Vraj Ketankumar Shah (vs921) **Ayush Narendra Dodia (and179)** **Nithik Hemant Pandya (nhp46)**

DIS Project Proposal:

Distributed Billing Aggregation: A MapReduce Approach Using Apache Spark on Kubernetes

1. Technical Goal:

The primary goal of this project is to develop a lightweight microservice architecture that efficiently handles billing data processing using a custom MapReduce algorithm to process API logs and generate detailed billing statements. Our main focus is on designing and implementing this algorithm for distributed data aggregation using Apache Spark. If time permits, we plan to integrate the backend architecture as a microservice using FastAPI and deploy it on Kubernetes for load balancing.

By the end of the semester, we aim to have a fully functional distributed system that processes API log data via Apache Spark's MapReduce framework, deployed on a Kubernetes cluster. Specifically, our target is to ingest at least 100,000 API log records stored in PostgreSQL and aggregate them into billing cycles—each defined by every five API calls—with an average processing time of under 60 seconds per cycle. Additionally, we will have automatic recovery mechanisms in place to ensure that the system will still sustain at least 95% task completion rate even when individual worker nodes are intentionally taken offline during stress testing so demonstrating both the fault tolerance and scalability of our solution.

2. Prior Work:

Existing research on microservices and MapReduce algorithms primarily focuses on data processing in large-scale cloud environments. While Hadoop-based systems are common for implementing MapReduce, our project focuses on a more lightweight approach using Apache Spark for better performance and scalability.

In addition, most microservice architectures tend to be complex, while our approach aims to simplify the system while maintaining scalability and fault tolerance.

3. Novelty and Differentiation:

Our project takes its inspiration from Google's original MapReduce algorithm, which broke down large-scale data processing into simple, clear steps: mapping individual data entries into key-value pairs and then reducing these pairs to summarize the results. We apply the same idea to our billing requirements by processing API log entries. In our system, each log entry is transformed into a pair consisting of a user ID and a billing cycle number. Then, in the reduce phase, these pairs are grouped together to add up usage values and calculate the total cost for each cycle.

What makes our work unique is that we are writing our own MapReduce-style algorithm tailored specifically for our billing data. Instead of using a pre-made solution, we have developed custom map and reduce functions that handle our data exactly the way we need. Our

map function reads each API log and assigns it to a billing cycle, while our reduce function groups these entries and sums up the usage before applying a cost rate.

This hands-on approach not only builds on the fundamental ideas of the original MapReduce but also gives us full control over how the data is processed for our specific billing needs. Also our approach is lightweight compared to traditional Hadoop implementations, making it more efficient and suitable for microservice architectures.

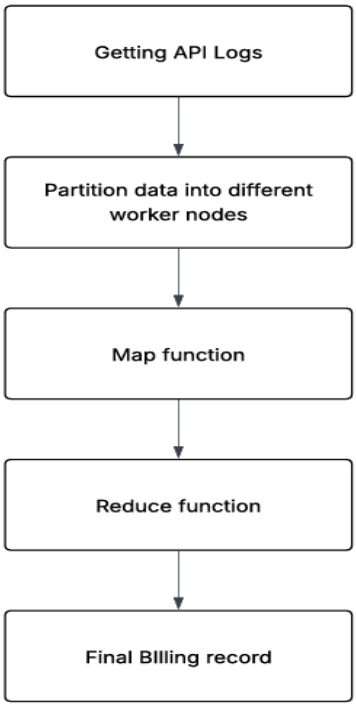
4. Technical Approach:

The project will consist of the following components:

MapReduce Algorithm Design:

- The algorithm will process API logs to compute billing data.
- Implemented using PySpark to leverage Spark's parallel processing.

Flowchart of the MapReduce Process:



Project Workflow

Implementation_Details:
API logs are ingested from PostgreSQL using Apache Spark's JDBC connector and partitioned across the cluster for parallel processing on a Kubernetes-managed Spark cluster running in Docker containers. During the map phase, PySpark transforms each log entry into a composite key-value pair—where the key is a combination of the user ID and billing cycle and the value is the usage metric—using Python lambda functions and Spark's RDD/DataFrame API.

