A Comparative Systems Study of Apache Spark and Hadoop MapReduce Using Word Count

Proposal

We will expand on homework by implementing a word count program using different methodologies and evaluating their performance. Methodologies will include the following.

- Spark dataframes: this will align with the method used for homework 3.
- Hadoop MapReduce: Use Hadoop's MapReduce framework to process text into a list of words and later reduce the words by key.

1. Performance evaluations will compare the runtime when using different data set sizes and different numbers of cores.

- Data set size: toy dataset (megabytes), small dataset (gigabytes, fits in memory), larger dataset (tens of gigabytes, exceeds memory). We will source data from Project Gutenberg, which offers public domain books in raw text. We'll use Python to programmatically download thousands of books to reach the required amount of data. If needed, we can use non-English books to reach the needed dataset sizes.
    - **Toy dataset (10–100 MB):** Used to measure Spark/Hadoop framework initialization overhead, not scalability.
    - **Small dataset (1–5 GB, fits in memory):** Expected to favor Spark due to in-memory DAG execution
    - **Large dataset (20–50+ GB, exceeds memory):** Expected to stress-test disk spills and HDFS shuffling.
    - Metrics for comparison include:
        - **Runtime:** Wall-clock time from job submission to completion. Measured via Spark/Hadoop job logs.
        - **Throughput:** MB or words processed per second.

        - **Scalability:** T_1 / $T_{\square}$ across core configurations (1, 2, 4, 8 cores).
        - **Parallelism Utilization:** CPU time vs idle time, measured via Spark UI/YARN ResourceManager.
        - **Initialization Overhead:** Time to start the job before the map/reduce or DAG begins. Especially important for toy datasets.
    - Number of cores: 1 core, 2 cores, 4 cores, 8 cores.
        - Spark: control via `--master local[n]` or cluster submissions.
        - Hadoop: control via `mapreduce.job.reduces` and CPU slot configuration.
    - Stretch goal: GPU processing (Spark + RAPIDS Accelerator)
        - Use **Spark + RAPIDS Accelerator** to test GPU speedup on tokenization/shuffling stages.
        - Compare GPU vs CPU performance only on the small/mid dataset due to GPU memory limits.

2. We will also consider how each method performs under stress testing. These tests will examine node failure mid-process. Metrics include:

- Time to recover
- Recomputation overhead

3. We will try to run experiments on multiple cluster configurations using AWS/GCP. We plan on exploring Amazon EMR or deploying from scratch on Amazon EC2.

Cluster Sizes:

- Small: 2
- Medium: 5
- Large: 10

We will assess the speed-up as cluster size increases and compare how both methods perform.

4. We will compare resource utilization using both methods on a particular fixed dataset. Metrics include:

- CPU utilization

- Memory usage
- Disk I/O (Hypothesis: Hadoop should show higher disk I/O usage)

5. We will analyze how both frameworks manage in-memory data. We plan on running the following experiments:

- Cache (`.cache()` / `.persist()`) vs no cache. Metrics include execution time, memory usage.
- Assessing Spill Behaviour on both Hadoop and Spark by controlling `mapreduce.task.io.sort.mb` and `spark.memory.fraction` parameters. Metrics include the number of spills, disk I/O volume, and execution time.

6. We will analyze the sensitivity of both Hadoop and Spark to configuration parameters.

- Vary parameters `mapreduce.job.reduces` and `spark.sql.shuffle.partitions` to assess the trade-off between parallelism and processing overhead that comes with managing the parallelism.

Milestone Goals

1. Refine code from homework 3 to count words with Spark DataFrames. (Included for posterity, as this was already done for homework 3). Write code to count words using Hadoop's MapReduce framework.
2. Test each method across data set sizes.
3. Explore ERM and AWS to deploy different cluster sizes.
4. Write multi-cluster implementations and run both Spark and Hadoop methods using YARN on the cloud.
5. Implement a way to cause a core to fail, such that we can stress test core failures. We can use the os library to send a segmentation fault to a core.
6. Write code using the psutil Python library to keep track of CPU, memory, and disk I/O utilization trends.
7. Run testing matrix.

Final Project Goals and Deliverables

- A chart showing the results of each of our performance benchmarks with each of our word count implementations. More details on the benchmarks and implementations are included above.
- Python and Java code for our word count implementations.
- The code for different benchmarks implemented.