Milestone 4 By: Angela Wang & Rebecca Mattie

Introduction:

The eau2 system is a distributed key-value (KV) store for big data analysis with a toolbox of machine learning algorithms running on GPUs. Values (either Strings, numbers, or booleans) can be added to the KV store and retrieved from the KV store. Although this information is not stored directly as a Dataframe, the data can be interpreted in the context of a Dataframe, and then functions can be performed over the Dataframe.

Architecture:

The system will be designed in three layers. The bottom layer is a distributed KV store running on each node and each store has part of the data. The KV stores talk to each other to exchange data when needed. All of the networking and concurrency control is hidden in the bottom layer. The level above the bottom layer provides abstractions like distributed arrays and dataframes and these are implemented by calls to the KV store. The application layer is where users write code that sits on top of all this. This code can include functions to analyze the data stored in the KV store.

Implementation:

Layer 1: Networking

Each of the nodes are objects of the `NetworkingDevice class`. The first node to start up will specifically be an object of the `Server` class, and the rest of the nodes will be objects of the `Client` class. `Server`s and `Client`s have much of the same functionality, and share many methods to set up and maintain connections between other NetworkingDevices, but the main difference between the two classes is that all of the Clients check in with the Server so that the Server can send update messages out to all of the Client nodes with information about how to contact all of the other nodes. There will be a KV store on each node. Each node will hold a part of the total data. Whenever one of the nodes needs data that is stored on another node, they will talk to each other to share the data. The data is serialized and deserialized using the `Serial` class so that it can be sent easily across the network. Each of the keys is composed of a string (the actual key, for searching) and a home node (that tells which node the actual data is stored on). The KV store supports functions such as `put(k,v)`, `get(k)`, and `getAndWait(k)`.

Layer 2: Abstractions

`Dataframe`: `The DataFrame` class represents the dataframe and has fields: `ObjectArray* columns_`, which stores the columns of the dataframe, `Schema* schema_`, which represents the schema of the dataframe, `size_t num_rows_`, which represents the number of rows currently added to this dataframe, `size_t num_cols_`, which represents the number of rows currently added to this dataframe. `DataFrame` supports `add_column`, `get_column`, `get_int`, `get_bool`, `get_float`, `get_string`, `fill_row`, `add_row`. The serialize function is overwritten in `DataFrame` to include the schema, the number of rows and the actual dataframe represented as an array of columns to make it easier to deserialize in the future.

`Schema:` The Schema class represents the schema of the dataframe; that is it tells how many rows and columns are in the dataframe, what the names are of these rows and columns, and what the types of each column are. The fields include: `size_t numCol_`, which stores the number of columns in the dataframe, `size_t numRow_`, which stores the number of rows in the data frame, `StringArray* cTypes_`, which is a list of the types of each column in the dataframe. Operations on the schema include: copying the schema, adding columns, adding rows, returning names, indices, and numbers of rows and columns, returning types of columns, and telling whether a schema and an object are equivalent.

`Column:` The Column class represents the columns in the dataframe. Similar to the `Array` class, the `Column` class is an `Object` that is never directly used; instead, there exist classes that extend the class called `BoolColumn`, `FloatColumn`, `IntColumn`, and `StringColumn`. Unlike the `Array` class, there is no `ObjectColumn` since these columns represent the actual columns in the dataframe, and the dataframe does not allow columns of any type other than `Bool`, `Float`, `Int`, and `String`, so there will never be any need for a column that holds `Object`s. The `BoolColumn`, `FloatColumn`, `IntColumn`, and `StringColumn` classes each contain a field `arr_`, which is an array that matches the type of element this column is meant to hold. For example, the `BoolColumn` contains an `BoolArray** arr_`. The `Column` field `len_` refers to the number of elements in this `Array`. The `Column` field `num_elements` refers to the number of elements in the `Column`. These numbers can be, but do not have to be, equal. Instead of directly storing all of the elements in the `Column`, `arr_` stores a list of pointers to other `Array`s of fixed size. These fixed-size arrays store the `Column`'s elements directly. In this way, when we increase the size of the `Column`, we don't have to copy over any payload, we only have to copy over the pointers to the fixed-size arrays. This is better, not only because we're copying over pointers instead of payload, but also because instead of having to do one copy for every element in the column, we do one copy for (number of elements in the column) divided by the size of the fixed-size arrays, which improves the speed of adding elements to the dataframe. Columns themselves support operations of `push_back`, `remove`, `get`, `size`, which returns the size of the `Column`, and `as_(type)` which returns the `Column`.

