Big Data - Spark

Overview

- Resilient Distributed Datasets (RDDs) which allow programmers to perform in-memory computations on large clusters in a fault-tolerant manner
- Motivated by two types of applications:
  - Iterative algorithms
  - Interactive data mining tools
- Performance of these can be improved by keeping data in memory
- To provide fault-tolerance, provides a restricted form of shared memory based on coarse-grain transformations
  - Can capture a wide class of computations

- Frameworks like MapReduce and Dryad have been widely adopted because they let users write parallel computations using high-level operators without worrying about work distribution / fault tolerance
  - They allow abstractions to access computational resources, but not distributed memory
  - This makes them inefficient for applications that reuse intermediate results across multiple computations
    - The only way to reuse data between computations (i.e. between MapReduce jobs) is to write it to external storage
- RDDs seek to allow users to explicitly persist intermediate results in memory, control their partitioning, and manipulate them
  - Existing abstractions that provide fault tolerance require replicating fine data across machines, which means that we have a lot of overhead which scales proportional to the data that we need to send
  - RDDs provide coarse operations (map, filter, join) that apply the same operation to many data items
    - We can then log these transformations rather than the actual data, allowing us to provide fault tolerance
    - If the log grows large, we can potentially checkpoint the data
- If a partition of an RDD is lost, then the RDD has enough information about how it was derived in its logs to recompute it

RDDs

- Formally, it is a read-only, partitioned collection of records
- Can be created through deterministic operations on data in stable storage or other RDDs
  - These are known as **transformations**
  - Can include map, filter, join
- Other actions act on RDDs to return a value to the application
  - Can include count, collect, save
- Users can control RDDs storage strategy (i.e. its persistence)
- Can also control how it is partitioned (i.e. based on a certain key)
- In Spark, each dataset is represented as an object and transformations act on them
  - RDDs are computed lazily the first time they are used in an action so they can be pipelined later
  - Programmers can call a persist method to indicate which ones to reuse
  - Spark tries to keep RDDs in memory by default, but it can spill to disk if there is not enough RAM
- RDDs are more restricted compared to distributed memory in that they are for applications that do bulk writes, but this allows for efficient fault tolerance
  - They do not need to incur overhead of checkpointing and the lost partitiosn can be reocmputed upon failure
- The immutable nature of RDDs also allows system to mitigate slow nodes by running backup copies just like MapReduce

    - This isn't possible with DSM (distributed shared memory)
- RDDs can use data locality to improve performance (using physical location of data
- RDDs are suitable for batch operations that apply the same operation to all elements
    - Not good for something like a storage system or incremental web crawler
    - These can be left to designing specialized systems