**Cloud Architecture**

The cloud architecture for our project is shown in figure XX below:

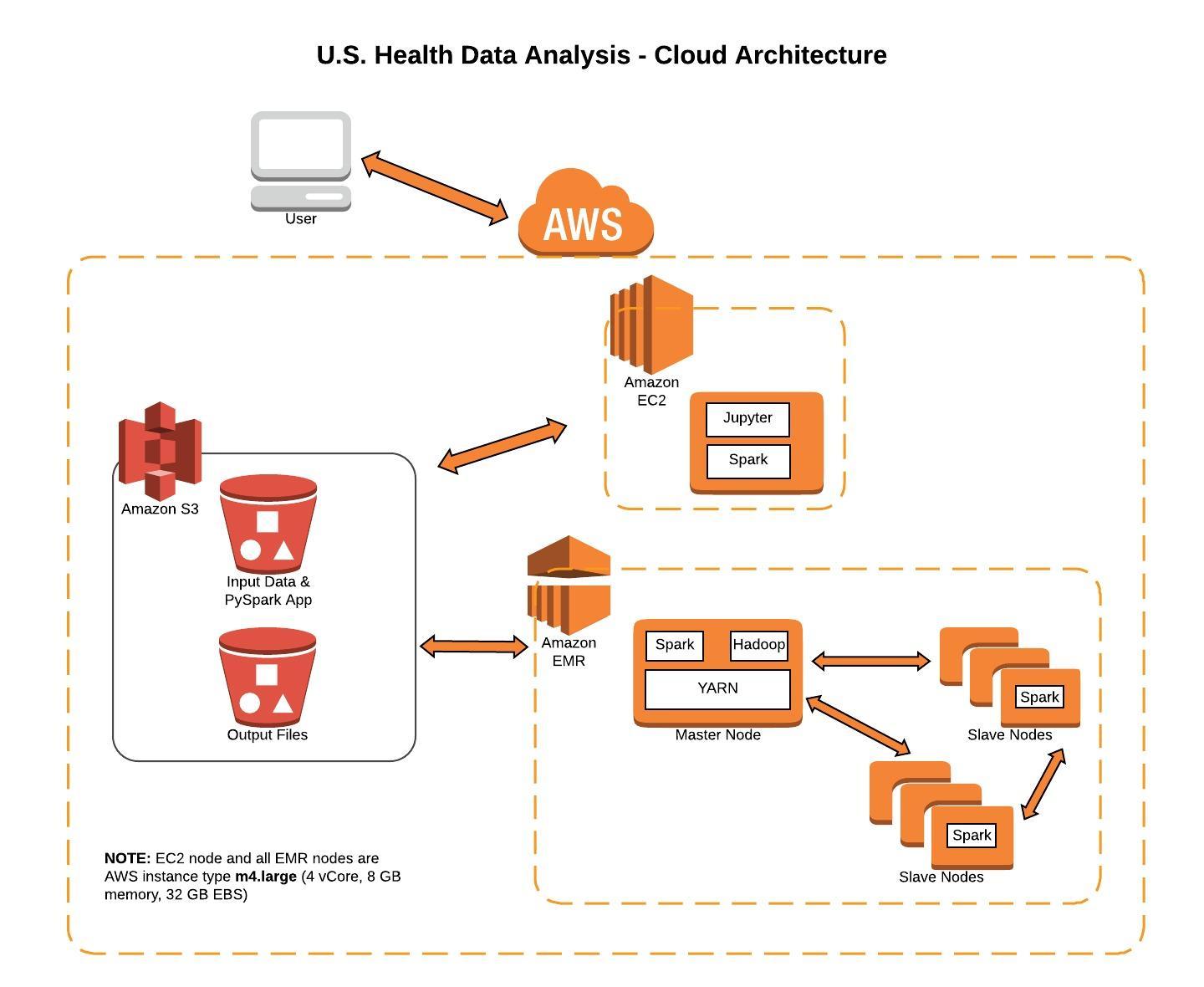


Figure XX

The system architecture consists of three components:

1. AWS S3 Data Store
2. AWS EMR Cluster
3. AWS EC2 Node

The PySpark application was divided into two parts: 1) the data transform part that ran on the EMR cluster and 2) the data visualization analysis part that ran on the EC2 node. The input and output files for the application resided on an AWS S3 bucket.

