Introduction:

The eau2 system is a distributed key value store that allows us to build complex data structures on top of a framework of interconnected computers that increases overall capacity of our system and computation rates. This is based around the interaction of two main APIs:

1. A client-facing interface that shares a lot of similarities with our previous dataframe implementation. To put it simply, the two should work almost identically
2. A private API that uses a distributed form of the column and array data structures that are able to spread data chunked by column across multiple nodes in the same network to increase overall capacity. Most clients will work with our Application API - our highest level interface needing next to 0 networking knowledge. This codebase comes with two example Applications: WordCount, and Linus.

Architecture:

The architecture of the system is three-tiered.

The bottom layer is a KV store, run on each of the networked nodes containing an underlying map for key to value. Each key passed to the KVStore includes Node information so the request can be routed to the correct storage location.
This is also where the networking and concurrency information live to be as far away from the end user as possible and really only necessary at the KVStore level. Everything else will be built under the impression there is only one KVStore to connect to/worry about. The KVStore will serve its API requests in a multithreaded fashion, with a main worker thread that provisions data that lives locally, as well as making requests for data that lives on remote nodes. There is also a seperate listening thread that is responsible for handling responses, as well as all requests from remote nodes. Network requests in the KVStore are made through a specialized object that allows for registration of all nodes in the network upon system startup and further supports sending and receiving messages from any of the registered nodes. Furthermore, we use mutex and conditional variables to ensure thread saftey between the worker and the listener threads.

On top of this networked foundation, we're able to build datastructures that take advantage of the distributed nature of our data. These include special versions of Array and Queue. In particular, we leverage the KVStore to build a Distributed Array class that is adapted to now store data in a KVStore, updating the appropriate networked-node to send the next chunk of data to and how to pull data effectively from the KVStore utilizing the network architecture appropriately. These distributed Arrays provide us with all the necessary network information and storage logic so we can comfortably build a set of columns and a dataframe that take advantage of these features.

The topmost layer is the application layer, which uses all the distributed logic baked into the distributed Arrays and KVStore to provide a seamless service for consumers who will benefit from the added networking infrastructure without having to get their hands dirty with those implementations. The lowest level application we provide as an example is the dataframe: a storage solution for the main types of SORER data. As an example of this dataframe in action, there is also the provided Application API for both Networked and non-networked programs. This provides an interface to connect programs that perform tasks of all kinds on an underlying dataframe. with little to no networking knowledge required. See: WordCount.

Implementation:

```
class NetworkIP {
    NodeInfo* nodes_;
    int sock_;
    ...

    void server_init(); // start up a server on this node. Wait for client registrations, and broadcast directory.
    void client_init(); // start up a client and register with server. Wait to receive Directory
    void send_msg(Message* m); //connect to a remote node and send a message.
    Message* receive_msg(); //accept an incoming connection and receive a message.
}
```

The NetworkIP class provides an interface to establish a network of remote nodes and to facilitate seamless inter-node-communication. The network is established through a registration process. A node may act as a server in the eau2 network where it will initialize itself and wait for fixed number of client nodes to register, at which point it broadcasts a Directory containing the ip and port information for each of the registered clients. On the other hand, a node may as a client where it registers with the known server, and subsequently waits to receive a Directory for network nodes. Once a server broadcasts the directory from its end, and the clients receive them on their respective ends, the registration process is complete.
At this point the nodes in the network have all the information they need, to send/receive messages, to/from any other nodes in the network.

```
class KVStore {
    std::map<Key*, Value*, KeyCompare> local_store_; // contains all the key's that are supposed to be on
the local store.
    NetworkIP* network_; //Composition of the NetworkIfc.
    Lock store_mtx_; // Provides utilites such as locks and waits that are harnessed to ensure thread safety.
    Array pendingGets; // An object array of WaitAndGet requests that are resolved when new Key's are put
into the local store.
    ...

    //methods:
    void put(Key k, Value v); // put the given Key, Value pair in the store.
    Value get(Key k); // get the associated Value for the given Key if it exists. nullptr otherwise
    Value getAndWait(Key k);  // get the associated Value for the given Key if it exists. If not, block until it's
available.
```

