Jessica Cheng
Sharon He

Introduction:

The eau2 system is a distributed system that consists of three layers. At the bottom layer, there is a distributed KV store running on each node. The KV store is essentially a map that has get and put methods. The eau2 system is a distributed system so there will be multiple nodes. On top of each node, it will run a KV store. Each KV store has part of the data. There exists a network communication layer that connects KV stores together so KV stores talk to exchange data when needed.

The level above provides abstractions like distributed arrays and data frames. Calls to the KV store will implement these abstractions.

The application layer is the top layer. An application can access any data that is hosted through the system. It is running on some number of nodes somewhere in the distributed system. These applications can call the KV stores to get and put keys

Architecture:

At its highest level, eau2 is a distributed structure, such as an array or dataframe. Below the distributed layer, there is a networking layer, for the KV stores to communicate, and even below that, there is a KV store node.

In the networking layer, we have a "server" node that registers each of the other client nodes. All the nodes are KV stores. First, they register with the "server" KV store to let the server know they are part of the network. Then, when a node needs to get something at a given key, it will check whether the key is in the current node. If not, the key has a node index, and the client current node can send a request to the node index of the key, and that node will, in turn, return the key and value necessary for the current node to reconstruct the data it needs.

For the distributed structure, we have distributed KV stores across several client nodes. The nodes are our local clients in the network, and each node contains a KV store. The KV stores each have an index, and communicate with one another by way of keys, as they each have chunks of the overall data, which they can share by sharing keys and rebuilding the data from getting the values at the given key in the KV store.

Implementation:
The Key class is a subclass of class Object and represents a key in the KV store. It contains fields String name and size_t home. It contains Object methods such as equals and hash. The Key class's home node field indicates which KV store the key is in, to be used when networking, so clients can directly request and send to the node that contains the data they need.

The Value class represents the value in a KV store and contains a field which is the serialized char* data as well as a length field for the total length of the serialized string.

The KV store is represented in a class KVstoreand is similar to Map. The class contains Key and Value fields and contains methods:
get(k) => DataFrame* returns a deserialized DataFrame based on the given key k if k exists in the current KV store, or nullptr if k is not in the current node
put(k,v) => void puts a new key k and value v pair into the KV store
getAndWait(k) => DataFrame* returns a deserialized DataFrame. First checks if the key exists on the current node, but then blocks and waits for a DataFrame to be put in the node given by the key k before sending the DataFrame key over to the current node, which allows for the current node to recreate the DataFrame from the values stored at the keys in the distributed kv stores.

The Server class opens connections and listens for new clients. This class should store a routing table of where to find other clients and who is currently part of the network. The server node should only listen for clients and send them updated routing tables when a client joins or leaves the network. The server also handles termination, and is responsible for shutting down the network after all the key value stores are complete (or if the server itself shuts down).

The Client class is responsible for registering with the server. It stores a port and an IP address on which it will listen, as well as its own id. It should have its own copy of the routing table the server sends out, so it knows how to talk to other clients in the network. Additionally, the Client class should contain a KV store, which uses the

Client class to get missing values. The Client then sends messages to other clients asking for these values. If a client dies, the server will be alerted, and all the other clients will subsequently be told a client has terminated/left the network.

The DataFrame class is built off of a Schema, which holds the row and column names in the DataFrame. The DataFrame itself stores data in a columnar format, keeping track of the columns according to their types, the order of which are determined by the initial schema. In our updated code (as of milestones 2 and 3), the DataFrame can be built off of arrays of data and scalar data. This is done by first constructing a new DataFrame, given the type of the element being added. We then create a new column, setting the values from the array inside the column, and add the column to the DataFrame. In this way, we create a DataFrame with a single column, which is built up from an array or scalar value.

The Column class has been changed and implemented using chunks, which are essentially a double Array pointer. We have arbitrarily determined the size of each chunk (currently 1000, but can be changed), meaning each “element” in the column is an array of length <size of chunk>. This allows us to build a DataFrame from a column without becoming too large, and allows for more abstraction between the application and the payload.

