INTRODUCTION:

High-Level description of the eau2 system ...

The eau2 system is a distributed key-value store used for large scale data processing. To optimize efficiency, the system will be distributed across multiple GPU's allowing for the development of different computations upon the stored data, including machine learning algorithms.

ARCHITECTURE:

Description of the various parts of eau2 at a high-level ...

1. Dataframes and Abstractions:
    - Our storage structure supports reading in large files (around 10GB) by using a data parser that converts "unstructured" schema-on-read data to a dataframe.
    - Dataframes act as components in our key-value store, specifically as the value portion of each tuple.
    - Data inside very large dataframes can be split and distributed in chunks stored on different nodes within the system for memory and efficiency reasons.
2. Network Layer:
    - Facilitates communication between nodes in the distributed key-value system.
    - Dataframe values will be serialized when being sent across the network.
    - Concurrency control built in to support multiple operations at once without interference, preserving data consistency and integrity.
3. Client Application:
    - Connects to above three layers via APIs exposing only necessary methods to support data input and retrieval.
    - Can be running on any number of nodes within the system and interacts with the node it runs on only, shielded from the cross-node backend communication.

Solution Structure: * application * src * dataframe * modified_df * network * serializer * store * util * test * report

IMPLEMENTATION:

Description of how the system is built, including relevant non-utility classes and their APIs ...

1. Utility Classes:
    - At the lowest level, we have utility classes such as Strings, Arrays and Maps that serve to store collections of data. Strings are a collection of characters, and are treated within the system as one of the primitive data types that may be present.
2. Dataframes
    - A dataframe consists of a schema and set of columns. The schema defines the structure of the dataframe; this includes data such as: column names, row names, and the type of data each column stores. The schema grows when columns or rows are added to efficiently reflect all updates. Dataframes also have a map() method to apply a specific function operation upon it, as well as a parallelized version to increase speed upon large amounts of data.
    - All columns are of one of four types, subclasses of an abstract Column. The four columns include: BoolColumn, IntColumn, FloatColumn, and StringColumn. These columns store data of the same type in a columnar representation. Values and column names can be get and set within columns. Columns support getting and setting of values (if the values match the column type).
    - Our implementation includes a Row class responsible for storing values across all columns at a specific depth. The purpose of the row is to facilitate addition of data in a dataframe. In order to optimize dataframe flexibility, our implementation of a dataframe accepts visitors upon rows. The visitors of the rows accept visitors of fields such that the entire dataframe can be traversed and manipulated with ease. Rows can store elements of different types.
3. Network Layer and Transport Protocol
    - The network communication utilizes a rendezvous server. Each client stores its own IP address and maintains a list of other clients connected to the same server.

    - Any time a new client registers to the server, each client's list of possible recipients is updated accordingly by the server. The client can then send any message directly to a client of its choice, so long as the recipient is registered to the same server. To ensure that each client can successfully send a direct message to each potential recipient, we generated a new socket for every registered client on the server.
    - Similarly, to ensure that a server can receive messages from more than one client, a new socket was generated for each client registered to that server. There are client and server objects that facilitate communication (sending and receiving messages via sockets).
    - When communication is finished the server will send out a `die` message to all its connected nodes, beginning an organized teardown where all connections are terminated
    - Messages sent over the network can contain data. This data is in the form a serialized string, that is then deserialized into actual object types by the receiver. Serialization and deserialization occur from the bottom up — primitive types are handled by `Serialize` and `Deserialize` classes, arrays/columns utilize this to serialize/deserialize the actual sets of typed data one element at a time, and then finally dataframes serialize/deserialize themselves by dealing with all their fields including sets of columns..
4. Key/Value Store
    - The key/value store is distributed meaning its components can be spread across many nodes. Each node will run its own key value store, which under the hood is just a map from `Key`s to `Values`. Keys are used for searching for values, and each key keeps track of which node owns its data. Values are simply serialized blobs of data, such as serialized dataframes. The KV store supports three major operations: put(k,v), get(k) and a blocking getAndWait(k).

USE CASES:

- Creating a dataframe from a client application is very simple. The data parser `Sorer` can be used to parse a file and create a corresponding dataframe with a schema and filled columns. `Dataframe* df = Sorer::(“filename.txt”);`
- Creating a dataframe from arrays and scalar values. Serializing and adding that serialized dataframe into a key value store.

  Key main("main", 0); Key check("check", 0); KeyValueStore* kv = new KeyValueStore();

  size_t SZ = 100 *1000; int* vals = new int[SZ]; int sum = 0;for (size_t i = 0; i < SZ; ++i) sum += vals[i] = i;

  DataFrame* from_array = DataFrame::fromArray(&main, kv, SZ, vals); DataFrame* from_scalar = DataFrame::fromScalar(&check, kv, sum);
- Retrieving (deserializing) a DataFrame from a key value store by giving a key associated with the serialized dataframe.

  DataFrame* v = DataFrame::deserialize(kv->get(main).serialized_data);

OPEN QUESTIONS:

List of things that are not concrete and would like the answer to ...

- How should one decide on how large each chunk of the distributed arrays should be?
- ~~For now, dataframes are meant to be immutable. How does this conflate with the mention of erasing and overwriting key/value pairs in the assignment description?~~
- ~~How to incorporate communication between completely separate key value stores running on separate applications?~~
    - ~~How a "master node" is spawned? And if that node exists on the client side?~~

STATUS:

Description of what has been done and an estimate of the work that remains ...

1. Done:

    - Utility classes are complete.

- Structures to store data such as Dataframes and all associated components like typed Columns and Rows are memory efficient and functional.

- Reorganized solution to a logical structure with relevant directories.

- Streamlined testing: Included tests for all completed layers and relevant methods.
- Completed serialization and deserialization of all relevant types from primitives up to and including DataFrame.
- All M2 functionality is complete including an integration test.

2. Need Improvement:
    - **The network layer does not properly send the directory messages to the registered nodes and therefore, client to client communication does not fully function.**

- ~~Serialization and deserialization for primitive types is complete as well as for arrays of integers and booleans. We are currently working on this for arrays of floats and Strings, and then upon Dataframes but this should be relatively straightforward in approach.~~

- Our data parser that converts text files into dataframes currently assumes the data is space separated; this must be modified to support the schema-on-read format and also refactored into its own class.

3. To Be Completed:
    - ~~Finalize networking layer by fixing minor bugs preventing socket connections. (sending directory messages to all registered nodes).~~

- The distributed nature of the data needs to be incorporated into our storage structures. Logic needs to be added to split dataframe components like columns into chunks that can be stored across the network.

- ~~WaitAndGet() is not fully implemented due to a lacking networking implementation.~~

    - ~~Our underlying Key and Value classes need to be created and hooked up to the revamped network layer.~~

- Threaded response in KeyValueStore. (We really tried our best.)