KVStore uses a C++ map utility to store a one-to-one association between Keys and Values locally. The KVStore also serves as an abstraction of data that may live on remote nodes elsewhere in the eau2 system. The network_ is where the majority of the all of the node delegation networking occurs.
The put and get methods do what one might expect, creating a Key/Value association, and the retreival of such an association respectively. The getAndWait method is special in that it blocks the worker thread until the requested key exists in the kv store.
While the local requests on any given node running the KVStore are processed by the worker thread, requests coming into the node from other remote nodes in the network are processed by a separate listener thread. Additionally any waitAndGet's, for which the key is not available in the local store, are queued up in the pendingGets array. These requests are resolved when the missing key's are put into the local_store, and a Reply is created with the Value serialized in the message payload.
The Lock API provides a mutex locking/unlocking API, essential in ensuring thread safety of shared resources between the listener and worker threads. Furthermore, it also provides conditional variables that are used to ensure request fulfillment on the worker thread.

```
class Key {
    String name;
    size_t node;
}

class Value {
    char* data;
}
```

Keys include a specific name that is unique to class, stored values, whatever helps identify it, and a node number so the KVStore knows how to dispatch.
Value is expected to contain serialized data that will be stored in the KVStore

```
class DistributedArray {
    KVStore* kv_;
    Array* chunkArray_;
    Array* keys;
    Array** cache_
    size_t uid_ = rand();
    size_t chunkSize, chunkCount, itemCount, curNode, totalNodes;
}
```

If KVStores are the airplanes shuttling data back and forth, the distributedArrays are mission control. They keep track of how much data goes in each "chunk" that is stored on a node before starting to store information on the next node in the system. They also keep track of all the keys associated with their data and randomly generate a UID to minimize the risk of storing data with the same key as another distributedArray. itemCount, ChunkCount, and chunkArray are all to further help the Array delegate data effectively and with a minimum amount of calls to the KVStore assuming every time we reach out to the KVStore it'll require a network request.
To reduce the number of queries made from the DistributedArray to the KVStore, and to avoid the expensive deserialization of Value's returned, a cache of the array chunks is included in this implementation. This provides a signifcant improvement in the performance of our DistributedArray and is a more robust design, expanding the scope of usage in terms of magnitude of data that can be handled.

```
class DistributedColumn : public Column {
    KVStore* kv_;
    DistributedArray* val_;
}
```

DistributedColumn is essentially a wrapper over the DistributedArrays that adds more metadata for our Dataframe implementation (Columntype, as_int, as_bool) that allow all DistributedColumns to inherit from the same baseclass with all the same functionality and allow for polymorphism in terms of each type of DistributedColumn (int, bool, float) having the same getters and setters.

```
class DistributedDataFrame() {
    Schema* schema_;
    Array* cols_;
    KVStore* kv_;
}
```

Since all of our delegation and networking calls are done on much lower levels - it allows our DistributedDataFrame to be virtually indistinguishable from our previous DataFrame model. Besides the addition of a KVStore in the constructor which can then be passed down appropriately, all the accessor methods of getting rows, columns, or specific values function very much the same as before but call upon DistributedColumns as opposed to regular columns. This is the layer our clients should be seeing.

```
class Application() {
   KVStore* kv_;
   size_t this_node_;

   void run();
}
```

The base API for our Applications that given a KVStore (with/without a NetworkIP object) and the node the application is running on provide an interface between higher level code and the underlying dataframe and network architecture. The run method is the entry point to begin all Applications.

```
class Items_() {
   Array keys_;
   Array vals_;
   ...
   get,set,contains,pushback();
}

class JVMap() {
   size_t capacity_;
   size_t size_;
   Items_* items_;
   ...
   get,set,contains();
}

class Num() {
   size_t v;
}

class SIMap : public JVMap() {

}
```

In order to get our WordCount demo in front of customers as soon as possible we implmented the 3rd-party "JanVitekMap" library suggested by another dev team as the package used in the design brief. While it did speed up development time in our understanding of example Applications, it is somewhat redundant considering our previous map imlpementation and our move toward C++'s standard map. A future release will remove this dependency but right now it is required by several of our applications and dataframe visitors. Sample Docs for the library can be found here: https://piazza.com/class/k51bluky59n2jr?cid=1049_f1

Use cases:

let's explore three different use cases that each take advantage of a different level of our distributed application architecture.

The biggest use case for our distributed data solution will be at that top level - utilizing the Application API to design programs that build off of all our distributed dataframe.
This would require you to inherit from the Application interface and provide it with network details so our

software knows how to run and implement your new program.
For example, imagine you want to count the number of unique words and occurances per word in a provided data file. You might be tempted to design an Application called...WordCount!

```
class WordCount: public Application {
public:
  static const size_t BUFSIZE = 1024;
  Key in;
  KeyBuff* kbuf;
  SIMap all;
  char* fileName_;
  size_t num_nodes_;

  WordCount(size_t idx, char* fileName, size_t num_nodes):
    Application(idx), in("data", idx), kbuf{ new KeyBuff(new Key("wc-map-",0))}, fileName_{ fileName },
num_nodes_{num_nodes} {}

    void run_() override {
        //read from file, store in dataframe, store dataframe in KVStore
        //run local_count
        //run reduce
    }

    void local_count() {
        //pull stored dataframe from this node
        //generate map that stores count of each word from dataframe
        //store total from this node into new dataframe containing words and counts
    }

    void reduce() {
        //merge all local dataframes together combining results into one map
        //print words and count from the mega map
    }
}
```

