May 9th, 2022

System Investigation Project – Part 2

DATA/CSE 514

THE OPTIMIZERS GROUP

SRAVAN HANDE
THARUN KUMAR REDDY KARASANI
ROHIT LOKWANI

Table of Contents

System Summary **2**

System Main Features..... 3

Data model..... 4

User Interface..... 6

Indexes..... 7

Consistency..... 8

Scalability and Replication..... 8

System Installation **8**

System Design **14**

Distributed Tensors..... 14

Infrastructure (CPU/GPU) 15

Data model decisions..... 17

Comparable variants..... 17

Data processing and structuring..... 18

Overall approach..... 18

Final Decision **19**

References **19**

1. System Summary

SystemDS (formerly known as SystemML was developed by IBM and then open-sourced in 2015. SystemDS is an open source ML system which supports end-to-end data science activities right from data cleaning, integration, feature engineering, model training, deployment and serving. It is platform and infrastructure independent, essentially running over CPUs, GPUs in local, distributed environments. It provides support for declarative languages like R and scripting languages like Python, compiles the code and can then run on local CPU/GPU, hybrid or distributed environments on Apache Spark. In contrast to existing systems - that either provide homogeneous tensors or 2D Datasets - and to serve the entire data science lifecycle, the underlying data model are DataTensors, i.e., tensors (multi-dimensional arrays) whose first dimension may have a heterogeneous and nested schema. Overall, it's a scalable Machine Learning system with distinguishing characteristics as follows:

1. Algorithm customizability via R-like and Python-like languages.
2. Multiple execution modes, including Spark MLContext, Spark Batch, Standalone, and JMLC.
3. Automatic optimization based on data and cluster characteristics to ensure both efficiency and scalability.

The latest version of SystemDS supports: Java 8+, Python 3.5+, Hadoop 3.x, and Spark 3.x, Nvidia CUDA 10.2 (CuDNN 7.x) Intel MKL (<=2019.x).

For this document, we stick to naming the system SystemDS apart from diagrams, which could be referenced from old documents, hence the name would be SystemML. We will be covering some of the key features and/or characteristics of SystemDS as follows.

1.1.1 Efficiency and Scalability of Database System

Generally, the experimentation and analysis of data is done by data scientists on a sample of data to develop initial prototypes and get the intuition. But when the data increases, code optimization, batch processing and distributed architectures come into picture.

In the following three diagrams, we showcase, how SystemDS can be helpful in improving workflows. In general, when a Data Scientist starts working with data which is small in size, the following is the workflow, where the model/scripts are written in Python/R on a personal computer and then the results are shared.

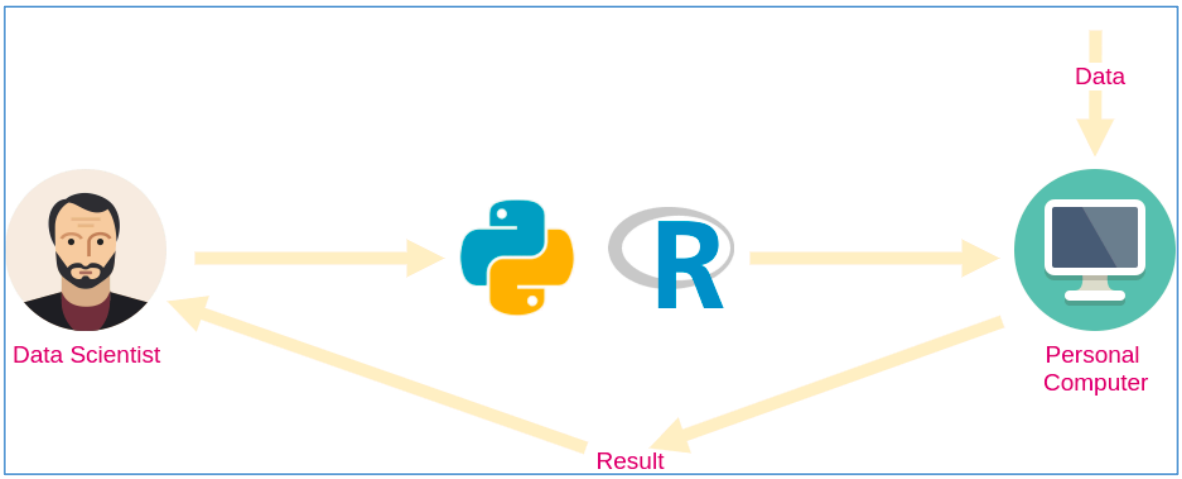


Figure 1.1: Local development workflow

Now, if the data is increased, the flow needs to change from local to distributed, this Python/R code is converted to Scala code which runs in distributed environments on Spark by a System Programmer. The results from these are then shared with the Data Scientist. The downside of this approach is that it could induce programming errors and doubles the manual effort.

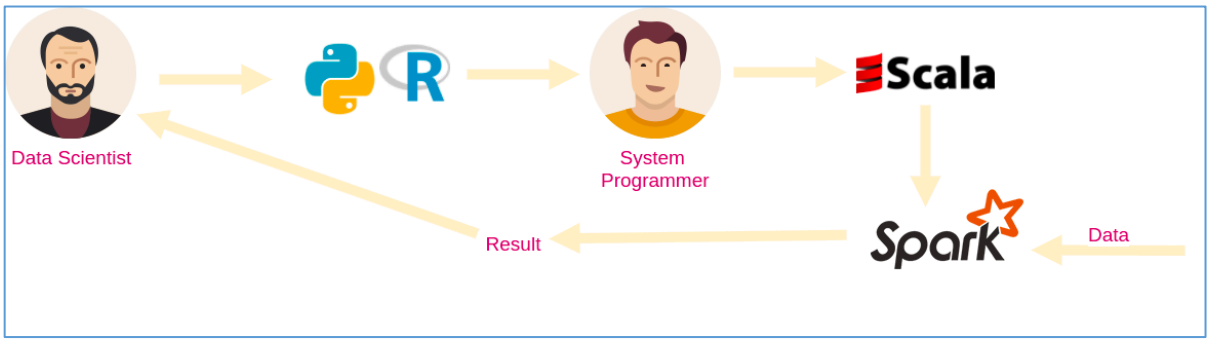


Figure 1.2: Scaling to big data workflow

SystemDS helps in overcoming these drawbacks, where the data scientist directly writes the DML scripts which are read by SystemML and run on Spark back-end. This helps in scalable, distributed work, reduces the programming effort and makes the entire process platform and infrastructure independent.

