS on-disk persistent
- *Support for Checkpointing*: Tradeoff between using lineage for recomputing partitions VS checkpointing partitions on stable storage

End of Lecture 20
Shark: Hive on Spark
What is Shark?

Port of Apache Hive to run on Spark

Compatible with existing Hive data, metastores, and queries (HiveQL, UDFs, etc)

Similar speedups of up to 40x

Figure 1: Shark Architecture
Motivation

Hive is great, but Hadoop’s execution engine makes even the smallest queries take minutes

Scala is good for programmers, but many data users only know SQL

Can we extend Hive to run on Spark?
Hive Architecture

Meta store

Client
- CLI
- JDBC

Driver
- SQL Parser
- Query Optimizer
- Physical Plan
- Execution

MapReduce

HDFS

Borrowed slide
Shark Architecture

Meta store

Client

- CLI
- JDBC

Driver

- Cache Mgr.

SQL Parser

Query Optimizer

Physical Plan

Execution

Spark

HDFS

Borrowed slide
Shark Engine: Extensions to Hive

- PDE (Partial DAG Executions)
  - To Support dynamic query optimization
    - allows dynamic alteration of query plans based on data statistics collected at run-time
  - use PDE to optimize the global structure of the plan at stage boundaries

- Skew Handling and Degree of Parallelism
  - Importance of DoP for Mappers vs Reducers (too few can overload reducers)
  - Skew mitigation: Fine-grained partitions are assigned to coalesced partitions using a greedy bin-packing heuristic

- Distributed Data Loading
  - Loading tasks use the data schema to extract individual fields from rows
  - Marshal a partition of data into its columnar representation
  - Store those columns in memory
Shark Engine: Extensions to Hive

- Join Optimizations

Figure 3: Data flows for map join and shuffle join. Map join broadcasts the small table to all large table partitions, while shuffle join repartitions and shuffles both tables.
Efficient In-Memory Storage

Simply caching Hive records as Java objects is inefficient due to high per-object overhead.

Instead, Shark employs column-oriented storage using arrays of primitive types.
Efficient In-Memory Storage

Simply caching Hive records as Java objects is inefficient due to high per-object overhead.

Instead, Shark employs column-oriented storage using arrays of primitive types.

Row Storage

<table>
<thead>
<tr>
<th>3</th>
<th>4.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sally</td>
<td>3.5</td>
</tr>
<tr>
<td>6.4</td>
<td>6.4</td>
</tr>
</tbody>
</table>

Column Storage

Benefit: similarly compact size to serialized data, but >5x faster to access.
Shark vs Spark SQL
Spark SQL Overview

- Newest component of Spark initially contributed by databricks (< 1 year old)
- Tightly integrated way to work with structured data (tables with rows/columns)
- Transform RDDs using SQL
- Data source integration: Hive, Parquet, JSON, and more
Spark SQL

A General Stack

Spark SQL
Spark Streaming real-time
GraphX graph
MLlib machine learning

Spark
Powerful Stack – Agile Development

- Hadoop MapReduce
- Storm (Streaming)
- Impala (SQL)
- Giraph (Graph)
- Spark

Your App?
- GraphX
- Spark SQL
- Streaming

non-test, non-example source lines

- amplab UC Berkeley
- databricks™
Relationship to SHARK

Shark modified the Hive backend to run over Spark, but had two challenges:

- Limited integration with Spark programs
- Hive optimizer not designed for Spark

Spark SQL reuses the best parts of Shark:

Borrows
- Hive data loading
- In-memory column store

Adds
- RDD-aware optimizer
- Rich language interfaces
Adding Schema to RDDs

Spark + RDDs
*Functional* transformations on partitioned collections of *opaque objects.*

SQL + SchemaRDDs
*Declarative* transformations on partitioned collections of *tuples.*
Example Dataset

A text file filled with people’s names and ages:

Michael, 30
Andy, 31
...

amplab UC BERKELEY databricks
RDDs as Relations (Scala)

```scala
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
import sqlContext._

// Define the schema using a case class.
case class Person(name: String, age: Int)

// Create an RDD of Person objects and register it as a table.
val people =
    sc.textFile("examples/src/main/resources/people.txt")
    .map(_.split","))
    .map(p => Person(p(0), p(1).trim.toInt))

people.registerAsTable("people")
```
Querying Using SQL

# SQL can be run over SchemaRDDS that have been registered
# as a table.
teenagers = sqlCtx.sql(""
    SELECT name FROM people WHERE age >= 13 AND age <= 19"")

# The results of SQL queries are RDDs and support all the normal
# RDD operations.
teenNames = teenagers.map(lambda p: "Name: " + p.name)
Reading Data Stored in Hive

```python
from pyspark.sql import HiveContext
hiveCtx = HiveContext(sc)

hiveCtx.hql(""
    CREATE TABLE IF NOT EXISTS src (key INT, value STRING)"")

hiveCtx.hql(""
    LOAD DATA LOCAL INPATH 'examples/.../kv1.txt' INTO TABLE src"")

# Queries can be expressed in HiveQL.
results = hiveCtx.hql("FROM src SELECT key, value").collect()
```
References:

IEEE Computer Society, Washington, DC, USA, 25-36


Computing (HPDC ’10). ACM, New York, NY, USA, 810-818

[6] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott
Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI’12). USENIX Association,
Berkeley, CA, USA, 2-2.

data?related=1>