`Row`: The `Row` class represents a row in the dataframe. Although the `Row` class does not appear as a field in the `DataFrame` class, it can be treated like a real row from the dataframe. `Rows` have fields: `Schema* schema_`, which stores the schema for the dataframe the row is to be part of, `size_t index_`, which stores the row index this row will be in the dataframe, `ObjectArray* elements_`, which is an `Array` of the elements in the row. They support operations to set elements to values, to set the index of this row, to return the index of this row, to get the element at the specified index, to return the type of the element at the specified index, to return true if all fields are set.

`Array`: The `Array` class is an `Object` with a capacity `cap_` representing the maximum possible capacity for this `Array`. It also has a length `len`, representing the length, or number of elements, in this `Array`. The `Array` class itself is never used; instead, we use classes that extend it `BoolArray`, `FloatArray`, `IntArray`, `StringArray`, and `ObjectArray`. The first four classes are `Arrays` that only hold the specific type mentioned in their name. `ObjectArray`s are more broad, and can hold any type of object. Each of these classes have a field `arr_`, which represents the list of elements in the array. The `Array` classes support operations such as `add`, `remove`, `index_of`, `get`, and `equals`.

`Key`: The `Key` class represents the key to use in a `KeyValueStore`. A key consists of a string and a home

node, represented as a number. The string is the actual key used for searching, the home node tells which KV store owns the data.

`KeyValueStore`: The `KeyValueStore` class represents a store for a singular node. Each `KeyValueStore` has a dictionary mapping Keys to serialized blobs, which are serialized `DataFrames` in our case. `KeyValueStore` supports `put(k,v)`, `get(k)` and `getAndWait(k)`.

`KDStore`: the `KDStore` class is a layer of abstraction between a `DataFrame` and a `KeyValueStore`. It contains a `KeyValueStore`, and it supports the operation get on it. It deserializes the serialized blob that `KeyValueStore`'s get method returns and it returns a `DataFrame`.

Layer 3: Application

An `Application` class was created. It has a `KeyValueStore kv` to store the data the application will process, a `KDStore kd` to store `kv`, and a `size_t index` to represent the node it's on. Users can write applications that are objects of this class.

The KV store supports functions such as put(k,v), get(k), and getAndWait(k).

#

ayer 2: Abstractions Dataframe: The DataFrame class represents the dataframe and has fields: ObjectArray* columns_, which stores the columns of the dataframe, Schema* schema_, which represents the schema of the dataframe, size_t num_rows_, which represents the number of rows currently added to this dataframe, size_t num_cols_, which represents the number of rows currently added to this dataframe. DataFrame supports add_column, get_column, get_int, get_bool, get_float, get_string, set (int, bool, float, string), fill_row, add_row, map, pmap, filter, and print. The serialize function is overwritten in Dataframe to include the schema, the number of rows and the actual dataframe represented as an array of columns to make it easier to deserialize in the future. Schema: The Schema class represents the schema of the dataframe; that is it tells how many rows and columns are in the dataframe, what the names are of these rows and columns, and what the types of each column are. The fields include: size_t numCol_, which stores the number of columns in the dataframe, size_t numRow_, which stores the number of rows in the data frame, StringArray* cTypes_, which is a list of the types of each column in the dataframe. Operations on the schema include: copying the schema, adding columns, adding rows, returning names, indices, and numbers of rows and columns, returning types of columns, and telling whether a schema and an object are equivalent. Column: The Column class represents the columns in the dataframe. Similar to the Array class, the Column class is an Object that is never directly used; instead, there exist classes that extend the class called BoolColumn, FloatColumn, IntColumn, and StringColumn. Unlike the Array class, there is no ObjectColumn since these columns represent the actual columns in the dataframe, and the dataframe does not allow columns of any type other than Bool, Float, Int, and String, so there will never be any need for a column that holds Objects. The BoolColumn, FloatColumn, IntColumn, and StringColumn classes each contain a field arr_, which is an array that matches the type of element this column is meant to hold. For example, the BoolColumn contains an BoolArray** arr_. The Column field len_ refers to the number of elements in this Array. The Column field num_elements refers to the number of elements in the Column. These numbers can be, but do not have to be, equal. Instead of directly storing all of the elements in the Column, arr_ stores a list of pointers to other Arrays of fixed size. These fixed-size