With very little knowledge of the network besides number of nodes, network connection, and a filename WordCount can calculate the results of an arbitrary length text file distributing the workload across the number of provided nodes. This same technique of "divide and conquor" can then be applied to any other combination of storing node-specific dataframes in the KVStore and combining results at the end - similar to our original forray into multi-threaded visitors in a previous rendition of dataframe.

Another potential use case for our distributed KVStore is with the aformentioned Dataframe implementation. This class gives clients access to large-scale data storage with essentially limitless capability depending on how many nodes they add to the KVStore network.

Let’s consider representing a dataframe with 100 rows in it on the eau2 system that has a single home node and 10 auxiliary nodes.

The home node creates a dataframe object and begins to add previously defined row objects.

```
KVStore(num_nodes: 10, this_node: 0)
.
.
.
KVStore(num_nodes: 10, this_node: 10)
//these are added on each node in the system

DistributedDataFrame* df1 = new DistributedDataFrame(schema, kvstore0);
df1->add_row(r1);
df1->add_row(r2);
.
df1->add_row(r100000000);
---
```

Once the KVStores are up and running on each of the individual nodes, passing the first one to the Dataframe constructor will link them to the Dataframe in a way that the end user can continue to use the Dataframe in the ways they're used to but with in this case up to 10x the previous capacity.

This example dataframe has 100,000,000 rows distributed across 10 nodes...but this is completely arbitrary and we could have simply kept going.

Another use case for the more technical users of this interface who aren't happy with our dataframe storage solution could elect to build their own or any further extension of the DistributedArray.

```
IntDistributedArray* ida = new IntDistributedArray(KVStore: kv, chunkSize: chunk);
```

Given another KVStore like earlier, you can also tweak the exact parameters that change how large data is allowed to get before storing on the next node in the sequence

```
void pushBack(int val) {
   //initialize first key
   if(itemCount_ % chunkSize_ == 0 && itemCount_ != 0) { //if current chunk is full..
      storeChunk(); //stores the previous chunk values in the kv_store
      chunkArray_->pushBack(val);
   } else {
      chunkArray_->pushBack(val);
   }
   itemCount_ +=1;
}; //add o to end of array
```

By changing the chunkSize you can minimize the number of network calls if that is something your application requires or for further redundancy you can reduce chunkSize to make sure data is constantly being stored in the other nodes.

Open questions:

Our current wordcount implementation uses a single node while we finalize a lot of the tests and safety nets in the networking implementation. While we don't think the connection of the two will be too painful since both parts were built with the other in mind (WordCount has certain networking specific lines prepared to "switch on") - it just wasn't something we had time for this week and that will be the focus of our next sprint - how easily will these two parts fit together?

Another is our reliance on the JV library of maps which we hope to get rid of in the near future but how easily will it be to detangle ourselves of unnecessary dependencies?

4/20/2020
While we will go over all important status updates in the next section, we wanted to include a quick update here referring to the questions that remained open until the final code examination. To address the first: while there were some minor hitches to get the networking protocols working with application. They didn't prove to be too challanging. On multiple terminal instances in the same machine we have seen success on both word count and Linus. There were issues when expanding this to multiple machines but that will be addressed in-depth in the status section.

To address the second question: The answer is we just didn't have the time. While it would have been nice to consolidate all our map implementations under one roof, we just didn't have the bandwidth to achieve it. For now, we'll continue to utilize the "JanVitekMap" 3rd party library for our applications and most likely we'll end up abandoning our custom map solution - opting instead for standard map and a privately hosted copy of JanVitekMap (provided their licensing agreement allows us to include their library in commercial software). The battle against code-bloat was one that in the grand scheme of development we just didn't have the power to deal with. Once we deliver this final product to customers we would like to address it for Eau2.2 but for now it's remaining on our trello board in the backlog.

I think the only open question remaining, and the status update for 4/20 will shed more light on the reasoning behind this, is if we would have been more successful in an actual school environment? A lot of our network troubles came from individual configurations and wifi not being something either of us could count on uninterrupted. It's been really frustrating, and just left us wondering if this wasn't working against us, would our final project have met more of the expected criteria?

