MapReduce Part 1: Comprehensive Review

DSC 208R - Data Management for Analytics

Introduction to Dataflow Systems

Dataflow systems, such as MapReduce and Spark, emerged to address limitations of traditional parallel Relational Database Management Systems (RDBMSs) in handling the scale and demands of web giants.

Parallel RDBMSs

Parallel RDBMSs are widely successful, typically employing shared-nothing data parallelism. They offer:

- Optimized runtime performance.
- Enterprise-grade features including ANSI SQL support, Business Intelligence (BI) dashboards/APIs, transaction management, crash recovery, and automated tuning with indexes.

Despite their strengths, new challenges from the web/internet giants necessitated moving beyond traditional RDBMSs.

Beyond RDBMSs: A Brief History and New Concerns

The rise of web giants like Google and Amazon introduced four new concerns that RDBMSs were not originally built to handle efficiently:

1. **Developability**: Custom data models and complex computations were hard to program using standard SQL/RDBMS APIs, leading to a demand for simpler programming interfaces.
2. **Fault Tolerance**: As systems scaled to thousands of machines, graceful handling of individual worker failures became critical.
3. **Elasticity**: The need to easily scale cluster size up or down based on fluctuating workload demands became essential.
4. **Cost**: Commercial RDBMS licenses were too expensive for the massive scale required, prompting companies to build their own custom, cost-effective systems.

This led to a new breed of parallel data systems, known as Dataflow Systems, which transformed the landscape of data processing.

What is MapReduce?

MapReduce is a seminal dataflow system (and programming model) developed by Google to process vast amounts of data in a fault-tolerant and highly scalable manner. It simplifies distributed programming by abstracting away many complexities.

Standard Example: Word Count

A classic example for illustrating MapReduce is counting word occurrences in a document corpus.

- **Input**: A set of text documents (e.g., webpages).
- **Output**: A dictionary of unique words and their corresponding counts.

MapReduce API (Simplified)

The MapReduce programming model exposes two primary functions for users to implement:

```
function map (String docname, String doctext) :
    for each word w in doctext:
        emit (w, 1)

function reduce (String word, Iterator partialCounts):
    sum = 0
    for each pc in partialCounts :
        sum += pc
    emit (word, sum)
```

- The ‘map‘ function processes an input Key-Value pair (here, ‘docname‘ and ‘doctext‘) and emits intermediate Key-Value pairs (here, ‘(word, 1)‘ for each word found).
- The ‘reduce‘ function processes a key and an iterator over all values associated with that key (here, ‘word‘ and ‘partialCounts‘), aggregates them, and emits the final output Key-Value pairs (here, ‘(word, sum)‘).

How MapReduce Works: Parallel Flow of Control and Data

The execution of a MapReduce job involves several stages managed by the framework:

1. **Input**: The raw data (e.g., a large text file or collection of files).
2. **Splitting**: The input data is logically split into smaller chunks (e.g., typically 128MB blocks in HDFS) by the MapReduce framework. Each split is then processed by a ‘Map‘ task.
3. **Mapping**: Multiple ‘Map‘ tasks run in parallel across the cluster. Each ‘Map‘ task applies the user-defined ‘map‘ function to its assigned input split, generating intermediate Key-Value pairs.
4. **Shuffling (Shuffle and Sort)**: The intermediate Key-Value pairs emitted by all ‘Map‘ tasks are grouped by key. All values for a given key are sent to a single ‘Reduce‘ task. This phase typically involves sorting the intermediate keys.
5. **Reducing**: Multiple ‘Reduce‘ tasks run in parallel. Each ‘Reduce‘ task receives all intermediate values for a subset of keys, applies the user-defined ‘reduce‘ function, and produces the final output.
6. **Final Result**: The outputs from all ‘Reduce‘ tasks are collected and stored, typically back into a distributed file system like HDFS.

This entire process is designed to handle fault tolerance and scalability automatically, abstracting these complexities from the programmer.