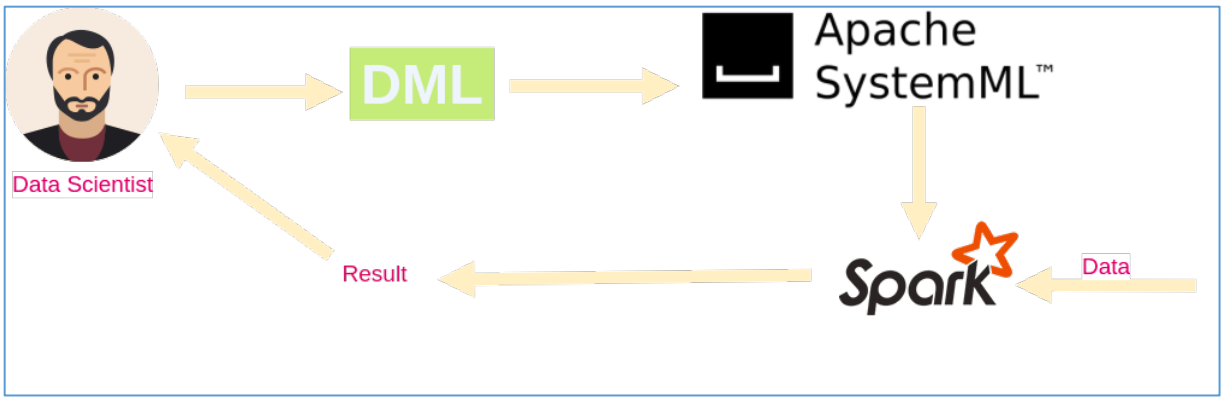


Figure 1.3: Local development workflow

1.1.2 Platform Independence

The following diagram shows high-level SystemDS architecture, where the DML scripts pass the script to compiler/interpreters and runtime and then run on in-memory single node or hadoop or spark clusters. It is important to note that the SystemML automatically determines the processing efficiency for a single node or multiple clusters. This further reduces the programming effort of the data scientist and systems programmer. It also gives the data architect/engineers the ability to use SystemML with flexibility of the underlying tools.

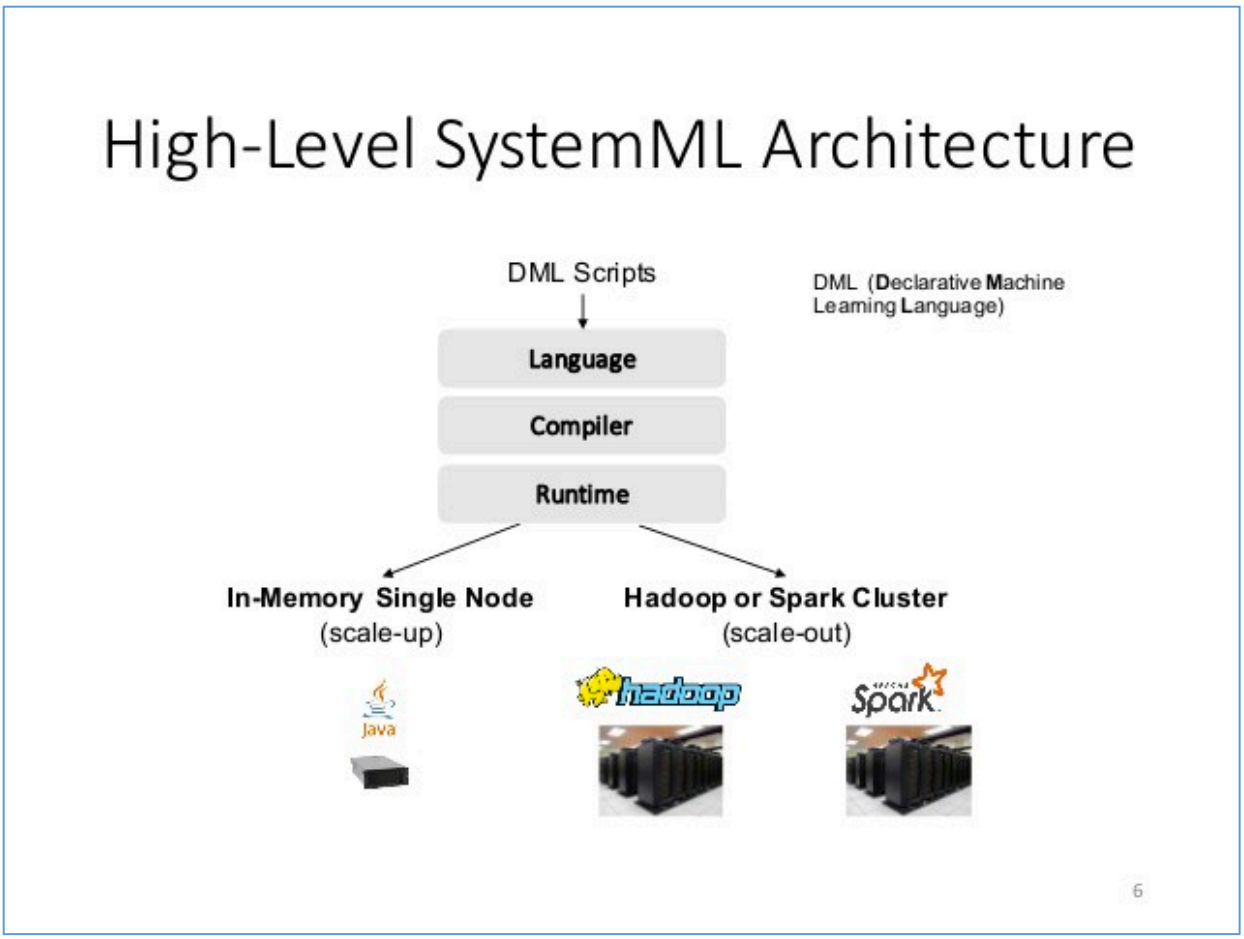


Figure 1.4: High-level overview of SystemDS architecture

1.1.3 Ease of programming and language flexibility

SystemDS provides the flexibility for the programmer to use Python or R like languages. The internal compilation is abstracted from the user. This abstraction also involves having simple functions which perform complex algebra and calculations in the background. The following figure gives an example of the same.

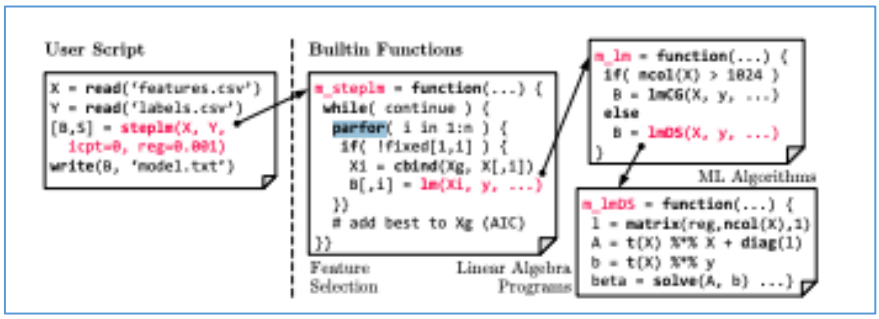


Figure 1.5: Conversion of DML scripts to algebraic operations

Some of the other key features include automatic code optimization and working with multiple execution nodes depending on Spark MLContext and others. We cover the details and further features in the rest of the document.