Status:

For MS1:
Currently we have brought our existing dataframe implementation up to speed and combined it with a C++ Sorer representation provided by github user euhlmann (https://github.ccs.neu.edu/euhlmann/CS4500-A1-part1). We are able to successfully read .sor files into our dp using the main.cpp file located in the sorer directory.

03/29/20:
To put it simply - coronavirus. We've been doing our best to keep up with the workload and adjust to this system of working remote and zoom calling but it definitely has taken a long time to get used to since that really wasn't our style prior to moving completely online. Milestone 2 took significantly longer than expected but we're happy with the result, re-evaluating extra baggage in the codebase. For this milestone our focus has been on the distributed aspect involving new code in:

- utils/distributedArray.h
- dataframe/distributedColumn.h
- dataframe/distributedDataframe.h
- store/
- updated serialization all around with tests in serialize/main.cpp and serialize/testSerialize.cpp

While we did encounter a malloc error when actually running the application code for MS2 (testApplication() in tests/testSerialize.cpp) we believe we have all the pieces to make the code snippet run, and it is only a case of debugging.

4/6/20:
Over the past several days we've been working to get caught up with Milestone3. Significant areas of work include resolving existing bugs in the code for milestone 2 that included fixing a malloc error, as well as improving the performance of our DistributedArrays drastically through local caching. We further worked on building out a simple network layer where nodes can connect and communicate with each other. With that done, we started out work on actually distrubting the KVStore across nodes in the network, sketching out an implementation that is, at least in theory, robust and more importantly fault-tolerant. While the implementation is complete, atleast in terms of code, we didn't have the chance to fully test and fix any bugs with our implementation and run the Demo application. However, we believe we're extremely close to getting there.

While we are much more used to the peer programming method of doing things, both of our bandwidths have made that somewhat...difficult. So instead we've opted to go forward with parallel programming with daily "standup" and bug reports. This has allowed us to progress in the networking side of things, getting it up to speed with the requirements of M3 while also pushing forward with M4. The one drawback of course being that building a distributed application on a system without a fully functioning distributed layer is not really ideal. Instead, the focus has been on designing the WordCount application on a single node with structures in place to make the transition to multi-node once networking is up and running as seamless as possible. The WordCount test is located in the makefile running on the provided 100k.txt sample file with results printing to console. Obviously this will be faster once we have it working across nodes but the integration of specs into our current implementation was something that took quite a bit of work. At this point we believe it'll really be a matter of plugging the two cables together and yelling "IT'S ALIIIIIVE" before it all blows up in a shower of sparks.

Lastly, drawing on feedback from our most recent milestone we realized there's a bit of redundancy in our codebase. Particularly around having multiple database implementations (distributed and non) as well as the support in terms of columns and rows for each. This also includes a few old school rowers and fielders that have been somewhat outstripped by the new Readers and Writers. This is something we considered addressing in this sprint, but unfortunately didn't have the time to detangle the non-distributed dataframe implementation from the SORER input for M1 and just having it use a single node distributed df. This is a code smell we are aware of planning to address further in this upcomming week to bring everything on board with the same implementation of dataframe and columns and rows. This will also clear up our repo quite a bit and nip any potential legacy bugs lurking under the surface....

Our next goal is to pair as much as possible to handle the integration and then proceed to M5 more or less back up to level with management's current expectations.

4/13/20:
Milestone5 has put some of the bugs with our Sorer Implementation front and center as we spent the first part of the sprint scrambling to adapt our non-distributed Dataframe Sorer file reading to the distributedDataFrame in the manner described in the Linus application. I.e. having a static method in the DistributedDataFrame class that can generaten a new DistributedDataFrame based on file input. While this is most certainly the goal, we wasted a lot of time dealing with circular dependency issues between this new static method and our SorParser class. Since the Dataframe::fromFile method would need to use the parser, and the parser would require a method that outputs a DistributedDataFrame...we had a classic chicken/egg scenario. We did explore separating the header files from their implementations like we did previously with deserialize...but we would have needed to do it for both files and add some extra "inline" method classifiers to prevent duplicate symbols. While this would allow us to have a Dataframe::fromFile method, we didn't think this entire code re-organization using paradigms we're not as readily comfortable with was worth the cost and opted for a helper function that exists in the Linus Application and in the test_sorer_dataframe tests that will allow DistributedDataFrame to not require knowledge of the parser.

Long-winded way of saying we know that this helper should not exist in the Linus Application but we have yet to find a cleaner alternative that doesn't put more of our code in danger.

