EAU2 Application
Steven Horn & Arthur Armani

Introduction:

The eau2 application will be used by customers to query large data frames for results in order to analyze big data. It will be able to read large (100+ GB) schema-on-read files and load the data into a data frame using distributed arrays. The data frame will be stored in several underlying nodes hidden in the network layer that use a distributed key-value store and communicate with one another to exchange chunks of data as necessary. It will employ a 3-layer architecture (network - data frame - customer-facing) described below.

Architecture:

The system will consist of three levels.

The first level will be the network component that contains many nodes divvying up the data in a distributed key value store. This network of nodes can interact with one another to exchange or retrieve data.

The second level will consist of the data frames and distributed arrays to store the values.

The third layer will be the customer facing layer. A data analyst can input a query and receive an answer based on computations that happen in the previous two layers.

Implementation:

The Sorer class will handle reading in data from a schema-on-read file and determining a DataFrame schema. Four types of data will be supported: booleans, integers, strings, and doubles. Data will be read only once. Basic operations happening within Sorer would include determining the overall size of passed in data and splitting it up into smaller dataframes, retrieving a schema from data, and handling rows with missing values by discarding them.

- DataFrame* generate_dataframe(Schema s)

The DataFrame class will have a particular schema derived from the result of running Sorer on the data. Basic operations will include returning the schema, getting the number of rows and columns, adding rows and columns, getting and setting values, map to run through each row, and printing. A DataFrame can be constructed from an array or a scalar value as well. Upon creation a DataFrame is linked as the value to a Key and Stored in a KDFMap. The DataFrame has a Key-Chunk Map (KVStore) which contains chunks/portions of columns. The KVStore knows which chunk belongs to which node in the network, thus it is our distributed array.

- size_t nrows(), size_t ncols()
- void add_row(Row r), void add_column(Column* col)
- const char* serialize(DataFrame* df), DataFrame* get_dataframe(const char* str)
- DataFrame* from_array(Key* key, KDFMap* kv, KVStore* kc, size_t sz, Array* from), DataFrame* from_scalar(Key* key, KDFMap* kv, KVStore* kc, <type> val)
- int get_int(size_t col, size_t row), bool get_bool(...), double get_double(...), String* get_string(...)
- void set(size_t col, size_t row, int val), set(..., bool val), set(..., double val), set(..., String* val)
- void finalize_all()
- void map(Rower r), void print()

The primary KV Store is a <Key, String> mapping where keys have a string value (name) and an index indicating which node the value belongs to, and the String is a serialized object, either a DataFrame or a Chunk. The KVStore stores chunks internally for columns so that an entire DataFrame is not stored in one node's memory, and it stores DataFrames for the application layer's use. It is also connected to a network. The network consists of various client nodes that are able to register with a server node and then exchange messages with other nodes directly. They will each be responsible for part of a distributed key-value store, so exchanging chunks of data will be necessary. These chunks of data are represented by the Chunk object and have a mapping to a home node. Data frames and messages will be able to be serialized to allow for easy travel through sockets. We can add pairs to the KV store with the put method.

- size_t index(), sockaddr_in getMyIP(), size_t port()
- void server_init(), void client_init()
- void init_sock()
- void send_m(Message* m), Message* recv_m();
- void put(Key* k, Chunk* value), put(Key* k, DataFrame* value)
- DataFrame* get(Key* k), DataFrame* getAndWait(Key* k)

The application will consist of customer-facing code that allows users to enter queries. The results of queries will be output to the user’s console (or whatever front-end they are accessing the eau2 application from).

- <return type TBD> sendQuery(String query)

Use cases:

We are not quite ready for the Degrees of Linus application use case, so we will instead show off use-cases of what we have now, which is basically reading from a file to create a dataframe and mapping a key to a dataframe.

Given a sor data file data.sor:

```
Sorer s(“data.sor”);

DataFrame* df = s.generate_dataframe();

df->print();
```

It is that simple; the Sorer object takes the data file and parses it to infer a schema. When the generate_dataframe() method is called, the Sorer will use the schema to create a data frame and then parses the entire file to populate the data frame with rows that match the schema. You can also give the Sorer other parameters to dictate where to start reading the file and how many bytes to read in the file:

```
Sorer s(“data.sor”, 100, 1000);     will read the 1000 bytes of data.sor that
come after byte 100

Sorer s(“data.sor”, 100, -1);          will read the entire file starting at byte 100

Sorer s(“data.sor”, 0, -1);    will read the entire file starting at byte 0 (same
as Sorer s(“data.sor”))
```

In our second milestone, we have added the distributed key-value store.