1.2 Data Model

Our data model is similar to a relational model in SQL. Each of the primary entities being represented as separate dataframes. The relationships depend on one of the keys in each of the entities. Since Spark 2.0.0 a DataFrame is a *Dataset* organized into named columns. It is conceptually equivalent to a table in a relational database or a DataFrame in R/Python, but with richer underlying optimizations [3].

These dataframes are converted into Resilient Distributed Datasets (RDDs). RDDs are the primary user-facing API in Spark. At the core, an RDD is an immutable distributed collection of elements of your data, partitioned across nodes in your cluster that can be operated in parallel with a low-level API that offers transformations and actions. We perform all the operations and transformation on these RDDs. Spark DataFrame doesn't have methods like map(), mapPartitions() and partitionBy() instead they are available on RDD hence we need to convert DataFrame to RDD and back to DataFrame.

The major benefits of RDDs include immutability, Fault tolerance, and ease of partitioning data on across multiple execution nodes. It also provides persistence of results to make process optimized and coarse-grained operations [1]. Also, our data has text streams RDDs that would help us process and analyze them in future.

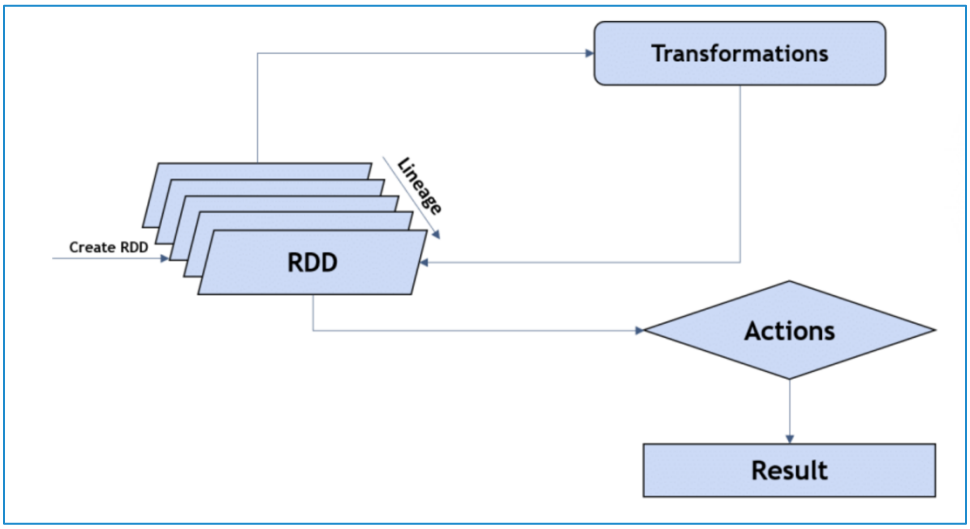


Figure 1.5: Working with RDDs (Data model workflow)

The logical separation of RDDs in our system is more about having neighborhoods, listings, and calendars dataframes as separate RDDs. These RDDs can then be used to perform transformations and actions like grouping, aggregation, mapping and reduction depending on the queries that need to be executed for our analysis.

Following is the example of loading our Airbnb Popularity Analysis data as a dataframe.

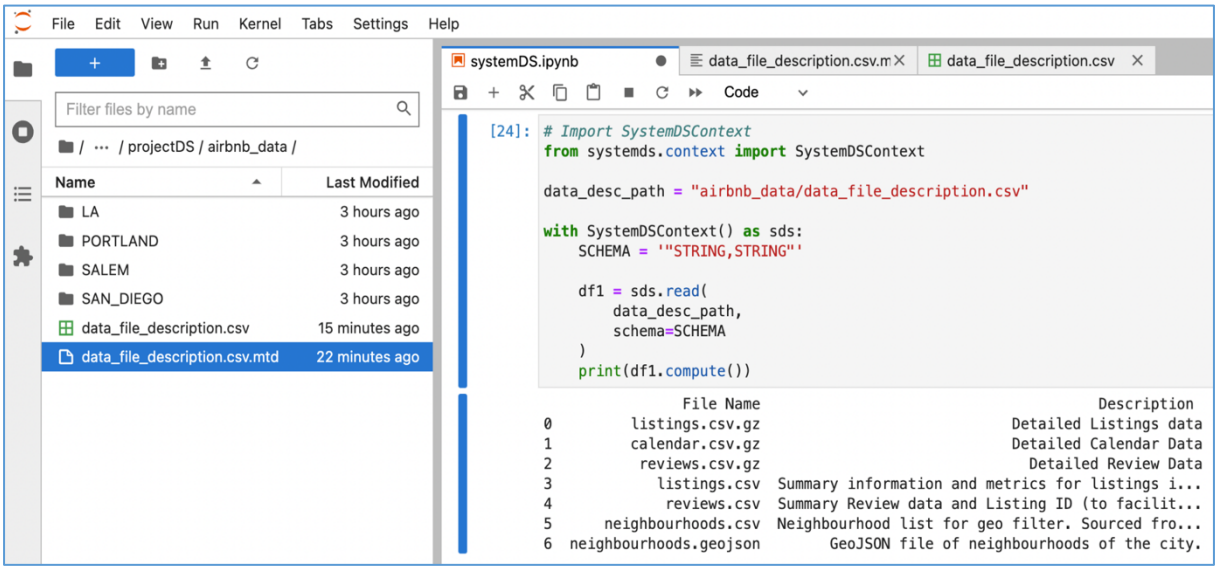


Figure 1.6: Data Loading example

These dataframes can be converted to rdds using basic syntax: *df_rdd=df1.rdd*

1.3 User Interface

SystemDS provides the users with APIs in Java and Python. Users can script the SystemDS instructions using either of the language apis along with .dml configuration files. The DML files contains R-like syntax instructions that are read through Java/Python apis. DML stands for Data Manipulation Language. Below is a sample DML script that can be used to impute missing values in data columns.

{

"impute":

[{ "name": "age", "method": "global_mean" }

,{ "name": "workclass" , "method": "global_mode" }

,{ "name": "relationship" , "method": "global_mode" }

,{ "name": "hours-per-week", "method": "global_mean" }

,{ "name": "native-country", "method": "global_mode" }

]

}

The above script instructs SystemDS to apply the imputation technique provided in the method parameter and to the columns provided as the name parameter. SystemDS documentation provides us a simple guide to convert existing R-Scripts to DML instructions here - https://systemds.apache.org/docs/2.2.1/site/dml-vs-r-guide.html.

