Introduction:

The eau2 system that we designed is a distributed key/value system, which is designed to be used for big data analysis. The goal is for it to be scalable and fast, handle large files with ease, and allow an analyst to issue interactive queries.

Architecture:

The architecture of the system is as follows. There is a central server, which has the sole responsibiltity of tracking all nodes on the system. It does not store any data itself. The data is split accross a variable amount of client nodes. On start-up, these nodes register with the server. Once a node is registered with the server, it receives a list of all other registered nodes, and all other nodes receive an updated list containing the new list of registered nodes. Using this list, the client node opens connections to all other nodes, allowing it to share its data accross the network, and to receive data from other nodes. To facilitate this, the nodes pass each other the list of keys that they are storing locally. For efficiency, the network connections each have their own thread, seperate from the thread listening for new connections, and in the clients case, they also have a main thread so that data can be queried. Any of the client nodes can be queried for data stored in the distributed system. The client node will first check if the data is local (the trivial case), and if it is, they will return it. Otherwise, it will check its list of non-local keys to see if it is, on the network, at a different node. If it is, then the lcoal node will request the data from the remote node that claims it has the key. This node will serialize the data, and send it over the network to the requesting node. This node will deserialize the data and return it to the requesting thread, allowing it to proccess it however it wants.

Network:

The Network allows for data to be stored remotely and to be returned to a local query as needed, allowing for fragmantation of large data accross the network. This is built using a client-server architecture, where the sole responsibility of the server is to maintain a list of active clients. All communication gets its own thread, to avoid blocking computation. There are three types of connections in our model:

1. Client to Client Connection (represented by the `CtCConnection` class).
2. Client to Server Connection (represented by the `CtSConnection` class).
3. Server to Client Connection (represented by the `ServerConnection` class).
   The networking is composed from a single server running on the network. Each node runs one client instance, which provides the networking for that node.

NetPort

The `NetPort` class is the abstract base class from which both the Client and the Server inherit from, which allows us to abstract a lot of the boilerplate network code required to receive and create connections. This base class stores the address and port it is bound to, the file descriptor of the listening socket, a list of active connections to the listening socket (which each get their own internal thread), a boolean representing whether it is running (as it should infintely loop if it is), and a time point that is updated on network activity, so that it can shutdown if there is no activity on the network (at least in a testing context). This class abstracts code to listen on the network, and provides a series of abstract methods for its subclasses to implement to customize the behavior of the main event loop. It also uses a modified singleton pattern to restrict a program to a single NetPort instance, either a Client or a Server (in our case), and no more, through the use of a static pointer and protected constructors. As a result, all move and copy sementatics are deleted.

Client

The `Client` class represents a client node on the network, and inherits from `NetPort`. It requires an IP address to bind to, and the IP Address and Port it can find the server at. It stores internally, the location of the server (IP Address and port), a pointer to its Client to Server connection which facilitates communication with the server in a seperate thread. A list of other clients, and a boolean representing whether a new client has joined the network recently, so that it can check if it needs to open a connection to said client. It has customized the NetPort event loop so that it initially connects to the server and registers, and then on new updates of other clients opens connections to the new client, if one does not already exist. It also provides a public method for the KVStore to use to request a remote value, by giving it a key with the address of the node. This finds the correct connection object, which is connected to the correct remote node, and moves the request there, to be handled. It can create two connection types, the `CtSConnection` (Client to Server) and `CTCConnection` (Client to Client). As a connection can be viewed as an extension of the client, these are friend classes of the client so that they can provide updates from remote clients to this client.

Server

The `Server` class represents the server on the network, and inherits from `NetPort`. It stores a list of all active clients on the network (clients which have registered with it), and some helper atomic variables to help ensure that all clients are updated when a new client registers or an old one deregisters with the updated list. Its event loop is much simpler, it just waits for new connections or for clients to deregister, and updates the entire network of that change. It also is designed to pass a teardown request to the whole network, so that all nodes close at once. It can create `ServerConnections`, which are connections to clients that connected to it running in their own thread. To facilitate updating these clients of the list of clients, this is marked as a friend class, as again, this type of connection can be viewed as an extension of the server itself.

Connection

The `Connection` class is the abstract base class of all connection types, and provides an abstraction of the common network operations required to maintain and utilize a connection from one node to another. As a result it stores the fd of the socket which is connected to a remote node, and the thread which is running the main event loop of this connection. It provides abstracted functions to send data, receive data, and parse received data into our internal packet representation, which are used by all connections. It also stores a time point to allow the connection to close if a long period with no network activity passes, to save on memory usage and CPU contention.

