ECE 473/573
Cloud Computing and Cloud Native Systems
Lecture 26 Batch and Stream Processing I

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

November 17, 2025

Outline

Computing with MapReduce

Google MapReduce

Reading Assignment

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

Outline

Computing with MapReduce

Google MapReduce

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.

Example: Sparse Matrix Multiplication I

- Matrix multiplication: $C = A \times B$, or $c_{i,j} = \sum_k a_{i,k} b_{k,j}$
- Sparse matrices: most of $a_{i,k}$, $b_{k,j}$ and thus $c_{i,j}$ are 0.
- A lot of tasks have similar communication pattern as sparse matrix multiplication problems.
- Naive MapReduce implementation
  - For each $a_{i,k} \neq 0$, map emits (multiple) $(i,j) \to a_{i,k}$ for all j.
  - For each $b_{k,j} \neq 0$, map emits (multiple) $(i,j) \to b_{k,j}$ for all i.
  - One reduce task per (i,j)
    - Receive $a_{i,k_1}, a_{i,k_2}, \ldots$, and $b_{k'_1,j}, b_{k'_2,j}, \ldots$
    - Compute $c(i,j)$ by summing $a_{i,k} b_{k',j}$ for $k = k'$.
- Since most $c_{i,j}$ are 0, the naive implementation is not efficient because all communications and computations at the corresponding reduce tasks are wasted.

Example: Sparse Matrix Multiplication II

- A better implementation
  - For each $a_{i,k} \neq 0$, map emits $k \to a_{i,k}$
  - For each $b_{k,j} \neq 0$, map emits $k \to b_{k,j}$
  - One reduce task per k
    - Receive $a_{i_1,k}, a_{i_2,k}, \ldots$, and $b_{k,j_1}, b_{k,j_2}, \ldots$
    - Emit (multiple) $(i,j) \to a_{i,k}b_{k,j}$ for each pair of (i,j) from $\{i_1, i_2, \ldots\} \times \{j_1, j_2, \ldots\}$.
  - Let another set of reduce tasks, one per each (i,j) that has any output to sum $a_{i,k}b_{k,j}$ into $c(i,j)$
- Most communications and computations where $c_{i,j} = 0$ are eliminated.

Example: Sparse Matrix Multiplication III

- Many sparse matrices have only a limited number of non-zero elements per column and per row.
  - If a single row or column can fit into a single machine, can we optimize the algorithm further?
- An optimized implementation
  - For each $a_{i,k} \neq 0$, map emits $k \rightarrow a_{i,k}$
  - For each row k of B, map emits $k \rightarrow (b_{k,j_1}, b_{k,j_2}, \ldots)$, non-zero elements only.
  - One reduce task per k
    - Receive $a_{i_1,k}, a_{i_2,k}, \ldots$, and $b_{k,j_1}, b_{k,j_2}, \ldots$
    - Emit (multiple) $i \rightarrow (a_{i,k} b_{k,j_1}, a_{i,k} b_{k,j_2}, \ldots)$ for each i from $\{i_1, i_2, \ldots\}$
  - Let another set of reduce tasks, one per each i that has any output to compute row i of C
- Aggregate and reduce communications on rows of B and C since they can be held in memory.

Outline

Computing with MapReduce

Google MapReduce

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.

Summary

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