1.4 Indexes

SystemDS operates on two primary datamodels - matrices and dataframes. Both the data models come with default row and column indexing. However, it is not possible to create index on a desired column within the data. Data from these models can be accessed using matrix indexing and data frame indexing respectively. The indexes in SystemDS start from 1. Below are the examples for each type of indexing.

Index operations on a matrix X

```
X[1,4] # access cell in row 1, column 4 of matrix X
X[i,j] # access cell in row i, column j of X.
X[1,]  # access the 1st row of X
X[,2]  # access the 2nd column of X
X[,]   # access all rows and columns of X
```

Index operations on the data frames A, F

```
# = [right indexing]
A = read("inputA", data_type="frame", rows=10, cols=10, format="binary")
B = A[4:5, 6:7]
C = A[1, ]
D = A[, 3]
E = A[, 1:2]

# [left indexing] =
F = read("inputF", data_type="frame", rows=10, cols=10, format="binary")
F[4:5, 6:7] = B
F[1, ] = C
F[, 3] = D
F[, 1:2] = E
```

1.5 Consistency

Since RDDs are immutable, which means unchangeable over time. That property helps to maintain consistency when we perform further computations. As we cannot make any change in RDD once created, it can only get transformed into new RDDs. This is possible through its transformation processes. Furthermore, this applies when you partition your dataset into different shards, hence the transformations are applied to each of the worker RDDs.

While working on any node, if we lose any RDD itself recovers itself. When we apply different transformations on RDDs, it creates a logical execution plan. The logical execution plan is known as lineage graph. Consequently, we may lose RDD as if any fault arises in the machine. So, by applying the same computation on that node of the lineage graph, we can recover our same dataset again. In fact, this process enhances its property of Fault Tolerance, Consistency and Reliability.

1.6 Scalability and Replication

Each dataset is logically partitioned and distributed across nodes over the cluster. They are just partitioned to enhance the processing, not divided internally. This arrangement of partitions provides parallelism. This parallelism can help maintain the performance of the system. Using SystemML on top of this allows to have multiple execution nodes using Spark ML context. So, even if the scalability of RDDs becomes limited at some point, SystemML makes up for the compute scaling execution nodes. We are currently trying to understand how logical partitions work and if we can enforce rules to horizontally shard the data using RDDs and how does it impact the performance of the system as a whole. We plan to perform a couple of experiments to determine that.

This method of lineage recomputation removes the need for costly data replication strategies used by other methods for abstracting in-memory storage across a compute cluster. However, if the lineage chain reaches a large enough size, users can manually flag a specific RDD to be check-pointed. Check-pointed RDDs are written to disk or HDFS to avoid the recomputation of long lineage chains. RDDs are not for storing or archiving final result data; this is still handled via HDFS or other file system[2]. An example of replication in this system would be as simple as duplicating a RDD with basic or no transformations.

2. System Installation

SystemDS is build on top of java and hence needs JDK installations. Currently SystemDS is compatible only with JDK8. In this section we will go through the step by step process to install systemDS on a mac system along with the python API for our implementation purpose.

1. Check for any existing incompatible Java installations and uninstall it. In case multiple java installations are required in your system proceed to next step without uninstalling existing versions.

2. Install JDK 8. Here we will install openjdk 8 version using homebrew tool

```
brew install --cask adoptopenjdk/openjdk/adoptopenjdk8
```

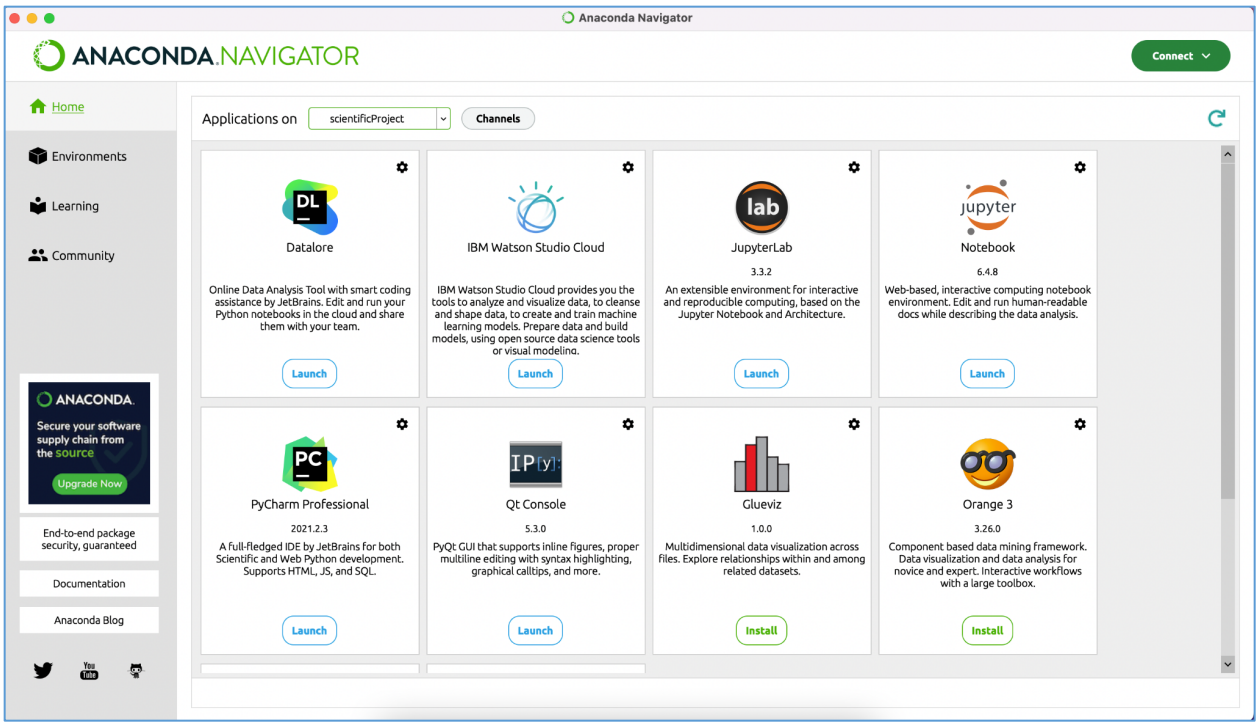
3. Set the PATH variable to point to JDK 8 and remove the other java versions from the PATH

```
PATH=/Library/Java/JavaVirtualMachines/adoptopenjdk-8.jdk/Contents/Home:$PATH
```