Use cases: examples of uses of the system. This could be in the form of code like the one above. It is okay to leave this section mostly empty if there is nothing to say.

```
FILE *f = fopen("../src/data.sor", "r");
   SOR* sor = new SOR();
   char* schemaFromFile = sor->getSchema(f, 0, 1000000);
   printf("schema is: %s\n", schemaFromFile);
   Schema s(schemaFromFile);

   DataFrame* df = sor->setDataFrame(f, 0, 100000);
   df->print();
```

```
void testGet() {
        Schema s("II");
        DataFrame* df = new DataFrame(s);
        KVStore* kv = new KVStore(0);
        Key* key = new Key("df_key", 0);
        assert(!kv->get(key));
        Serializer ser;
        Value* v = new Value(df->serialize(ser));
        kv->put(key, v);
        assert(kv->get(key)->equals(v));
        printf("kv store get success!\n");
}

void testWaitAndGet() {
        Schema s("II");
        DataFrame* df = new DataFrame(s);
        Key* key = new Key("df_key", 0);
        Serializer ser;
        Value* v = new Value(df->serialize(ser));

        KVStore** all_nodes = new KVStore*[3];
        for (size_t i = 0; i < 3; i++) {
                all_nodes[i] = new KVStore(i);
        }
        // we don't yet have a network layer, but this would be our expected res
ult
//      assert(all_nodes[1]->waitAndGet(key)->equals(df));
//      all_nodes[0]->put(key, v);
}
```

```
void test_from_array() {
        size_t SZ = 100;
        double* fa = new double[SZ];
        for (size_t i = 0; i < SZ; i++) {
                fa[i] = 5.0;
        }
        Key* k = new Key("fa", 0);
        KVStore* kv = new KVStore(0);
        DataFrame* df = DataFrame::fromArray(k, kv, SZ, fa);
        Serializer ser;
        Value* v = new Value(df->serialize(ser));
        assert(df->get(0, 30) == 5.0);
        assert(df->nrows() == 100);
        assert(kv->get(k)->equals(v));
        printf("test fromArray success!\n");
}

void test_from_scalar() {
        double sca = 500.0;
        Key* k = new Key("f", 0);
        KVStore* kv = new KVStore(0);
        DataFrame* df = DataFrame::fromScalar(k, kv, sca);
        Serializer ser;
        Value* v = new Value(df->serialize(ser));
        assert(df->get(0, 0) == 500.0);
        assert(kv->get(k)->equals(v));
        printf("test fromScalar success!\n");
}
```

```
void testStringArrSerialize() {
    Serializer* serializer = new Serializer();
    StringArray *sa = new StringArray();
    String * strfoo = new String("foo");
    String * strbar = new String("bar");

    assert(sa->length() == 0);
    sa->append(strfoo);
    assert(sa->length() == 1);
    sa->append(strbar);
    assert(sa->length() == 2);
    printf("before serialize strArray\n");
    char* result = sa->serialize(serializer);
    printf("after serialized\n");
    Deserializer* deserializer = new Deserializer(result);
    StringArray * dsa = sa->deserializeStringArray(deserializer);
    assert(dsa->length() == 2);
    String* s1 = dsa->get(0);
    assert(s1->equals(sa->get(0)));
    String* s2 = dsa->get(1);
    assert(s2->equals(sa->get(1)));
    delete serializer;
    delete[] sa;
    delete strfoo;
    delete strbar;
    delete deserializer;
    printf("test StringArray de/serialize passed!\n");
}
```

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

- We're still not sure on how to do networking. Our networking layer from past assignments was very basic, and we were primarily communicating through the server. One thing we would like to know (and that we will work on) is how to have clients speak directly with one another at the given node.

Status:

- One big issue we have right now is networking. We are working on our serialization and deserialization, but due to misunderstandings about the specifications of the eau2 system, we have fallen behind on networking.
- We have implemented our Key and Value classes, as well as have 90% of the implementation of the KVStore class. The issue there is our waitAndGet should be blocking, whereas we currently have a while loop that listens for a put at a given key.
- Networking and direct communications between clients will make it easier to test waitAndGet, and will help us better understand how to implement it.
- We have built DataFrame from reading in a SOR file, but we need to improve upon this and perform more operations, as well as scale upward for larger files we may read in. (Again, this networking is integral to this operation.)
- We will continue to work toward refactoring, as our current code has a lot of repetition for different types. While this is a speedup at runtime, it costs us a lot of time when adding/modifying functionality and when debugging.