CtCConnection

The `CtCConnection` class represents a connection from one client to another, and inherits from `Connection`. It is either in a receiver state or a client state, which describes its initial value (eg: the client sends the id of the node it is on to the receiver). Following this, they keep the connection alive, and pass

the list of local keys to the remote client whenever said list is updated. It
also handles requests for remote data. To do this it has a request queue, which
it periodically checks as part of its event loop, and if it finds a request,
it asks the remote node it is connected to for the data corresponding to the key.
On receiving a request it asks its local KVStore for the relevant data, and then
passes that data back over the network to fufill the request. When receiving a
response to a request it made, the connection provides a value to the pointer of
the request object, and then wakes the thread waiting for the request to be completed
(using a condition variable).

CtSConnection

The `CtSConnection` class represents a connection from a client to the central server,
and inherits from `Connection`.
It initially registers the client with the server, and then afterwards simply facilitates
updating its client instance with what other clients are active on the network. It
also provides a way to ask the server to teardown the network. It also allows a client
to deregister from the network before shutting down.

ServerConnection

The `ServerConnection` class represents a connection from the server to a client, and
inherits from `Connection`. It simply allows a way for a client to register with the
server initially, and to deregister when the client wants to shut down. It also
lets the server keep the remote client updated with the list of active clients on the
network. Furthermore it can tell the client that the network is being torn down,
and to shut itself down.

Packet

The `Packet` class represents a packet we use to facilitate network communication.
It uses the Type-Length-Value (TLV) encoding format. It stores a type (a 8-bit unsigned number),
and a vector of unsigned 8-bit numbers representing serialized bytes. The vector
inherently tracks the length of the data as well. It provides a method to pack itself
into the encoding to be used to transmit it over the network, and a method to
unpack data from the network into the representation that the class provides. It
also supports partially unpacking an array of data, and repeated calls to that
method to parse large amounts of data into this representation.

Data

The data of the distributed store is stored in a local KVStore which provides the
interface to request both local and remote data (the networking is opaque to the
user). This store returns DataFrames, which allow for efficient operations over
large amounts of data.

KVStore:

The `KVStore` class represents the storage of keys to values, and also tracks
the keys who have values being stored on remote nodes. Internally it uses the STL
`unordered_map` to store a set of Key-Values, and the STL `unordered_set` to store
the set of keys on remote nodes. These keys store the ndoe they can be found on
as metadata, but this is not used to compute its hash or equality, so that one
does not need to know which node a key is on to request it. It interfaces with the
client to request remote data and return it as though it was stored locally.

DataFrame:

The DataFrame represents the actual structure of the values stored by the KVStore, allowing for an abstract represention and access method of the internal data. This also allows for missing values, which are represented by the STL optional class. It provides a threaded map (and a single threaded map and filter) to efficiently operate on the set of data, or one can request specific data by indices from it. It is internally composed of a `Schema` and a list of `Columns`.

Schema

The schema represents the format of the data, specifically the type of each column. It also allows one to name a given row or column, instead of using indices.

Column

The `column` abstract class stores a list of a single type in column form. Internally it is a vector of optionals (to support missing types), and provides support for Integers, Booleans, Floating Point Numbers, and Strings through subclasses.

SorerDataFrameAdapter

This is an adapter written around the sorer implemntation provided by another group. The upstream implemntation can be found at: https://github.com/gyroknight/boat-a1p1. Our fork containing a few bug fixes, and which generates a static library instead of a binary can be found at: https://github.com/NeilResnik/boat-a1p1. This adaptor namespace simply wraps their implementation and uses their parser to generate our own internal DataFrame objects from Schem-On-Read (SOR) files.

Serializable

`Serializable` is an abstract base class which allows any object to serialize and deserialize by inheriting from this class. It also provides some static template methods to be specialized to serialize non-custom objects, and to deserialize any object which implements a serialization through this class.

Application

Application is an abstract base class which represents the overall application running on this node.