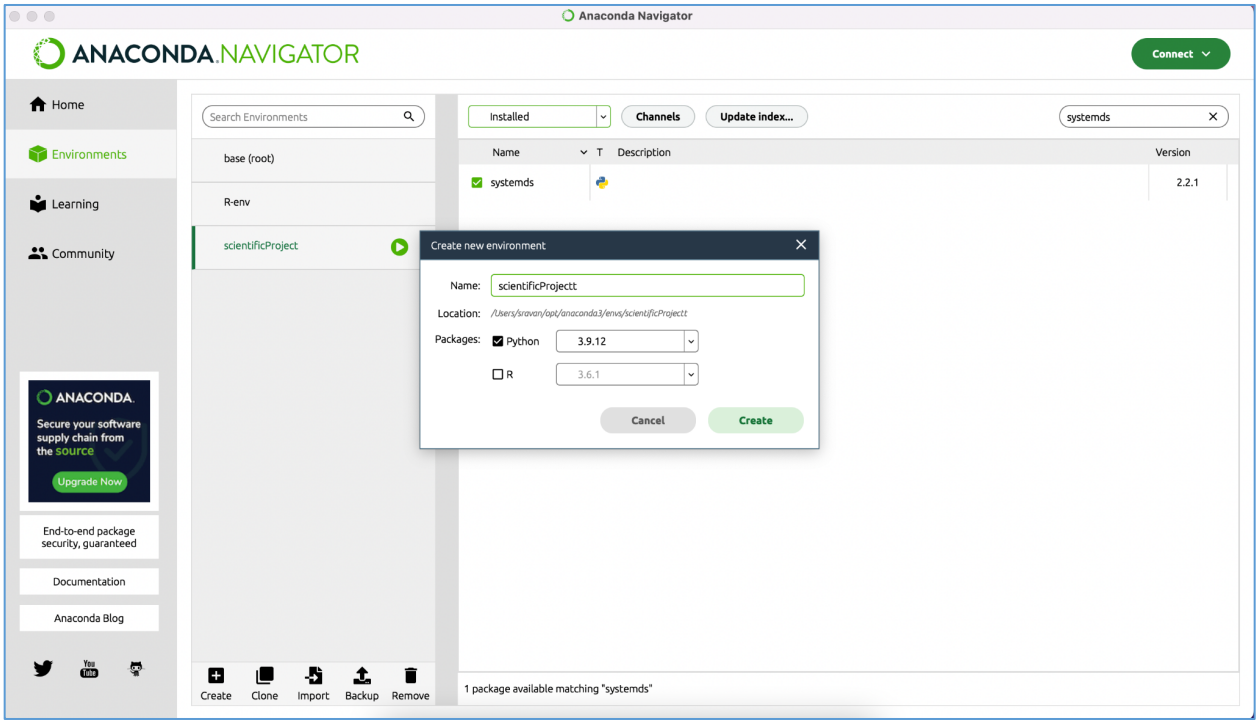
Once the installation and PATH variable setup is complete, check your installation using the version command and your output should look like below,

```
(base) sravan@Sravans-MacBook-Air ~ % java -version
openjdk version "1.8.0_292"
OpenJDK Runtime Environment (AdoptOpenJDK)(build 1.8.0_292-b10)
OpenJDK 64-Bit Server VM (AdoptOpenJDK)(build 25.292-b10, mixed mode)
```

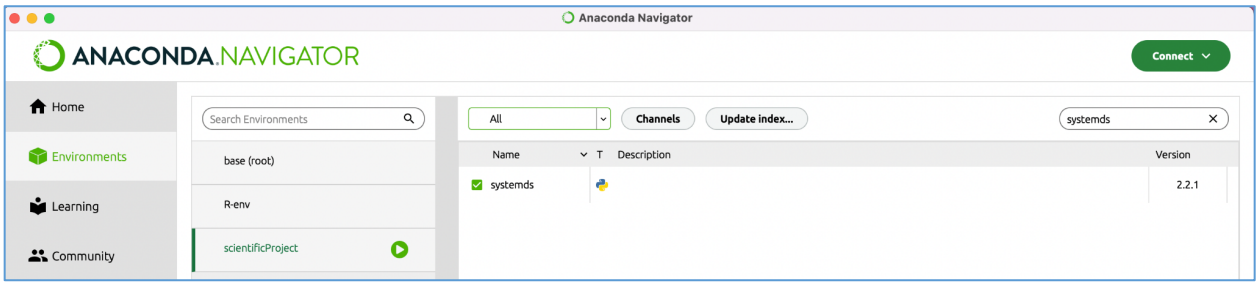
4. Installing Python: We will be using Anaconda distribution of python, which comes with a wide range of libraries prepackaged with the installation. SystemDS requires Python 3.5+ and we will install the latest stable version of anaconda python i.e Python 3.9. You can download the .dmg file from here (https://www.anaconda.com/products/distribution) and execute it to install. Once complete you should be able to run the anaconda navigator and spawn a jupyter notebook from these as shown in below image.

