Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Zaharia et al.
NSDI 2012
BigData compute

• What’s the context for Spark? (Year is 2012)

• MapReduce: based on a distributed FS (HDFS, or GFS), used disk for all the data
  • Google search -> PageRank -> Graph of the web as input -> vertex ranks as output (run as infrequently as users are willing to tolerate)
  • Bulk/batch data processing that happens periodically (weekly/monthly) :: okay to be inefficient

• What’s changing?
  • 64-bit OS happens around this time
  • Memory grows in size (1GB -> 4GB)

• Spark: in-memory — efficiency
  • Also bulk, includes complex topologies like iterative ML algos
  • On-demand / interactive / lazy / ad-hoc / unscheduled / one-off :: person who is waiting for a result => efficiency becomes a key concern
  • Trending towards data science

• Context: BigData becomes more common. MR invented at Google. But… over time BigData being generated/monetized everywhere! If you’re not Google and you want interactivity with BigData, you need efficiency.

• Systems research: new abstraction (low availability: few people) -> broaden adoption (research focus turns to efficiency)
Spark key ideas

- **RDD**: resilient distributed datasets
  - **Read-only = immutable**
  - *Created using coarse-grained operations (in contrast to DSM)*
  - Tranformations: RDD -> RDD
  - Actions: materialization of RDD data in a specific location
  - RDD iface: deps, compute, partitioning, location

- RDD lineage: connect RDD/partitions into a dep graph. Keep track of which RDDs are available (caching), use graph for fault tolerance

- Keep RDD data in memory unserialized (or serialized memory, or on disk)

- RDD metadata is tiny (and kept at central node)
Big deal about immutability

- Functional!
- **Immutability**: never modify an RDD, create a new one
  - Simplifies concurrency control: have multiple nodes working on the same RDD without conflict (multiple nodes can read, and create a different, independent RDD)
  - Larger memory requirement (but only if materialized)
  - Expression *purity*: operations determinism
  - Building an RDD = building a description of data. Operations on data ~ operations on description of previous operations over data — Lazy evaluation delays computation, which allows the compiler/runtime to make a bunch of optimizations
Distributed systems + Spark

• Spark’s immutability \(\iff\) distribution

• Previously: replication as a key FT mechanism

• Fault tolerance: can recompute/recover the missing data based on lineage graph
  • Only pay fault tolerance cost during failure: no need for replicas that keep up with each other

• Why can’t I use immutability for consensus?
  • RDD per consensus ballot, or for all consensus state?
  • Lineage would provide the ordering for you (requires dep. between RDDs)
  • Orthogonal? RDD captures what you should be doing, so once you have an RDD, you know what to do (consensus decides what work should be done)
  • Consensus assumes work is trivial; RDDs focus on the work part (consensus is never actually deployed for its own sake: you use it as a means towards something else)
  • CRDTs ~ immutable view on “eventual” consensus (with very different consistency semantics)
    • RDD lineage graph ~ CRDT lattice (only proceed forward)
    • Paxos only proceed forward with counters
  • Consensus immutability = once a decision is made, any future decision must come to the same thing
Spark the implementation

- What do I need to realize spark (besides the abstraction)?
- “Implement 1/2 of Emerald”: RDDs encompass data; need to migrate these. Objects of compute, also have to be serialized/migrated.
- Spark – take a program, compile it with knowledge about eventual deployment, deploy it/orchestrate the runtime.
- Driver that knows all the things, a worker node may need to lookup RDD state/data location from this central driver
- Integration with the Scala interpreter (no changes to compiler): packaging and shipping code to nodes
- Job scheduler: assign RDDs to nodes (efficient assignment is key)
  - Move compute to data (co-location)
  - Sequencing in executing stages: execute dependencies first!
  - Pipelining of narrow dependency ops on the same compute node: best distributed compute is local compute (as long as you have resources)
- Memory management: need a policy to determine with RDDs live in memory and which do not
  - LRU policy for deciding RDDs in memory
  - Data can be dropped entirely, since there is a lineage plan to recompute it
- Monitoring: detect faults, and recover/reassign work
  - Simplest part of the entire idea (immutability + lineage gives us “free” fault tolerance)
- Debugging (tbd): Linear graphs gives you an easy view on the entire computation that you can analyze/visualize/…
Spark the implementation

- When is a good time/place for checkpointing (forced materialization)?
  - Paper: at wide dependencies for efficient FT
  - In general: avoid high cost of recomputes
  - Wide dependency as a proxy ~ involves many nodes; involves many RDDs => definitely not pipelineable, so more costly to recomputes
Spark eval

• Key eval criteria:
  • End-to-end time to compute
  • Scalability (time versus # nodes)
  • Efficiency (cost of a no-op)

• Baseline:
  • Hadoop (MR): disk-based
  • HadoopBinMem: materialized in-memory dataset; memory-based, but has all the other Hadoop costs

• Spark deals with Java objects in memory (best case): formats matter!

• Discussion section: RDDs encompass MR, DryadLINQ, SQL, Pregel, iterative MR, Stream processing…

• Will we ever need processing that is not Spark-based?
  • Fine-grained operations (not bulk processing)
  • Non-deterministic / external inputs (sensors)
Next: TensorFlow

- ML-specific distributed computing framework

- *How does it build on and also differ from Spark?*