Implementation:

network

client

when client gets request, it usually starts by a client requesting to get value;

```
std::shared_ptr<DataFrame> Client::get_value(const KVStore::Key& key);
```

the get_value will get value return a dataframe, the implementation includes locks to
make sure the progress is working properly. In this method, it will iterate through all the connections. If the key
finds the same address it will unlock, and call the get_value_helper() with giving the connection and key name.

```
std::shared_ptr<DataFrame> Client::_get_value_helper(std::shared_ptr<CtCConnection> c, const std::string& key
```

What this helper does is that it will add this connection to request queue in the connection. And put the client to a
sleep status and wait until it gets responese then wake up.

ctc_connection(Client to Client connection)

(Not all methods is listed, but it contains the main functionalities)
The ctc_connection contains methods as the following:

```
void CtCConnection::run();
```

Does the whole process of running a connection:
declares a role either a receiver or client, it contains check for finish, send keys and check requests. After it receives it will parse.

```
void CtCConnection::add_request(std::shared_ptr<CtCConnection::ValueRequest> request);
```

it will be called from when the client needs value, and it will add the request to a request queue.

```
void CtCConnection::_check_requests();
```

On the other side of the clients, it always looping until the requests is cleared, if it finds a request, it will pack a packet with the receiving string, and insert the string to a packet after serilizing it and send the packet. This method is being called in the run();

The connections will handle the communication between clients and clients; the way they communicate
is using the packet so the packet contains the purpose of it. For example, if the message is sending all the keys, the type of the packet is key_List…

```
void CtCConnection::_send_keys();
```

The way they send keys is by using the `get_local_keys()` (which will be explained later in kvstore) to retrieve the keys
and by sending it with a key_List packet type.

```
ParseResult CtCConnection::_parse_data(Packet& packet)
```

This method parse a result after get the responses, if the results is response value, or request value, or sends a key list.

```
ParseResult CtCConnection::_parse_request_response(Packet &packet)
```

This is a helper of parse data, deal with value response, it deserialize value to a dataframe, and wake up the server and tells it parse correct.

```
ParseResult CtCConnection::_update_keys(Packet& packet)
```

helper of parse data, by iterating the key that the packet has.

```
int CtCConnection::_respond(Packet& msg)
```

respond to client, by sending a package that is ValueRequest type. Or tell if it fails to response. It needs a helper methods called get_requested_value(string key), to send the packet after inserting the value.

server_connection

only contains a run_() and parse_data() method, because server only talks to client.

packet

a packet contains mostly two functionalities `pack()` and `unpack()`.

`pack()` is inserting the size and value;
`unpack()` functionality has another method called `partially_unpack()` which allows to unpack a large size dataset and by spliting them into different parts. The size is max 4096 bytes most each time.

KVstore

The kvstore contains the functionalities of the following:

```
std::shared_ptr<DataFrame> get(const Key& k);

std::shared_ptr<DataFrame> get_local(const Key& k);

std::shared_ptr<DataFrame> get_or_wait(const Key& k, time_t timeout_ms = 0);

void set(const Key& k, const std::shared_ptr<DataFrame>& df);

std::unordered_set<std::string> get_local_keys() const;
```

kvstore is able to get the DataFrame by a provided key.
kvstore is able to get the local DataFrame if the key is contained locally. (This is a helper method for `get()`)
get_or_wait will take it to other branchs if the current node can not find the key.
set will do as a `put()` which stores a key with dataframe.
`get_local_keys()` will help to retrieve all the keys that are locally storing, it is built for convinience when the other client is looking for keys, and to check if this node has the one the client wants.

other medthods are mostly related to extends from object or serializble or doing hash..

Key

Each is key is constructed by a string name, and a sockaddr_in tells where it belongs. Also each key is extends by a Serializable. The name in the key is used as an identifier which is mutable, because it doesnt change the hash.

Dataframe

In Dataframe class, We stored three things: the schema object, all the columns and the column type.
For the basic functionalities of the dataframe, please take look at the use-case.

Serializable

The implementation of this class is depends on where it want to be serialized.
Under the serializble class, the class works like a java interface.
The way we serialize and deserialize is to split them into three types:

1. primitive types
2. string
3. other objects

serialize

The methods that does serialize contains the following:

```
    //for primitive_type
    static std::vector<uint8_t> serialize(T t);
    //for base class
    virtual std::vector<uint8_t> serialize();
    //for string
    std::vector<uint8_t> Serializable::serialize<std::string>(std::string s);
```

deserialize

deserialize just memcpy the data by shifting bytes to an object.

```
  //deserialize to primitive type
  static T deserialize(const std::vector<uint8_t>& data, size_t& pos);
  //deserialize to string
  std::string Serializable::deserialize<std::string>(const std::vector<uint8_t>& data, size_t& pos);
```

The way to serialize and deserialize string is different, so they are both implemented in the serializble, if the types are other primitive type, they will use the method as above commented.