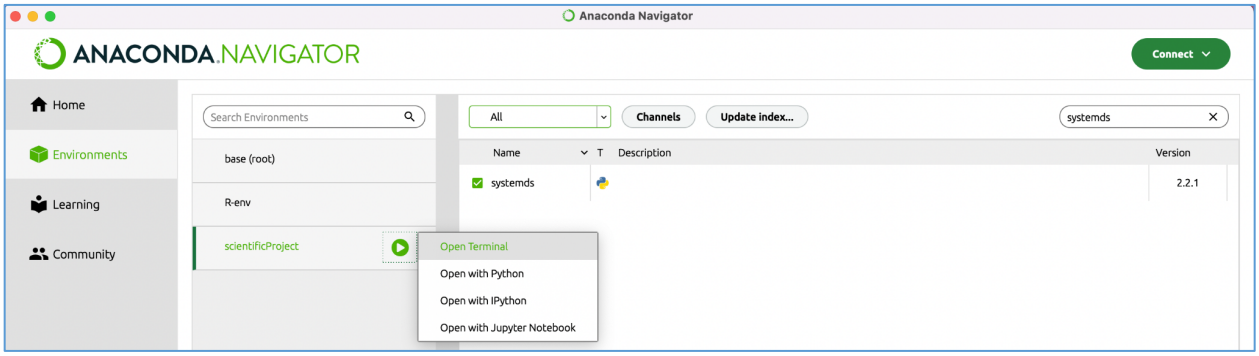


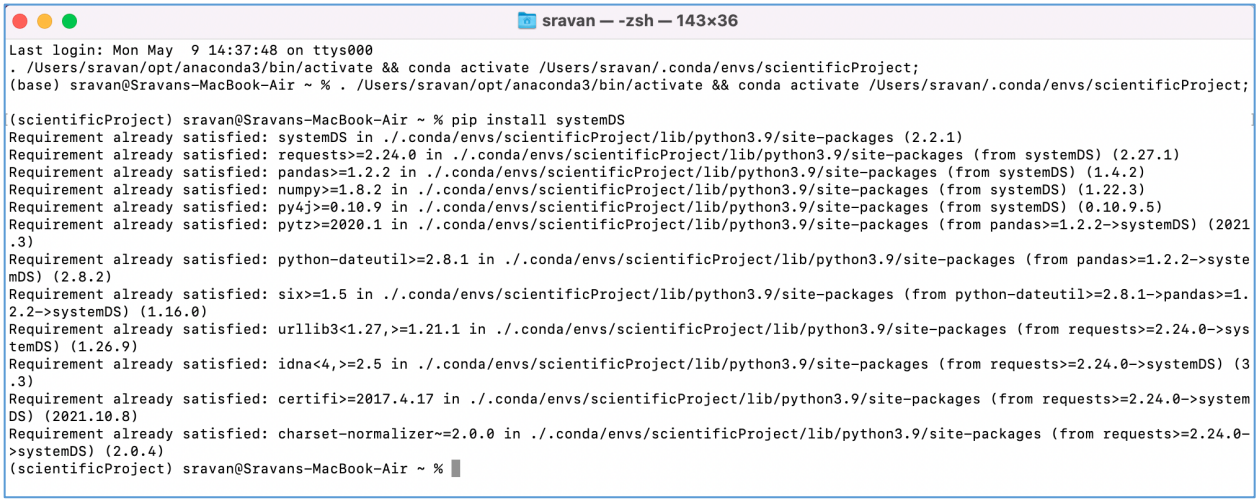
5. Creating a virtual environment to install libraries specific to this project. Navigate to Environments on the left panel and click create icon on the bottom panel. Check Python and proceed to create as below



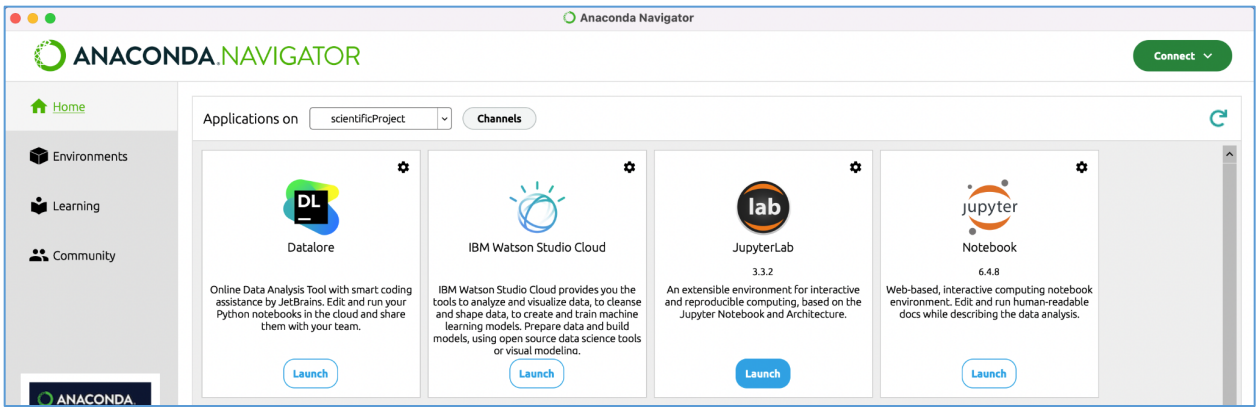
6. Installing SystemDS library. You can search in the searchbox on the right top corner or open a terminal and execute the command to install the library.



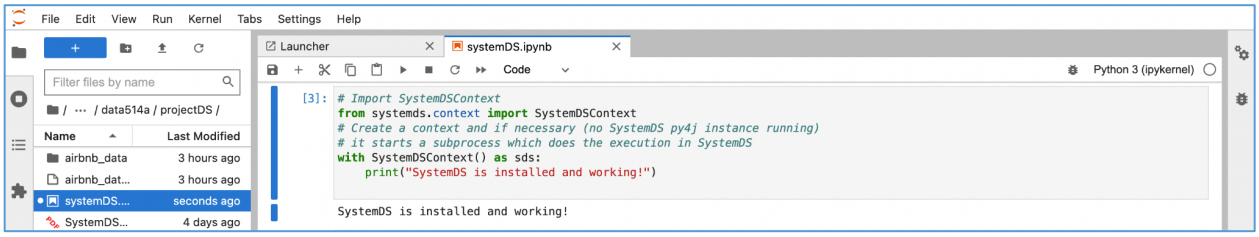




7. Navigate to home, switch the environment to newly created environment and start the Jupyter lab or Jupyter notebook server.

