Introduction: The eau2 system is a 3 layer, distributed system that can store and retrieve a high volume of organized data that can be used for big data analysis, whether that be machine learning or data visualization.

Architecture: As mentioned earlier, the eau2 system has three layers. At the bottom, there is a distributed key value store running on each node in the system. Each store has a section of the data, and the stores can communicate with each other if the data that needs to be retrieved is not in the key value store of the current node being queried on. All networking and concurrency is happening in this layer. The next layer contains abstractions such as distributed arrays and data frames which are implemented by calls to the bottom layer, which contains the key value stores that stores the data in a distributed manner. In this layer, there is what is called a KDStore. Each KDStore has its own key value store that it can use to access data that is stored in the system. Finally, we have the application layer, which allows users to write code to access the stored data. We use this layer to demo potential applications such as Trivial, Word Count, and Linus.

Implementation:

Key Classes:

DistDataFrame: This class represents a dataframe that has been distributed across the network. Its data is stored evenly across the nodes. Like a regular dataframe, a distributed dataframe has a schema and columns. Like the distributed dataframe itself, these are also stored across the network. Finally, a distributed dataframe has a kv store that is responsible for performing the distribution and a boolean field called locked_. This ensures that, for network operations, the dataframe is fully stored before we start trying to access its data from different threads.

Distributable - This is a class that essentially represents a kvstore. It is the core piece that we use for distributing data across a network. A Distributable has a map from String to a Transfer. A Transfer is a union data type that represents any kind of data we can store in the network. The logic in this class is relatively complex, but we essentially can send data to other nodes in the network using various kinds of

Messages from the Message class. We serialize and deserialize the messages to make this process more efficient.

KDStore - This class is responsible for storing and working with data in the application layer. Every single kdstore has its own kvstore, which it uses to store and access data that will be used in the given application. There is one kdstore on every node in the network.

Key - Class for a key in a kvstore. Has a string that represents its id and an integer that represents which node the value for this key is stored on.

Transfer - Class for a value in a kvstore. We use the union data type to be able to accommodate different types that a value can encompass.

Use cases:

1. Trivial

```
class Trivial : public Application {
   public:
       Trivial(size_t idx) : Application(idx) { }
       void run_() {
           size_t SZ = 1000 * 1000;
           float* vals = new float[SZ];
           double sum = 0;
           for (size_t i = 0; i < SZ; ++i) {
               vals[i] = i;
               sum += i;
           }
           Key key("triv",0);
           DistDataFrame* df = DistDataFrame::fromArray(&key, &kd, SZ, vals);
           assert(df->get_float(0,1) == 1);
           DistDataFrame* df2 = kd.get(key);
           for (size_t i = 0; i < SZ; ++i) {
               float cur = df2->get_float(0,i);
               sum -= cur;
           }
           assert(sum==0);
           delete df; delete df2;
           delete vals;
       }
```

2. Word Count - this is an application that takes in a text file and generates the counts of each word in the file.

```cpp
class WordCount: public Application {
public:
   static const size_t BUFSIZE = 1024;
   Key* in;

   WordCount(size_t idx, KDStore* net):
           Application(idx, net) {
       in = new Key("data", 0);
   }

   ~WordCount() {
       delete in;
   }

   /** The master nodes reads the input, then all of the nodes count. */
   void run_() override {
       if (index == 0) {
           FileReader fr{"100k.txt"};
           delete DistDataFrame::fromVisitor(in, kd, "S", &fr);
       }
       local_count();
       reduce();
   }
```

Open questions: where you list things that you are not sure of and would like the answer to

1. What is the best way to debug in projects like this? We realized that when there are multiple threads and a lot of data going around simultaneously, it is very difficult to diagnose and correct issues. CLion's debugger has proven to be useful, but we are curious if there is a better way.

Status:

1. We have successfully implemented a distributed system that can store dataframes efficiently and run complex applications that use their data. Unfortunately, we were unable to work enough on the Linus application, but given a little more time, we believe that this would be relatively straightforward, as our design is so that we can use new applications without much effort.