Applications

Trivial is left for M2 for testing;
Demo is used to test as M1 provided and M3 requested.
Application has an ip, server ip, server port and a mode.

Demo

This class was implemented by using a SumRower();
This rower is joined and can only work with type of float. What it does is as mentioned in the Architecture part.

Trivial

This is a class only used for testing, which inputs an array of prmitive type.
And to make them to dataframe without doing any other functions.
(what is required for M2)

Linus

This is an application could retrieve the people
who worked on projects up to seven degrees of Linus Torvalds.
Degree 1, is people who worked on the same projects as Linus,

Degree 2 is people who worked on projects with Degree 1 folks, and so on.
This class was implemented with `IntsetGenerator` , `UnorderedFilter` , `UUIDsToProjectsFilter` , `ProjectsToUUIDsFilter` , `UUIDToNamesFilter` .
How each one is being used will be explained in the use case:

SorerDataFrameAdapter

The implementation of this class contains a sorer that we used from other group (previous assignments). With that parser, and the file (also filename) that we were given, we want to return a Dataframe in a shared pointer and initialize the schema of the dataframe. And by having the dataframe, we can add row by row with a private method called `parse_and_fill_row()` , and while there is no more rows need to be parsed, it will return the dataframe.

For `initialize_schema()` , we take in the sorer, and iterate through column by column, and store the format in a string.

For `parse_and_fill_row()` , we implenented by using a switch cases, it iterates thrugh all the columns and get column type for each column, and in each primitive type case, there is a parse function for it.
For boolean and integer, we used std::stoi to parse the value.
For double, we used std:: stod to parse the value.
For string, we copy construct a heap allocated the string to parse the string.

Use cases:

Here are some simple examples of how to use them, for more implementation, please look at our tests.

DataFrame:

the methods under DataFrame:
fromArray(kvstore kv, key k, (primitive_type)* arr , size_t len)

```
        size_t SZ = 1000*1000;
        double* vals = new double[SZ];
        double sum = 0;
        for (size_t i = 0; i < SZ; ++i) sum += vals[i] = i;
        Key key("triv",0);
        DataFrame* df = DataFrame::fromArray(&key, &kv, SZ, vals);
```

assume we have a kvstore setup above, fromArray will store the double array to a dataframe,
under the given kvstore, with the `key("triv",0)` .

DataFrame::fromScalar(key, kvstore, field value);

```
    Key check("ck",0);
    double sum = 0;
    for (size_t i = 0; i < SZ; ++i) sum += vals[i] = i;
    DataFrame::fromScalar(&check, &kv, sum);
```

assume we have a kvstore setup above, fromScalar will store a single value to a dataframe,
under the given kvstore, with the `key("ck",0)` .

```
add_column(Column col, string name):
```

this method will add the column to the current schema with the given name.

```
get_int(size_t col, size_t row);
get_string(size_t col, size_t row);
get_bool(size_t col, size_t row);
get_double(size_t col, size_t row);
```

these methods will return the value for that position at the place in the dataframe.
Wrong type,column or row out of bound is undefined….

```
set(Row r, Column c, int val);
set(Row r, Column c, double val);
set(Row r, Column c, bool val);
set(Row r, Column c, string val);
```

These methods will set the primitive type of value on that position.

```
get_col(string col);
get_row(string row);
```

These methods will return the col or row from the schema with the given name.

```
nrows();
ncols();
```

return the number of rows and number of columns

```
add_rows(Row r);
```

add the row at the end of the dataframe.

```
fill_row(size_t idx, Row r);
```

set the idx's row with the provided row. If the type is wrong, results are undefined.

```
map(Rower r);
```

map a function by using Rower, and edit the dataframe row by row.

```
pmap(Rower r);
```

using multi-thread to do the map's work, and merge the results.

```
filter(Rower r);
```

The Rower gives a function to check if the data is proved by the condition, if it is not, the data will not be used in the new dataframe.

```
print()
```

print the dataframe

KVStore:

the methods under KVstore:

```
get(Key k);
put(Key k, Dataframe v);
waitAndGet(Key k);
```

Asssume we are storing

Node1: k1,v1
Node2: no key and value;
Node3: k2,v2; k3, v3

if we are currently on node 1:
get(k1) -> v1;
put(k4,v4) will input the value under the node that home provided;
Because `(k2,v2)` is not on the current node, if we want to get `v2`, we need to call `waitAndGet(k2)` to get the value from Node3.

