Milestone 3

Introduction

The eua2 system is a system that allows users to run applications involving dataframes built on a distributed key-value store. The system can be used to map and filter through large dataframes that do not fit in the memory of a single machine.

Architecture

Network and Key-Value Store Layer

This layer is a distributed KV store running on multiple nodes. Each KV store node has part of the data, and the KV store nodes talk to exchange data when needed. All of the networking and concurrency control is hidden here.

Server and Client startup/setup

To setup the system, the registration server must first be running. Clients (KV Store nodes) will connect to the server. The server keeps track of all registered clients and broadcasts all registered clients to each of the registered clients. This allows the clients to discover all other clients in the system.

Distributed DataFrame and Array Layer

This layer provides abstractions like the distributed dataframe and arrays (columns).

Distributed DataFrame and Column

A DataFrame represents multiple columns of data that each hold a specific type (`String`, `double`, `int`, `bool`). A column holds chunks that each hold actual values. A dataframe object does not actually hold the values in the columns directly. There is a lookup process when accessing data inside the dataframe.

A dataframe can hold multiple columns and each column holds keys to its chunks that are stored within the key-value store. When accessing a value inside a dataframe, the dataframe finds the correct column and finds the correct chunk inside that column. Then, the dataframe looks for the chunk in the key-value store by using the key for the chunk and gets the value it wants from that chunk.

Application Layer

The user can operate on the dataframes. (Queries, Machine Learning, AI)

Implementation

Networking

Server

A `Server` class handles registraion of `Client` objects. It holds a directory of clients and can broadcast the registered clients to all registered clients.

Client

A `Client` class holds the ip and port of all other clients on the network. The client registers with the registration server on startup and discovers the other clients with the help of the server.

Network Protocol

Get a Value from another node using a Key

1. Client 1 sends header (message type and payload length) to client 2
2. Client 2 sends ready message to client 1
3. Client 1 sends a serialized key to client 2

if key exists on client 2:

4. Client 2 sends the header for the response message to client 1
5. Client 1 sends ready message to client 2
6. Client 2 sends the serialized value for the key to client 1
7. Client 1 sends ack to client 2 and closes connection
8. Client 2 recieves ack and closes connection

if key does not exist on client 2:

4. Client 2 sends ack and closes connection
5. Client 1 recieves ack and closes connection

Get and Wait for a Value from another node using a Key

1. Client 1 sends header (message type and payload length) to client 2
2. Client 2 sends ready message to client 1
3. Client 1 sends a serialized key to client 2
4. **Client 2 waits until the key appears in its key value store**
5. Client 2 sends the header for the response message to client 1
6. Client 1 sends ready message to client 2
7. Client 2 sends the serialized value for the key to client 1
8. Client 1 sends ack to client 2 and closes connection
9. Client 2 recieves ack and closes connection

Put a Key Value pair on another node

1. Client 1 sends header (message type and payload length) to client 2
2. Client 2 sends ready message to client 1
3. Client 1 sends a serialized key and value together to client 2
4. Client 2 sends ack to client 1 and closes connection
5. Client 1 recieves ack and closes connection

KVStore

Key

A `Key` object is used as key that is provided to a K/V store. A `Key` object contains a `String` object and a node index. The node index determines which key-value store node the value can be found on and the string key is what is used to find the value. `Key` is serialized to the format `<node index><key name(null terminated)>`.

Value

A `Value` object is what a `Key` object is associated with. It holds a `char*` byte array of serialized data and a `size_t` representing the size of the data.

KeyValueStore

The `KeyValueStore` class manages associations between `Key` objects and `Value` objects. A `KeyValueStore` on one process knows about `KeyValueStore` objects on other nodes and can access `Value` objects on other nodes. It is also possible to put `Value` objects into `KeyValueStore` objects on other nodes. `KeyValueStore` objects each hold a `Client` object that is used for network communication with other `KeyValueStore` objects.

Putting a key and value:

```
void put(Key* k, Value* v)
```

Getting a value, returns `nullptr` if `Key` does not exist.

```
Value* get(Key* k)
```

Getting a value, blocks/waits until `Key` exists if it does not exist.

```
Value* getAndWait(Key* k)
```

DataFrame

Chunk

A `Chunk` represents a subset of a `Column` Object. A `Chunk` always holds a constant number of values. A `Chunk` is a `Value` object that holds a subset of the `Column`. The `Chunk` is deserialized based on the type of the `Column` it is a part of.

The full key name to a chunk will have the format: `<dataframe_name>:<column_num(hex)>:<chunk_num(hex)>`. Example: `DATAFRAME_NAME:0x123ADF:0x321FDA`.

Column

A `Column` is a single column of a `DataFrame` Object. A `Column` can hold values of one of the following types:

- `String` object
- `double`
- `int`
- `bool`

A `Column` object holds `Key` objects that are associated with `Value` objects representing `Chunks` that hold the data in the column. A `Column` is serialized with the format `<column type><column length><column name(null terminated)>[<key>...]`.

