Introduction

The EU2 system is distributed system that will be used for large scale data analysis. The system will allow queries on the data. It will be written in CwC, with some C++ as needed. Each node in the distributed system will hold its own data. Data will be loaded from the SoR file format.

Architecture

- There is a central server that all of the nodes talk to to figure out what nodes are connected and where they are reachable
- Each node runs an instance of an Application class. This is where the application specific logic lives
- Each node also runs an instance of the KBStore. This talks to the central server to get the list of all of the nodes. If the application requests data that is not on the node that the KBStore is running on, it will use the data it received from the central server to talk to the node that has the data.
- KVStore wraps KBStore to provide dataframe specific logic
- When a dataframe is stored in the KVStore, the columns of it are split into chunks and the chunks are distributed along the connected nodes. When a dataframe is requested, the list of column chunks and the schema is pulled. As values are read from the dataframe the chunks for the columns are lazily loaded from the network.

Implementation

Classes

`KVStore` - Allows for putting and getting data frames

`KBStore` - Abstracts away the distributed nature of the key store - Manages concurrency and networking - Stores raw bytes under keys

`Key` - Stores the home node and the name of an entry in the distributed key store

`Application` - Provides a 'main' like run method - Handles teardown of the entire system once root node is completed

`Dataframe` - Stores columns of data - Has convience methods to create a dataframe from a single value or an array of values - Ensures that columns adhere to a schema - Allows reading / writing data

`Server` - Central rendezvous server for nodes - Accepts incoming connections and broadcasts IPs and ports of currently connected clients

Use cases

```
 /// A simplification of the Linus application
class Linus : public Application {
```

```
public:
  /** Compute DEGREES of Linus.  */
  void _run() override {
    readInput();
    for (size_t i = 0; i < DEGREES; i++) step(i);
  }

  void readInput() {
    Key pK("projs");
    Key uK("usrs");
    Key cK("comts");
    // Read in all of the data and store it
    if (this_node() == 0) {
        pln("Reading...");
        DataFrame::fromFile(PROJ, &pK, &kv);
        DataFrame::fromFile(USER, &uK, &kv);
        DataFrame::fromFile(COMM, &cK, &kv);
        DataFrame::fromScalar(new Key("users-0-0"), &kv, LINUS);
    }
 }

  void step(int stage) {
    p("Stage ").pln(stage);

    // Load the data and
    DataFrame* newUsers = kv.waitAndGet(uK);
    Set delta(users);
    SetUpdater upd(delta);

    // Perform the operations on data found locally
    ProjectsTagger ptagger(delta, *pSet, projects);
    commits->local_map(ptagger);

    // Merge the data together
    merge(ptagger.newProjects, "projects-", stage);
    merge(utagger.newUsers, "users-", stage + 1);
  }

  void merge(Set& set, char const* name, int stage) {
    if (this_node() == 0) {

      // Merge all of the data from all of the nodes
      for (size_t i = 1; i < NUM_NODES; ++i) {
        DataFrame* delta = kv.waitAndGet(nK);
        SetUpdater upd(set);
        delta->map(upd);
        delete delta;
      }
```

```
      SetWriter writer(set);
      DataFrame::fromVisitor(&k, &kv, "I", &writer);
    } else {

      // Send the data back to the main node to have it merged
      SetWriter writer(set);
      DataFrame::fromVisitor(&k, &kv, "I", &writer);
      DataFrame* merged = kv.waitAndGet(mK);
    }
  }
}; // Linus
```

SOR files can be loaded by using an instance of the `Schema` class and using the build method.

Open questions

There are no open questions at this time.

Status

We have implemented the dataframe adapter to read the SoR file format, we have written the dataframe class and we have written the client-server communication. The only thing that remains to be done for the current requirements is to implement the keystore network communication. This should be aproximately 15 hours of work.