arrays store the Column's elements directly. In this way, when we increase the size of the Column, we don't have to copy over any payload, we only have to copy over the pointers to the fixed-size arrays. This is better, not only because we're copying over pointers instead of payload, but also because instead of having to do one copy for every element in the column, we do one copy for (number of elements in the column) divided by the size of the fixed-size ar themselves support operations of push_back, remove, get, set, size, which

Open Questions:

We assume that when the first node starts up, it holds all of the data from the KV Store, and then when more nodes are added to the system, the data from the KV Store is distributed between them. How is this done without large overhead? When a second node is added, the data can be split in half. But when a third node is added, in order to split the data evenly in 3 parts, there will be a lot of overhead.

Status:

Currently, the prototype supports reading in data from a specific format, building a dataframe, and doing trivial operations on that dataframe. It also supports storing dataframes in nodes and returning them. It can also support multiple nodes running at once, though at the moment they need to be run in different threads so that they can all share memory rather than running on the network. It supports multiple applications such as the wordcount application.

Remaining work:

Layer 1: Integrate NetworkingDevice classes and KVstores Integrate Network.h and the Serial class Build network so that nodes can send dataframe information over the network instead of all sharing the memory. class itself is never used; instead, we use classes that extend it: BoolArray,

Arrays that only hold the specific type mentioned in their name. ObjectArrays are more broad, and can hold any type of object. Each of these classes have a field arr_, which represents the list of elements in the array. The Array classes support operations such as add, remove, index_of, get, and equals.

Key: The Key class represents the key to use in a KVStore. A key consists of a string and a home node, represented as a number. The string is the actual key used for searching, the home node tells which KV store owns the data.

KeyValueStore: The KeyValueStore class represents a store for a singular node. Each KVStore has a dictionary mapping Keys to serialized blobs, which are serialized DataFrames in our case. KeyValueStore supports put(k,v), get(k) and getAndWait(k).

KDStore: the KDStore class is a layer of abstraction between a DataFrame and a KeyValueStore. It contains a KeyValueStore, and it supports the operation get on it. It deserializes the serialized blob that KeyValueStore's get method returns and it returns a DataFrame.

Layer 3: Application An Application class was created. It has a KeyValueStore kv to store the data the application will process, a KDStore kd to store kv, and a size_t index to represent the node it's on. Users can write applications that are objects of this class.

Use Cases:

class Demo : public Application { public: Key main("main",0); Key verify("verif",0); Key check("ck",0);

Demo(size_t idx): Application(idx) {}

void run_() override { switch(this_node()) { case 0: producer(); break; case 1: counter(); break; case 2: summarizer(); } }

void producer() { size_t SZ = 100*1000; double* vals = new double[SZ]; double sum = 0; for (size_t i = 0; i < SZ; ++i) sum += vals[i] = i; DataFrame::fromArray(&main, &kv, SZ, vals); DataFrame::fromScalar(&check, &kv, sum); }

void counter() { DataFrame* v = kv.waitAndGet(main); size_t sum = 0; for (size_t i = 0; i < 100*1000; ++i) sum += v->get_double(0,i); p("The sum is ").pln(sum); DataFrame::fromScalar(&verify, &kv, sum); }

void summarizer() { DataFrame* result = kv.waitAndGet(verify); DataFrame* expected = kv.waitAndGet(check); pln(expected->get_double(0,0)==result->get_double(0,0) ? "SUCCESS":"FAILURE"); } };

Status: Currently, the prototype supports reading in data from a specific format, building a dataframe, and doing trivial operations on that dataframe. It also supports storing dataframes in nodes and returning them. It can also support multiple nodes running at once, though at the moment they need to be run in different threads so that they can all share memory rather than running on the network. It supports multiple applications such as the wordcount application. There are no memory leaks in the code!

Remaining work: Layer 1: Integrate NetworkingDevice classes and KVstores Integrate Network.h and the Serial class Build network so that nodes can send dataframe information over the network instead of all sharing the memory. Complete linus application.