Distribution Strategy

Each `Column` knows the number of nodes with key-value stores that are available. Columns distribute their chunks to the nodes with even distribution by sending chunks to the node with index equal to the chunk index modded by the total number of nodes. ie) If there are three nodes, then starting with the 0th chunk, every third chunk goes to the first node. This strategy also ensures that the Nth chunk of every column in a dataframe is distributed to the same node. This makes local mapping easier.

Caching

Each column caches a single chunk for both puts and gets to the key value store. The cache can be commited into the key value store when pushing elements onto the column. The cache is effective when fetching elements or putting elements into the same chunk index (e.g. iterating over a dataframe row by row). The cache is not effective if the user is randomly accessing the column.

The column has a boolean flag to indicate if the cached chunk has been mutated. This flag is set when the cached chunk is mutated. If the flag is set when a new chunk is accessed, the cached chunk is commited before the new chunk is loaded. If the flag is not set, then the cached chunk is not committed when being swapped out.

The `StringColumn` has an additional cache that is built on a `String` array. Because strings are not always the same size like the other data types, they cannot be easily serialized and deserialized from `String` objects into `Value` objects. The method of finding where to put strings in a value is by iterating through the entire value buffer of strings that are delimited by null bytes. This operation can get very expensive if the value buffers are very large. To prevent this find operation on each get and push, `String` objects are stored in an array cache and the entire array is serialized at once when needed.

DataFrame

The `DataFrame` class holds `Key` objects that are associated with `Value` objects representing `Column` objects that hold data. A `DataFrame` is serialized with the format `<dataframe key><num column>[<column1>,<column2>,...]`.

Application

The `Application` class handles interactions with `DataFrame` objects using the `KeyValueStore`. The application is allowed to create `DataFrame` objects, store `DataFrame` objects in the `KeyValueStore` and

retreive `DataFrame` objects from the `KeyValueStore`.

Putting a key and DataFrame:

```
void put(Key* k, DataFrame* v)
```

Getting a DataFrame, returns `nullptr` if `Key` does not exist.

```
DataFrame* get(Key* k)
```

Getting a DataFrame, blocks/waits until `Key` exists if it does not exist.

```
DataFrame* getAndWait(Key* k)
```

Configuration

There needs to be a `config.txt` file in the directory that the application is run from. In the `config.txt` file there should be specifications for the number of clients that will be run as `CLIENT_NUM`, the ip address of this application as `CLIENT_IP`, the size of the distributed chunks as `CHUNK_SIZE`, and the amount of time that the server should stay up in seconds as `SERVER_UP_TIME`. These allow each application to have variable configs.

Example config.txt

```
CLIENT_NUM=3
CLIENT_IP=192.168.1.86
SERVER_IP=192.168.1.86
CHUNK_SIZE=1024
SERVER_UP_TIME=20
```

Use cases

- The user starts several applications on different devices on the same network. The user of the first application would create a dataframe and then all other applications would be able to read the data. Any other user could create DataFrames and share them using the KVStore.
- A User would create their own subclass of Application and override the `run_()` method to get their own code to run. The user's application has access to the KVStore to get and put dataframes. The user's code must use Keys to find the DataFrames, which means the node index of the DataFrame is known.
- Below are some examples of user applications:

```
static const size_t SZ = 100 * 1000;
```

```
class Demo : public Application {
  public:
    Key main;
    Key verify;
    Key check;

    Demo() : Application(), main(0, "main"), verify(0, "verif"), check(0,
"check") { }

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

    // this node will create a dataframe and calculate the sum of the
values and store both in the kvstore
    void producer() {
      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);
    }

    // this node will wait for the dataframe to be created and calculate
the sum of the numbers in the dataframe and store the sum inside the
kvstore
    void counter() {
      DataFrame* v = getAndWait(main);
      double sum = 0;
      for (size_t i = 0; i < SZ; ++i) {
        sum += v->get_double(0,i);
      }
      p("The sum is  ").pln(sum);
      DataFrame::fromScalar(&verify, &kv, sum);
    }

    // this node will compare the sum values created by the producer node
and the counter node
    void summarizer() {
      DataFrame* result = getAndWait(verify);
      DataFrame* expected = getAndWait(check);
      pln(expected->get_double(0,0)==result->get_double(0,0) ?
"Milestone3: SUCCESS":"Milestone3: FAILURE");
    }
};
```

Use case of reading a SoR file into a dataFrame and parallel mapping a function over the dataFrame. The result is verified by comparing the sum of the fibonacci numbers.

```
KVStore kvs(false);
SOR sorer("../../data/data.sor", &kvs);

DataFrame* df = sorer.read();

Fibonacci fib(df);

df->pmap(fib);

assert(fib.get_sum() == 146);

delete df;
```

Status

What the team has:

- Ability to read SoR format and create a DataFrame from SoR files
- Ability to filter, map and parallel map over a DataFrame
- Ability to serialize and deserialize some objects
- Ability to send messages between clients and server
- Unit tests for existing components: DataFrames and Sorer
- Networking protocol
- Manually tested Networking code, and KVStore code
- Ability to serialize and deserialize dataframes
- Create a Key/Value Store class
- Create an application class
- Create a Distributed DataFrame that can be stored across multiple nodes
- Distribute the column chunks evenly across all nodes on the network
- Created chunk cache for Columns
- Created String array cache for StringColumns