ECE 473/573
Cloud Computing and Cloud Native Systems
Lecture 27 Batch and Stream Processing II

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

November 19, 2025

Outline

Resilient Distributed Datasets and Apache Spark

RDD Implementation Details

Reading Assignment

- This lecture: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing `http://people.csail.mit.edu/matei/papers/2012/nsdi_spark.pdf`
- Next lecture (11/24): please watch this video on cloud security architecture rom RSA Conference 2019 `https://www.youtube.com/watch?v=4TxvqZFMaoA`
  - No in-person or zoom session.
- Last week (12/1, 12/3): consensus algorithms

Outline

Resilient Distributed Datasets and Apache Spark

RDD Implementation Details

Motivation

- Google MapReduce and similar implementations store task outputs to drives before they are used as inputs to other tasks
  - A lot of overhead in disk I/O and serialization
- This is ineffienct for iterative algorithms where intermediate results are reused frequently across multiple computations.
  - E.g. for machine learning and graph algorithms.
- Interactive tasks would also require a faster turnaround time.
- Can we make better use of the memory distributed across machines in the whole cluster?
  - What is the main reason for MapReduce to store intermediate results to drives?

Resilient Distributed Datasets (RDDs)

- A fault-tolerant and parallel data structure.
- Allow users to explicitly persist intermediate results in memory, with partitioning to optimize data placement.
- Manipulate via coarse-grained transformations.
  - Avoid costly replications for fault tolerance.
  - Transformations are idempotent: record and reapply them to rebuild the data set if it is lost due to failures.
- Applicable to computations where the same operation is applied to multiple data items.
  - A good fit for many parallel applications.
- Supported via Apache Spark, an open-source framework running on top of JVM for data processing on clusters.

RDD Abstraction

- An RDD is a read-only, partitioned collection of records.
  - Distributed across many machines.
  - Created from data in stable storage that will survive failures, or
  - From other RDDs via transformations like map and filter.
  - Actions like count and save output data derived from RDDs to be consumed by other systems.
- Transformations are lazy operations.
  - Enable optimizations across multiple transformations.
  - A program can recompute a RDD after failure if lineage is known (how to compute it from data in stable storage).
- Users control persistence and partitioning for RDDs.
  - Persistence defines RDDs that will be reused, and chooses a storage strategy like in-memory to save I/O.
  - Partitioning controls placement of records, usually via a key within them, e.g. to make certain records from two RDDs available on the same machine when generating a new RDD.

Example: Log Analysis

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))

errors.persist() // make errors reusable later
errors.count() // action: count errors

// Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")).count()

// Return the time fields of errors mentioning HDFS as an array
// (assuming time is field number 3 in a tab-separated format):
errors.filter(_.contains("HDFS"))
  .map(_.split('\t')(3))
  .collect()
```

- Process log messages to locate errors.
  - In Scala where _ starts an anonymous function.
- Once `errors` are available from memory, subsequent queries can be answered quickly, supporting interactive applications.

Example: Data Query

```
// SELECT users.id, users.name, SUM(orders.total) total_spending,
// FROM users JOIN orders ON (users.id=orders.buyer_id)
// GROUP BY users.id

// format: id, name, etc.
users = spark.textFile("hdfs://.../users.csv")
  .filter(_.nonEmpty).map(_.split("[,\t]"))
  .map(user => (user(0).toLong, user(1)))

// format: order_id, buyer_id, total, etc.
orders = spark.textFile("hdfs://.../orders.csv")
  .filter(_.nonEmpty).map(_.split("[,\t]"))
  .map(order => (order(1).toLong, order(2).toDouble))

// group by buyer_id and calculate total_spending
totalSpendings = orders.reduceByKey(_ + _)

// apply join to remove buyers without a matching user
joined = users.join(totalSpendings)

// now joined is a RDD of (user.id, (user.name, total_spending))
```

Outline

Resilient Distributed Datasets and Apache Spark

RDD Implementation Details

RDD Representation

- Each RDD consists of
  - Partitions as atomic piece of dataset.
  - Dependencies to parent RDDs.
  - A function to compute it from parent RDDs.
  - Metadata of partitioning scheme and data placement.
- Dependencies define communication needs.
  - Narrow dependency: each partition of the parent RDD is used by at most one partition of the child RDD, e.g. map and filter.
  - Wide dependency: multiple child partitions may depend on a single partition of the parent RDD, e.g. join and groupByKey.
  - Narrow dependencies allow for pipelined execution on a single node, elimating communication and simplifying fault recovery.
  - Wide dependencies require communications like MapReduce, and need complete re-execution for lost partitions.

Job Scheduling

- Since transformations are lazy, scheduling is trigger by actions.
- The scheduler will build a plan to compute the RDDs.
  - From RDD's lineage graph, as a directed acyclic graph (DAG) where vertices are partitions and edges are transformations.
  - The DAG is optimized by grouping vertices into stages, where within each stage transformations are merged, and no intermediate partitions are stored or communicated.
- The scheduler then decides what partitions are available and schedules tasks to compute missing partitions.
  - Follow the order of DAG to only schedule a task when all its input partitions become available.
  - Consider locality of data either in-memory or on-disk.
- Rerun failed task, persist RDDs to local drives if they require expensive communications to compute.

Memory Management

- RDD persistence options
  - In-memory storage as deserialized Java objects: fastest performance but large overhead in memory usage.
  - In-memory storage as serialized data: efficient memory usage.
  - On-disk storage: slowest, for RDDs too large to fit into memory, or too costly to recompute, usually due to expensive communication requirements from wide dependencies.
- Apply LRU policy to evict RDDs to make memory available.

Checkpointing

- While one can always recompute RDDs given their lineages, it is not efficient to do so for long lineage chains.
  - Specifically for wide dependencies within that require expensive communications.
- Checkpointing: store RDDs as output from wide dependencies on long lineage chains into stable storage.
  - Replicated as needed so no need to recompute if nodes fail.
  - RDDs are read-only so they can be written out in the background without impacting computation.
- Should checkpointing be specified by users or automatically decided?

Summary

- RDDs improve performance of distributed algorithms by making better use of local memory and CPUs to save on expensive disk and network I/Os.