ECE 473/573
Cloud Computing and Cloud Native Systems
Lecture 23 Batch and Stream Processing I

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

November 11, 2024

Outline

Computing with MapReduce

Google MapReduce

Resilient Distributed Datasets and Apache Spark

Reading Assignment

- This and next lecture:
  - MapReduce: Simplified Data Processing on Large Clusters `https://research.google/pubs/pub62/`
  - Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing `http://people.csail.mit.edu/matei/papers/2012/nsdi_spark.pdf`
- We will also introduce cryptography for cloud security next lecture.

Outline

Computing with MapReduce

Google MapReduce

Resilient Distributed Datasets and Apache Spark

MapReduce Model

- A model to specify parallel algorithms.
  - Consist of tasks that communicates with each other.
- A few types of tasks: input, map, combine, reduce, output.
- Communication is implicit: tasks communicate by exchanging their inputs/outputs.
  - Inputs/outputs are (key,value) pairs where key indicates the destination and value is the payload.
  - Pre-defined communication patterns: input $\rightarrow$ map $\rightarrow$ combine $\rightarrow$ reduce $\rightarrow$ output.
- Simplify parallel programming on clusters.
  - Easy to reason with pre-defined communication patterns.
  - Usually the map and the reduce tasks are specified by users.
  - Underlying implementations like Apache Hadoop provides cluster management for tasks scheduling, data movement, fault resilience, etc.

Map Tasks

```
class Map_WordCount extends ... {
  public void map(
    LongWritable key, Text value,
    OutputCollector<Text, IntWritable> output,
    Reporter reporter) throws IOException {

    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);
    while (tokenizer.hasMoreTokens()) {
      output.collect(
        new Text(tokenizer.nextToken()),
        new IntWritable(1));
    }
  }
}
```

- A map task consume what an input task generate and outputs pairs to combine tasks.
- Multiple map tasks running in parallel are able to consume and generate a lot of data.

Reduce Tasks

```
class Reduce_WordCount extends ... {
  public void reduce(
    Text key, Iterator<IntWritable> values,
    OutputCollector<Text, IntWritable> output,
    Reporter reporter) throws IOException {

    int sum = 0;
    while (values.hasNext()) {
      sum += values.next().get();
    }
    output.collect(key, new IntWritable(sum));
  }
}
```

- Combine tasks group output pairs from map tasks by keys, and output these groups.
- A reduce task consumes a key and the associated values, and generate pairs for output tasks.

Discussions

- Good for embarrassingly parallel algorithms.
  - It was difficult to implement and deploy parallel algorithms, even if they are conceptually simple, because one also need to manage the cluster.
- Advantanges
  - Theorectially deadlock free with predefined communication patterns and no other synchronization between tasks.
  - Stateless tasks are idempotent, which makes it possible to build fault resilient implementations.

Outline

Computing with MapReduce

Google MapReduce

Resilient Distributed Datasets and Apache Spark

Google MapReduce

- Research paper published in 2004.
  - One of the earliest work of cloud computing.
- Originated from Google's need to analyze large-scale web data efficiently, e.g.
  - Build reverse index for searching
  - Process logs to calculate URL access frequency
  - Reverse web-link graph for page ranking
- On a large cluster of commodity servers.
  - Instead of HPCs.
  - Provide scalability by adding more servers.
  - Fault resilience as servers fail, which is more likely to happen as number of servers increase.

Cluster Hardware

(Keep in mind this was around 2004.)

- Large clusters of commodity PCs connected with Ethernet.
- Dual-processor with 2-4GB memory running Linux.
- Commodity networking hardware with 100Mb or 1Gb connections to individual machine.
  - Bottlenecks may exist if many machines need to talk with many other machines at the same time.
- Storage provided by inexpensive hard drives attached to machines locally.
- Failures are common with hundreds or thousands of machines.

Execution Flow

- User program provides a map function and a reduce function.
  - Assume there will be M map tasks and R reduce tasks.
  - M and R should be larger than available number of machines.
- The MapReduce library splits input files into M chunks and starts up copies of user program on many machines.
- A copy of the program runs as master and the rest are workers. Master assign map or reduce tasks to idle workers.
- A map worker calls user's map function to read an input chunk and outputs key/value pairs to a memory buffer.
- Pairs in memory buffer are written to local disk periodically.
  - The pairs are partitioned into R regions on the disk, one for each reduce task, according to the keys.
  - Locations of the regions are passed to master, and then forwarded to reduce workers.

Execution Flow (cont.)

- A reduce worker receiving locations from master will request its regions from map workers via RPC.
  - There are more keys than R so the regions for a single reduce task will contain many keys.
  - The reduce worker groups pairs by their keys.
- The reduce worker calls user's reduce function multiple times, one for each group of pairs with the same key.
  - Outputs from these function calls are appended to the end of the final output file of this reduce task.
- The master notifies the user program when all map and reduce tasks complete.
  - Results are available from R final output files – usually as inputs to other MapReduce calls or distributed applications.

Locality

- Both input files and final output files are stored in a distributed file system.
  - On local drives of the machines across the whole cluster.
  - Data are replicated to survive machine failures.
- Network bandwidth is a relatively scarce resource.
  - Whenever possible, schedule a map task to a worker where the input data is available locally.
  - If not possible, schedule it to the worker that is close to the input data to reduce overall network traffic.

Batch Processing

- High system utilization to reduce cost of computing.
  - Leverage paralellism within large amount of data to process them in parallel.
  - Many different keys and many pairs lead to large M and R.
  - Large M and R keep all workers busy, saturating compuational resources like CPU, memory, local drives, and networking.
- High latency from when inputs are available to when outputs are computed.
  - Cannot complete processing for a key before all pairs with the same key become available to the reduce worker.
  - Pairs need to be written to local storage first.
  - Pairs need to be sent across network to a different worker.
  - A single bad worker may delay the completion of the whole computation.

Fault Tolerance

- Worker failure
  - Each task has a state among idle (wait for scheduling), in-progress, and completed.
  - Master discovers worker failures via liveness check.
  - Completed reduce tasks on failed workers, if the final output files are available from replicas, need no further action.
  - All other tasks on failed workers (completed map tasks, in-progress map and reduce tasks) are marked as idle, waiting to be scheduled again.
  - Running a task multiple times won't cause issues as map and reduce functions are stateless and idempotent.
- Master failure
  - Master state includes states of tasks and which workers run them if they are in-progress.
  - Master may write its state to storage periodically so it could restart from a previously known state.
  - Nevertheless, it is less likely master will fail so one just restart the whole process if it fails.

Outline

Computing with MapReduce

Google MapReduce

Resilient Distributed Datasets and Apache Spark

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.

Spark Example

```
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.

Summary

- What Google MapReduce trying to achieve becomes common practice for cloud computing nowadays.