Resilient Distributed Datasets

Presented by Henggang Cui
15799b Talk
Why not MapReduce

• Provide fault-tolerance, but:

• Hard to reuse intermediate results across multiple computations
  – stable storage for sharing data across jobs

• Hard to support interactive ad-hoc queries
Why not Other In-Memory Storage

• Examples: Piccolo
  – Apply fine-grained updates to shared states

• Efficient, but:

• Hard to provide fault-tolerance
  – need replication or checkpointing
Resilient Distributed Datasets (RDDs)

- Restricted form of distributed shared memory
  - read-only, partitioned collection of records
  - can only be built through coarse-grained deterministic transformations
    - data in stable storage
    - transformations from other RDDs.

- Express computation by
  - defining RDDs
Fault Recovery

• Efficient fault recovery using lineage
  – log one operation to apply to many elements (lineage)
  – recompute lost partitions on failure
Example

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
hdfs_errors = errors.filter(_.contains("HDFS"))
```
Advantages of the RDD Model

• Efficient fault recovery
  – fine-grained and low-overhead using lineage

• Immutable nature can mitigate stragglers
  – backup tasks to mitigate stragglers

• Graceful degradation when RAM is not enough
Spark

• Implementation of the RDD abstraction
  – Scala interface
• Two components
  – Driver
  – Workers
Spark Runtime

• **Driver**
  – defines and invokes actions on RDDs
  – tracks the RDDs’ lineage

• **Workers**
  – store RDD partitions
  – perform RDD transformations
Supported RDD Operations

• Transformations
  – map (f: T->U)
  – filter (f: T->Bool)
  – join()
  – ... (and lots of others)

• Actions
  – count()
  – save()
  – ... (and lots of others)
Representing RDDs

• A graph-based representation for RDDs

• Pieces of information for each RDD
  – a set of partitions
  – a set of dependencies on parent RDDs
  – a function for computing it from its parents
  – metadata about its partitioning scheme and data placement
RDD Dependencies

• Narrow dependencies
  – each partition of the parent RDD is used by at most one partition of the child RDD

• Wide dependencies
  – multiple child partitions may depend on it
RDD Dependencies

Narrow Dependencies:
- map, filter
- union
- join with inputs co-partitioned

Wide Dependencies:
- groupByKey
- join with inputs not co-partitioned
RDD Dependencies

- Narrow dependencies
  - allow for pipelined execution on one cluster node
  - easy fault recovery

- Wide dependencies
  - require data from all parent partitions to be available and to be shuffled across the nodes
  - a single failed node might cause a complete re-execution.
Job Scheduling

• To execute an action on an RDD
  – scheduler decide the stages from the RDD’s lineage graph
  – each stage contains as many pipelined transformations with narrow dependencies as possible
Job Scheduling

Stage 1
A: [Diagram]
B: groupBy

Stage 2
C: [Diagram]
D: map
E: [Diagram]

Stage 3
F: union
G: join

16
Memory Management

• Three options for persistent RDDs
  – in-memory storage as deserialized Java objects
  – in-memory storage as serialized data
  – on-disk storage

• LRU eviction policy at the level of RDDs
  – when there’s not enough memory, evict a partition from the least recently accessed RDD
Checkpointing

• Checkpoint RDDs to prevent long lineage chains during fault recovery

• Simpler to checkpoint than shared memory
  – Read-only nature of RDDs
Discussions
Checkpointing or Versioning?

- Frequent checkpointing, or
  Keep all versions of ranks?