**AWS S3 Data Store**

The input and output files shared a common Simple Storage Service (S3) bucket. The PySpark data transform application also resided on the same S3 bucket, so that no local EBS storage was utilized either on the EC2 node or EMR cluster.

**AWS EMR Cluster**

The PySpark data transform application ran as a standalone step on the EMR master node. The number of slave nodes varied from 0 to 16 for our scalability testing. All EMR nodes were AWS instance type **m4.large** with the following hardware characteristics: 4 vCore, 8 GiB memory, 32 GB EBS storage. The default Ubuntu AMI was installed on each node, along with Hadoop 2.8.3, Spark 2.3.0, and Python 3.6.

The PySpark application ran as a YARN (Yet Another Resource Negotiator) client and YARN and Spark handled the distribution of work over the EMR cluster. Spark, running on EMR, used the EMRFS (EMR File System), to directly access the input data stored on S3 and to write the output data to S3.

**AWS EC2 Node**

The Elastic Compute Cloud (EC2) node ran Jupyter Notebooks using Python/PySpark. The EC2 node accessed the output data files on S3 that were generated by the EMR cluster to perform data analysis using Spark MLib and data visualization using the Seaborn library.

The EC2 Node utilized the same node type as EMR, i.e. the AWS instance type **m4.large**. The Linux system configuration on the EC2 node was the Amazon Machine Instance (AMI) *Deep Learning AMI (Ubuntu) Version 8.0.*

The EC2 node could be accessed by any browser on the public internet that had access to the correct security key.

**Scalability Testing Results**

We ran the PySpark application on the master cluster node with N Cities = (500, 5000, 50000) and number of slave cluster nodes = (0, 1, 2, 4, 8, 16). The results are shown in figure XX below:

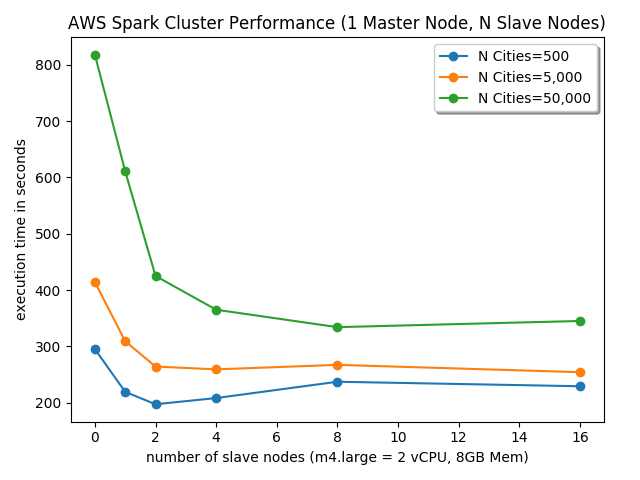


Figure XX

For each of our test sets of cities, we calculated the optimal number of slave nodes and the execution time improvement for that configuration. This data is shown in the table below:

|  |  |  |
| --- | --- | --- |
| **N Cities** | **Optimal Cluster Slaves Nodes** | **Execution time improvement** |
| 500 | 2 | 33% |
| 5,000 | 4 | 37% |
| 50,000 | 8 | 59% |

As an additional test, we ran a 500,000 cities instance. We did not include this in our results, as there are not 500,000 cities in the world, but we used it as a scaling check for optimal node calculation. It resulted in the same number of optimal slave nodes (8) as the 50,000 city test.

The data from the scalability tests are included in **[REFERENCE APPENDIX]**