Introduction:

The eau2 is a distributed key value store. Nodes are clients from our client server network we created previously. Each node has part of the data.. How it works: Store a dataframe by putting it into the key value store. The key value store then converts the DataFrame into a byte array. The byte array is then split up into *n* chunks, where *n* is the number of nodes in our network. Each chunk is sent to a different node. When a dataframe get call is made to the key value store the store gets the different chunks from each of the nodes and combines them back together. It then deserializes the byte array into a DataFrame object and returns the object to the caller.

Architecture:

Application Layer: The user creates a key value store class and can put dataframes into the store. The key value store takes in a list of nodes which need to be created prior to the creation of the store.

Key Value Store: Converts the data into bytes and distributes the bytes between the nodes in the network. Each node uses the same key for the same dataframe. For example if a user wanted to add a dataframe with the key "df1", each node would have an entry in its map where the key is "df1" and the value is a portion of the serialized dataframe.

Nodes: Store the serialized data frame in a map. They have get and put methods to access the bytes the node is keeping track of.

Implementation:

Key: essentially a wrapper for a string at the moment.

Home node was removed as each dataframe is stored equally on each node.
Still exists in case we want to bring back the home node field.

KVStore: takes a list of existing nodes and uses them as distributed KV storage.

Takes and returns dataframes, but internally sends around serialized byte sequences.
put() chunks up dataframes into num_nodes chunks, then distributes them evenly.
get() requests the chunks from each node, then reassembles them in order and deserializes.

Node: uses an unordered_map internally to store KV pairs.

Use cases:

Makefile commands:

- make - builds each test case
- make build_[dataframe/serialize/fake_network] - builds a specific test case
- make run - build and runs each test case
- make run_[dataframe/serialize/fake_network] - builds and runs a specific test case

- make valgrind - builds each test case, then runs valgrind (with --leak-check=full) on each of them
- make valgrind_[dataframe/serialize/fake_network] - builds then runs valgrind on a specific test case

```
Dataframe df1();

Node n1();
Node n2();
vector<Node> lon = new vector<Node>();
lon.push_back(n1);
lon.push_back(n2);

KVStore kv(lon);

Key k1("key_name");
kv.put(k1, df1);
Dataframe df2 = kv.get(k1);
```

Open questions:

We were thinking of serializing a dataframe into a char*. We then want to split up that char* among our different nodes. Our problem is that once the dataframe gets a certain size we think the char* gets too large and our program crashes. Would it be a better solution to serialize columns of the data frame individually and then send those to different nodes instead of serializing the entire dataframe at once?

***Status**:*

- Everything leaks memory at a truly astounding rate, we will be working on this.
- We have a networking layer that is suited to the task, but is not integrated.
    - Fake networking with threads is working. But the test breaks when we include serialization.
- Dataframe works.
- Serialization generally works, but is slow and can break with large dataframes. We might change the implementation of this.
- We have separate large scale tests for each “section” of work in the tests directory. These tests cover many of the smaller unit cases as well as testing the ability to perform the full task.
    - The fake_network test has pieces of the other two, at a smaller scale. But the test fails.
    - The dataframe test runs on a huge dataframe, so it takes some time.