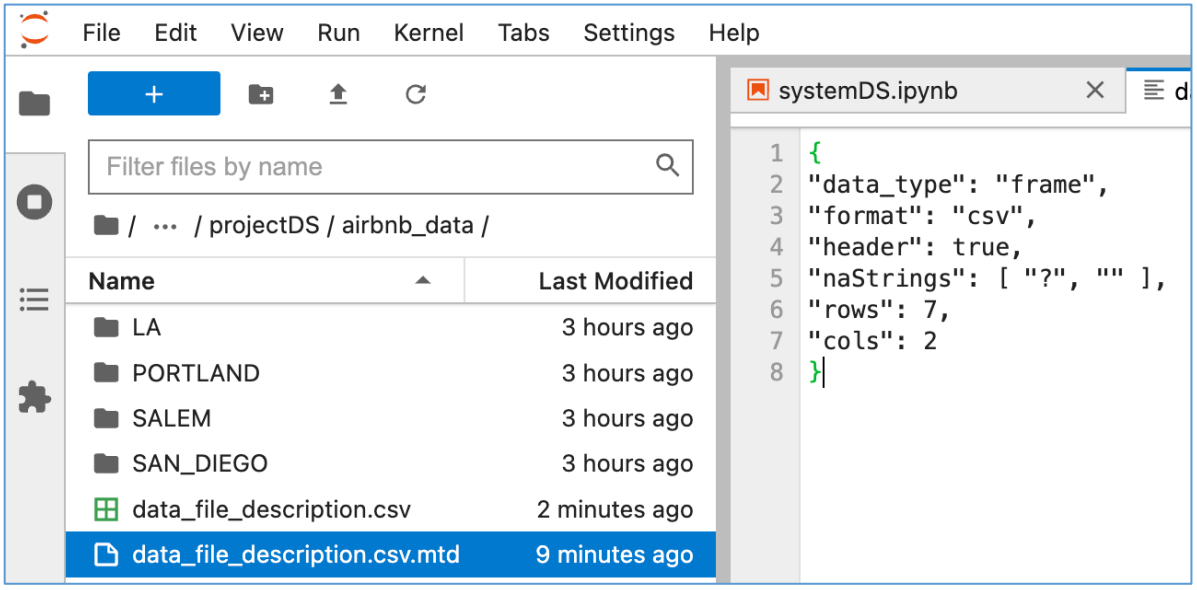


8. Validating installation with a simple Hello SystemDS script.

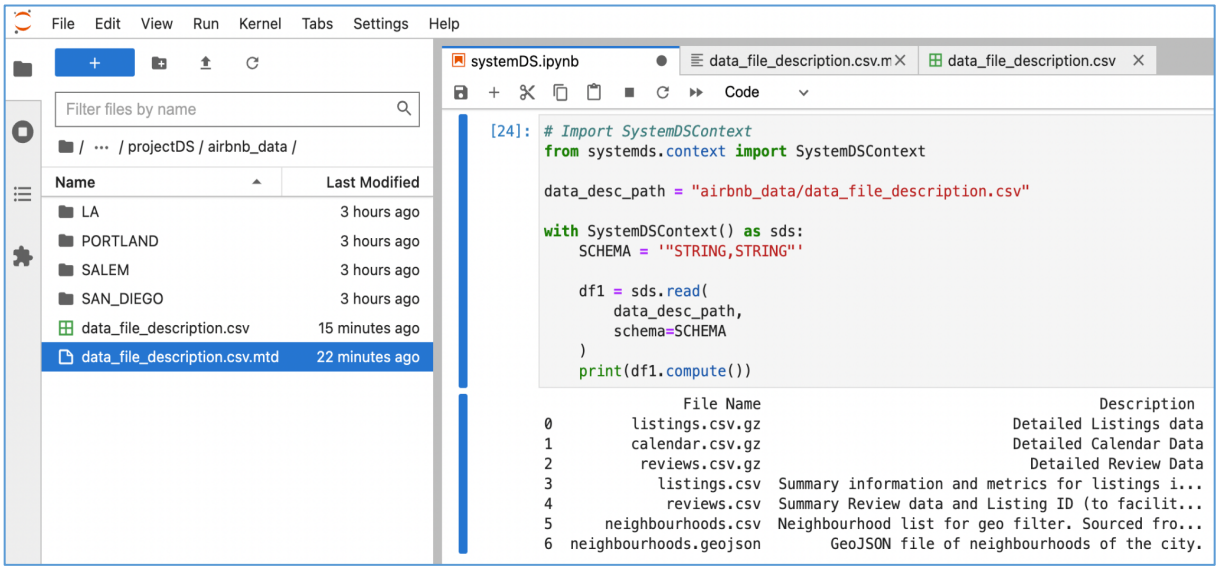


9. Reading data description from a CSV file.

a. Create a metadata(.mtd) file to mention the properties of the CSV file and place it in the exact location as the CSV file.



b. Reading the data into a frame and printing the dataframe

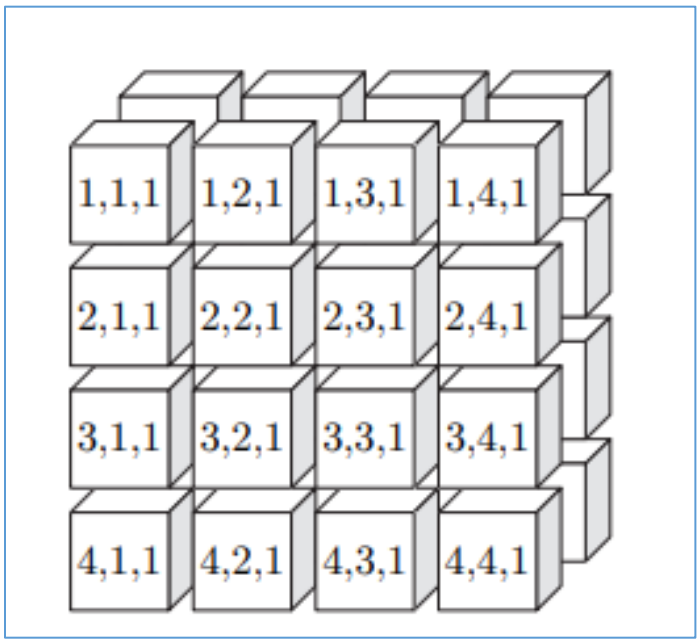


3. System Design

SystemDS, in consideration of the diversity of ML algorithms and applications, supports multiple back-ends. It includes local CPU and GPU instructions and distributed Spark instructions. In addition, it also includes the federated instructions, which support the federated ML architecture listed below. These instructions rely on a common TensorBlock operation library, which extends SystemDS from numeric matrices to heterogeneous, multi-dimensional arrays. Below is the list of back-end systems supported by SystemDS and their respective representation of the data model in the data store.

3.1 Spark Architecture - Distributed Tensors

The SystemDS distributed tensor representation is a **Spark RDD** which is a distributed collection of tensor indexes and fixed-size, independently encoded blocks (PairRDD). Each block size is configured to provide a good balance between block overheads and moderate block sizes. The effective selection of block sizes simplifies join processing because blocks are always aligned. However, fixed-size blocking for n-dimensional data is challenging. SystemDS uses a scheme of exponentially decreasing block sizes (1024^2 , 128^3 , 32^4 , 16^5 , 8^6). This dynamic representation of data in blocks helps in quick transformation/conversion tasks.