Application (will also be editing later on)

The method that application should have:

```
run_();
this_node();
```

run_() will start to run each node;
this_node() could tell the index of the current node();

Demo

use case is provided as the assignment says…

Serializable

Here is an example of a simple primitive type use-case:

```
    float first = 3.9;
    bool second = true;
    int third = 1000;

    Serializer s;
    s.write((float)3.9);
    s.write((bool) true);
    s.write(1000);

    char* buf = s.get();
    size_t buf_len = s.length();
    Deserializer ds(buf, buf_len);

    t_true(std::abs(ds.readFloat() - first) < 0.00001);
    t_true(ds.readBool() == second);
    t_true(ds.readInt() == third);
```

and the `char*` allows the system to convert them back to their originial type, such as float, boolean and integer. Serializer&Deserializer could also work for any customized class objects, for more detailed please look at the code.

for more information please take a look at our test cases.

SorerDataframe Adapter

```
    std::shared_ptr<DataFrame> parse_file(const std::string& filename);
```

The input is the file name, and outputs a shared pointer dataframe, and what it does is it will initialize the schema and parse the file row by row.

Demo

This is our implementation of the Demo application requested in M1. There are three functionalities for nodes: `producer`, `counter`, and `summarizer`. It can also be told to run the server instance. The functionalites are as follows:

1. Producer

- Stores a dataframe constrcuted from an array of doubles, and store a dataframe of the sum of previous DataFrame.

1. Counter

- Waits and retrieves the large dataframe of doubles from the producer. After finding it, sum the large dataframe and store that sum as a local dataframe.

1. Summarizer

- Retreive the sum of the producer (the expected value for the sum of the initially constructed dataframe), and the sum the counter found, and compares them to ensure they both have the same value and no corruption on the network occured.

WordCount

This is our implementation fo the WordCount application. To implement an app could count the word of the document, we implement a CounterRower.

CounterRower:
Make sure the type of the column in this row is String, then counts the word in each row.

Linus

This is the application for Linus which is asked in M5.
The functionalities is implemented by using the different rowers, and iterate through each name or id to find the next degree.

1. IntSetGenerator
   Rower that operates on a dataframe containing integers,
   and adds all integers in that dataframe to a set (preventing
   duplicates), and returns said set when finish_set() is called.

2. UnorderedFilter
   Abstract class that creates a dataframe of values from the
   given set of values.

2a. UUIDsToProjectsFilter :
Given a dataframe of user ids stores a set of those ids. When mapped over
the commits dataframe constructs a unordered dataframe containing the PIDs
that those users have worked on.

2b. ProjectsToUUIDsFilter :
Given a dataframe of pids, stores a set of those ids. When mapped over the
commits dataframe, constructs an unordered dataframe containing the UUIDs of
users who have worked on those projects.

2c. UUIDsToNamesFilter :
Given a dataframe of uuids, stores a set of those ids. When mapped over the
users dataframe, constructs an unordered dataframe containing the user names
of the users in the set.

Open questions:

Current Known Issues:

- All memory issues that valgrind detect appear to originate in the STL `unordered_map` and `unordered_set` implementations.
- There appears to be a race condition, as sometimes the Demo will complete sucessfully, sometimes it will compute the wrong result, and sometimes it will fail with a Segfault.
- Sometimes threads appear to try to join themselves.
- Teardown results in failure with memory issues.

Status:

Week 1:
This week we set up the directories as required and wrote tests for key/value store.

Week 2:
Clean up the code for network, now network is working.
Implemented the KVStore on one node.
Adding a sorer functionality.
Wrote a FromArray inside columns, and the tests for this part.
Catching up the report.
added sorer from another group
used sorer to read dataframe and wrote tests
use the optional to handle primitive type with null

Week 3:
test network;
recreate serializable;
rework packet;
adding tests;

implement summer producer counter in application;
change Key's home_ from `size_t` o `sockaddr_in`
wrote from Fromscallar (single value);
rewrote client；
Added teardown and demo
Seems to work sometimes, but not always.

week 4:
finish network

week 5:
implement linus
add more test cases..