Spark then automatically shuffles and sorts the intermediate data to group all records with the same key, which are subsequently aggregated in the reduce phase via reduceByKey operations that sum the usage values and calculate total costs based on a predefined billing rate. Finally, the aggregated billing records are written back to PostgreSQL using Spark's JDBC writer, completing the MapReduce pipeline.

Technology Used

Technology	Purpose	Usage
Python & PySpark	Implement Map and Reduce functions for billing data processing.	Uses Python's lambda functions and Spark's parallel processing to compute usage and billing data.
Apache Spark	Enables distributed data processing.	Processes API logs in parallel using a MapReduce-inspired approach.
PostgreSQL	Acts as the data store for raw API logs and computed billing data.	Stores input logs and final billing records, accessed via Spark's JDBC connector.
Docker	Containerizes the application components for consistency and portability.	Packages the Spark jobs and PostgreSQL instance for easier deployment and runtime.
Kubernetes	Orchestrates the containerized Spark cluster for scalability and fault tolerance.	Manages Spark master and worker nodes, ensuring load balancing and resilience.
JDBC Connector	Facilitates interaction between Spark and PostgreSQL.	Reads logs from PostgreSQL and writes processed billing data back to the database.
Job Scheduler	Automates the periodic execution of the data processing pipeline.	Uses cron jobs or Kubernetes CronJobs to schedule the MapReduce job at specified intervals.

5. Evaluation Metrics:

- **Performance & Scalability:**

Our system's performance will be evaluated by testing how well the distributed Spark cluster and the FastAPI microservice scale under increased loads. We will simulate varying volumes of API log data and monitor how efficiently the Spark job processes these logs to calculate billing, measuring metrics such as processing time and throughput. The goal is ensuring that as the data volume increases, system distributes the workload across multiple worker nodes without much delays. The Kubernetes architecture will also tested to verify that additional pods are deployed automatically under high load, ensuring consistent performance even during peak usage. This will help us confirm that the system is robust enough to handle real-world scaling requirements.

- **Fault Tolerance:**

Ensuring fault tolerance and resilience is one of the major things we'll be testing for. We'll add failures by design in certain components, like failing worker nodes in Spark, to observe

how the system behaves and recovers from them. We'll focus on making sure that our system gracefully recovers from these outages, redirecting tasks to healthy nodes without losing data or performance. Kubernetes' built-in features such as pod auto-restart and Spark's fault tolerance features are anticipated to reduce downtime to a minimum. In these tests, we will evaluate the system's overall reliability so that under unfavorable conditions, the system does not fail and continues to process data correctly.

6. Potential Risks:

We aren't sure what the potential risks are, but as the volume of data increases, we may face delays in processing data, which ultimately slows down spark jobs. We are also not sure how Kubernetes responds to a sudden rise in demands. While doing the project's performance evaluation, we will understand these risks better.

7. Future Work:

If time permits, we would like to include an extension to our project by implementing a small backend microservice using FastAPI. This microservice level will serve as a user-interaction interface for authentication, file uploads, and API log collection. Including this feature enables us to demonstrate how an API microservice can seamlessly integrate with our proprietary MapReduce pipeline in a full end-to-end system for billing and processing.

This not only adds an extra layer to our project but also shows the integration of distributed data processing with modern microservice design standards, further enhancing the overall usability and functionality of our solution.

8. Conclusion

In conclusion, our project idea is implementing a distributed data processing system taking inspiration from Google's MapReduce algorithm. We will be ingesting API logs from PostgreSQL and partition the workload across a Kubernetes-managed Apache Spark cluster, we efficiently process data in parallel. Our custom map and reduce functions transform individual log entries into key-value pairs and aggregate them to compute billing cycles accurately. This project shows the effectiveness of parallel processing for real-time billing analytics as well as highlights the power of combining Spark's distributed computing capabilities with Kubernetes' load balancing and fault tolerance to deliver a scalable and resilient solution.

References

[1] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1 (January 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[2] Zaharia, Matei & Chowdhury, Mosharaf & Franklin, Michael & Shenker, Scott & Stoica, Ion. (2010). Spark: Cluster Computing with Working Sets. Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. 10. 10-10.

[3] Changpeng, Zhu & Han, Bo & Zhao, Yinliang. (2022). A comparative performance study of spark on kubernetes. The Journal of Supercomputing. 78. 10.1007/s11227-022-04381-y.