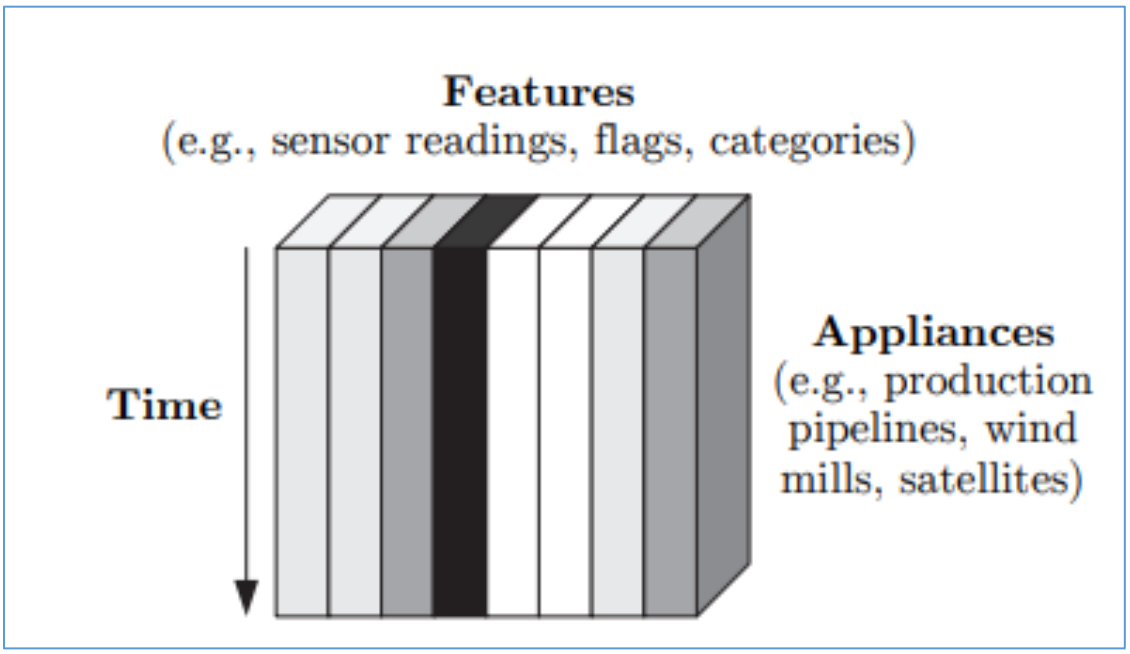
3.2 Local CPU and GPU - Basic and Data Tensors

1. Basic Tensor Blocks

A basic tensor is a linearized, multi-dimensional array of a single type. Hence, it is a homogeneous dense or sparse tensor block with associated operations based on its sparsity.

2. Data Tensor Blocks

A data tensor block has a schema on the second dimension. It is nothing but multiple basic tensor blocks for each schema. Hence, it is heterogeneous and holds various types of features. Below is the pictorial representation of the data tensor block.



3. Federated Architecture - Federated Tensors:

A Federated tensor is a metadata object referencing remote distributed tensors discussed above residing in other configured worker nodes. They are used to execute operations in a federated architecture and are the basis for federated learning. The master control program holds the federated tensors and the federated instructions leverage these tensors to push the computation operations to the remote worker nodes. This metadata object also generalizes operator placement to operations over multiple devices (Data residing among CPU and GPU)

Example: Airbnb data - Derived fields creation

The data is first partitioned and federated into multiple worker nodes. The master node first broadcasts the function to the workers, lets them perform a local computation, collects the results, and constructs the output matrix to compute a derived logic on top of the federated data.

3.3 Data Model Decision

We have decided to use the federated architecture with Spark instances running on each node. Below are the key advantages of such a model:

1. **Lazy evaluation**
   The transformation operations are not performed until there is any action operation. Instead, a DAG (Directed Acyclic Graph) is built to execute the operations.
2. **In-memory computation**
   It helps avoid unnecessary IO operations and stores most of the intermittent data within the memory, helping in speeding up the program execution
3. **DAG optimization**
   Along with the lazy evaluation, a directed acyclic graph is built, with each node representing an operation independent of any other downstream operations. This enables the framework to run certain operations in parallel.
4. **Speed of Execution**
   The overall execution speed is cut down drastically with horizontal scaling and the above optimization techniques.
5. **Robustness/Platform-agnostic**
   SystemDS provides us an abstraction using which we can write a single set of operations that can be executed across multiple platforms – federated mode, local CPU, Spark Cluster mode, GPU mode.

3.4 Design Description and Comparable Variants

SystemDS provides us with an API that can be used in various languages like R and Python to customize the algorithm computations for efficient processing. SystemDS uses SystemML's declarative machine learning language to perform aggregations or calculations that use linear algebra. The API can be invoked from the command line or the programming languages through language bindings. Before performing the actual calculations, SystemDS comes with a DAG of all the High-Level Operators (HOPSs) and tries to optimize this to come up with the most efficient order of operation for these statements. After this step, it computes the memory needed for these statements and creates the executable program.

The code turns into a series of code statements called a control program (CP) when this is compiled. The control program is equipped to recompile and restructure some of the instructions, clean the intermediate variables, and even support reads and writes from distributed file systems like S3 or HDFS. This facilitates multiple back-ends and producescomes up with instructions for various platforms like CPU, GPU, and Spark, along with federated instructions that use the common TensorBlock operation library. The advantage of this is that it can be used to perform calculations of heterogeneous matrices/arrays as well. Another advantage is that it holds the entire tensor in one block for local operations, and RDD collections of fixed sizes are used to represent distributed tensors. This gives a consistent way of tackling both local and distributed operations.

The variant we considered was to use GPU instances instead of the current variant that we are using - Spark instances. GPUs are more preferred when the number of operations is high, and the kind of data we deal with is homogenous. GPUs are generally optimized for tasks like image processing and other heavy numerical computations, but data in our use case does not entail these, so we have chosen to go ahead with Spark instances instead of GPU instances.

3.5 Data Preprocessing and Restructuring

The SystemDS supports various operations ranging from data cleaning, data augmentation, and transformation to feature selection, machine learning model execution, and evaluation. Below are the theme of data cleaning and transformation steps we will be performing on the Airbnb dataset with the help of SystemDS API.

1. Data Cleansing

- Impute the missing values in rows
- Convert the feature values into relevant data types

2. Data Filtering

- Filter out the redundant columns based on the scope of the queries
- Filter out the redundant rows based on outliers or data gaps

3. Data pre-processing

- Creation of derived/categorical fields like flags that represent a data property
- Encoding the qualitative predictor variables into dummy variables

3.6 Approach

Below are the high-level steps that we follow in order to restructure the data to fit our model and answer the finalized questions.

Step 1: Read the data file and metadata

Step 2: Setup multiple federated environments

Step 3: Defining the preprocess operations

Step 4: Applying the preprocessing steps

Step 5: Execution of queries

Step 6: Model training and evaluation (Execution performance benchmarking)

3.7 Queries for project implementation

While SystemDS library's main goal is to enable users to write platform agnostic instructions for machine learning applications, it also comes with a wide range of features that are required for data preprocessing and cleaning. As part of the implementation, we aim to focus on writing queries/instructions to load, transform and use the resulting dataset to answer a variety of questions posted as part of the Airbnb dataset.

Along with the 6 queries to answer the questions about datasets, we also aim to produce a simple machine learning model using SystemDS instructions and explore the possibility of training the model on various platforms like local CPU, federated mode and spark platforms.

4. Final Decision

- Final Submission: Implementation
- Revisions: No

5. References

[1] Shashikant Tyagi, Things to know about Spark RDD(2022), https://blog.knoldus.com/things-to-know-about-spark-rdd/

[2] Stephen Bonner (2017), Exploring the Evolution of Big Data Technologies, https://www.sciencedirect.com/science/article/pii/B9780128054673000144

[3] https://systemds.apache.org/