Given a key in the form of some string, we can retrieve, set, or remove a dataframe mapping:

```
DataFrame* df = kv.get(key);

SDFMap* m = new SDFMAP();

String* key1 = new String("FRAME1");

m.put(key1, new DataFrame(new Schema("")));

m.get(key1);

m.getAndWait(key1); // blocking

m.remove(key1); // removes mapping and returns DataFrame
```

In our third milestone, we have added a pseudo network running on threads. This milestone demonstrated that one node (thread) can see updates to the KV store made by another node (thread).

```
KDFMap* masterKV = new KDFMap(); // KV Store to distribute among nodes

DemoThread d1(0, 100, masterKV); // Creates a demo that produces a data frame under key "main"

DemoThread d2(1, 200, masterKV); // Creates a demo that produces a data frame under key "confirm" with the sum of the "main" data frame

DemoThread d3(2, 300, masterKV) // Creates a demo that checks whether the sum of "confirm" is equivalent to the sum of another data frame under the key "validate";
```

In our fourth milestone, we have added a real network that relies on communication over sockets. This network is capable of registering clients, sharing a directory of nodes, sending serialized messages between nodes, and thus sharing dataframes/information whenever necessary. We have not quite linked the distributed KeyValue stores to the network as this presented a significant challenge for us this week.

```
./compiledName 1 3 127.0.0.2 8084 127.0.0.1 8083 &
./compiledName 2 3 127.0.0.3 8085 127.0.0.1 8083 &
./compiledName 0 3 127.0.0.1 8083 127.0.0.1 8083
```

The above code would create a network with three nodes, the first two being clients, and the last being the server.

```
Network* n = new Network(node_info, num_nodes);

n->server_init(); OR n->client_init();
```

The above code allows for the creation of a network and the initialization of the type of node.

In the fourth milestone, the way we run the compiled executables is the same as above, but creating the network is very different. Now, the network is directly tied to the KVStore, which has also been reworked.

Now, we create a network/kvstore by doing:

```
kv = new KVStore(node_info, num_nodes, this_node, server_ip_str, server_port);

 NetworkThread n1(kv);
```

The KVStore takes in all network information and sets up the network. We then create a NetworkThread, which launches a while loop so that the node can listen for incoming requests. After the above code, you can run your application code with the created KVStore.

Open Questions:

Status:

[Week of March 9] Our project is currently in the planning and preparation stage. Technical debt from the individual components such as the networking layer, the dataframe, and the schema-on-read parser has been resolved. We are able to read from large files and store reasonably sized dataframes. Operations can be carried out on the dataframes such as summation and filtering. At the moment, the network layer is not connected to a distributed Key-Value store. We plan to connect all the individual parts together to form the three layer architecture in the coming week. After that, we will likely need to make performance improvements and do thorough load testing with multi-gigabyte data sets.

[Week of March 16] Our project now has the Key-Value store with operations such as get, put, remove, and wait_and_get, taking in a key and returning a dataframe. We have improved the performance of our system significantly, halving execution time. In order to arrive at this performance boost, we removed the need to copy strings in our columns and arrays. We have changed one of the datatypes - floats - to now be doubles. In getting the trivial example to work for numbers with decimals, we noticed that floats were too small and lost precision after a certain threshold. Thus, doubles allowed us to maintain precision. In changing the datatypes, we had to modify our implementation of schema, columns, serialization, and arrays. The trivial example works for integers as well. We discovered a memory management bug with valgrind where empty schemas (character arrays with a null terminator) were not being appended to correctly. This bug was fixed by splitting our add_column method in schema into two cases (empty schema/non-empty schema).

[Week of March 23] We got the Demo example working correctly by distributing the KV store. We currently do this using threads instead of our network layer, and by creating a common KSMap that is shared between each Demo instance. We also cleaned up our file structure a bit by removing unnecessary files. We added unit tests into our testing suite, along with keeping tests from each milestone so that are tests are more cumulative and comprehensive. We still need to switch from using threads to using our network. We will first need to create serialization methods for DataFrames, then alter the KV store to communicate with the network layer. After that, we will have to figure out how to make columns/arrays distributed instead of just the KV store.

[Week of March 30] We got the network functioning. Clients are able to register and the server can message clients with a directory. We spent a majority of our time refactoring our code with Chunk and KVStore so that each node did not have to be responsible for the entire data frame. We added some tests and completed serialization for DataFrames/Chunks/Arrays. We faced challenges with cyclic dependencies that forced us to refactor another significant portion of the code. We did not quite get the word count example running as we were stuck at how to insert the DataFrame into the network.

[Week of April 6] We broke the network. We needed to overhaul our implementations of both the network and of the KVStore, which took a lot of refactoring in much of the codebase. We have everything pre-milestone 3 working, including a bunch of separate unit tests. We also resolved a lot of issues from previous milestones. We are currently at a point where the network is almost functioning again how we want it to. We are currently able to send data frames and chunks over the network, but we are running into an issue when a key-value pair isn't present yet. Because of this, the Demo example is not quite working, let alone Word Count or Linus. We fully intend on getting all functionality working before our final code walk. If you run make test, you will see the results of all unit tests and milestones 1 and 2. We have milestone 3 networking stuff attempting to run, but it fails after a little bit and gets caught in a loop. We wanted to keep it in to show how far we have gotten with the networking, but make sure that you CTRL+C to stop the loop once the feedback in the terminal stops coming in.