We also ran into a couple issues with constructor overloading that will require further examination. All of our serialization constructors take in a char* as their only argument. In the cases of a few key classes, they also have const char* constructors. This can...lead to some problems. Right now, the main concern is with the Schema class where the problem was discovered in Sorer debugging. For now, a String* constructor has been added to eventually replace the const char* Unfortunately we didn't have enough time to find all the problem areas so what we'll end up doing is removing the problem constructor and watch all the red warning lights pop up one by one and systematically take them out. Shouldn't be too bad but will take time.

Apart from fromFile implementation, the majority of this week was spent getting our networking code up to par...
We were finally able to get our KVStore to support distribution of data across network nodes. An example that demonstrates this aspect of the KVStore by running the Demo application that was due for M3 in separate shells and have them communicate with each other to produce the correct sum as is expected. This example can be from the top-level Makefile in the project by running the make demo target.

Lastly, we adapted the Linus application for our Dataframe implementation. This step wasn't too difficult since we did get a feel for it with last assignment's wordcount but does always have a few tricky parts with different function names and signatures that need to be changed or better understood why certain things aren't pointers, etc. We did implement a conceptually correct version of the local_map method, which was a todo item last week, however, we were running into some segfaults which we didn't have enough time to debug and fix before the deadline, and therefore haven't been able to test the Linus application yet.

This was a long week, and while we didn't fully complete the goal at hand we plan to spend time in this final week before the reviews touching up on these last few problem areas before the presentation. Most of the work to be done is more clean-up and optimization focused which is a really good place to be. Almost there!

4/20/20:
This final week has been split into two significant intiatives: getting the Linus application working, and preparing a demo across two computers to assess the networking capabilities of our eau2 system.

For Linus: after coming so close at the end of ms5, we did manage to get Linus working in the beginning of the week. This involved squashing lots of seg-faults line by line, tracing them all the way from the application layer down to the individual arrays in some cases. But luckily for every hidden bug, there was another on the surface that caused more ego damage than time lost finding them. A personal favorite was forgetting how we implemented deserialization and trying to cast a char* as a DistributedDataFrame...good times. Once we got through all that, and called in professor Vitek at least once for a memory issue (deleting a pointer from the stack then needing it again), Linus proved operational and we ran countless tests at the 10,000 and 100,000 entry levels. There aren't a lot of concrete tests for these experiments which is an area to clean up but we had already spent so long on this that we knew we needed to move on to the more networking heavy presentation side before we could go back and clean this up.

For the Code Walk/Preparing Linus on multiple machines. Here...we ran into a lot of problems. Some unique to our current situation, others due to the compromises we made along the way to reach this point in the time alotted. Starting with the latter, we've never put a lot of attention into Valgrind. Yes, in a language like CwC that requires us to keep track of memory this is considered a Bad Thing but with the schedule management asked us to keep and the sprint goals we routinely kicked the Valgrind issue down the road in favor of new features and being able to show a working prototype to management. This eventually came back to bite us when attempting to shift code onto AWS. On any of the t2 instances, even with only 100,000 entries in the Linus data files, we would run out of memory before the files were even loaded. We did eventually get the Linus application working on a single node using a...m5ad.12xlarge but something told us we probably shouldn't need something so big with such a paltry amount of data to process.

However, once we started trying to run Linus on two terminals of the same instance the process would start...but unlike when we ran the same code on our own machines...it would eventually crash with the same amount of data. All else being equal, we have to assume this was a memory thing that makes it require far more space to run than it really should. Now, there were other things that indicated these instances had different requirements than ours to run. For example, we needed the -pthread argument when compiling which we didn't need on our local machines. It wasn't an area we really had time to fully explore but it did serve as a pretty efficient roadblock to our AWS efforts.

So, we went to fallback plan A) getting Linus working on our personal machines. At least in this configuration we could put a name to our enemy: network routers. To form a stable connection between our two machines we'd need to set up port forwarding on both of our home routers which we both found to be a relative challenge. Neither of us have access or the permissions required to make router changes in our housing situations so this made the entire venture an impossibility.

These leads us to our final option of demoing the application on two threads of the same machine which is not exactly where we wished to be but at least due to SOME events outside of our control, it's the best we can do in the timeframe.

But for all of the troubles we faced this week in our final networking attempts we're still incredibly proud of the success of our Linus application and the Eau2 system as a whole. While memory sacrifices and code bloat do still linger in the darkness, they are known problems that would require nothing more than consistent time and energy to unravel. With other finals, internet circumstances, and COVID being as they are these things couldn't be resolved before our walk but if this class would have continued - memory and tech debt would have been first on our list to fix. Thanks so much for an exciting semester and we wish you all the best.