Introduction

The eau2 system allows the user to implement applications that can analyse and manipulate data stored in a key/value store. Under the hood, this key/value store is distributed accross a network of nodes.

Architecture

eau2 is a program with three layers. The bottom layer is a network of KVStores running on different nodes that can talk to each other, store serialized data and share it upon request. The second layer is a dataframe which is a read-only table represented as an object. It is made up of rows and columns, with each column containing data of a specific type. Each column is implemented as a distributed array which is an array of keys used to retrieve data distributed accross multiple KVStores. The application layer is the final layer that sits on top of both previous layers. From the application layer, the user can write their programs that analyse and manipulate data.

Implementation

^ `type` is one of `(String*, int, bool, float)`.

KVStore

This class receives keys from the layer above and either serializes data and puts it in its map (from string keys to deserialized data) or retrieves data from the map, serializes it, and returns it. If the key corresponds to data from another KVStore then it will ask that KVStore for the data. The KVStore class also sends its own serialized data blobs to other KVStores upon request. Upon start up, one of the KVStores acts as a server that maintains new node registration on the network. All other nodes are clients.

fields: * `size_t idx_` - The index of the node running this KVStore. * `Map* map_` - A Map object that can map objects to objects. In this case, it will be used to map String containing keys to Strings containing serialized data. * `int* nodes_` - An array of socket file descriptors where the array indices are the indices of the nodes that the sockets are connected to. * `size_t num_nodes_` - The number of nodes in the system

methods: * `void put(Key& k, const char* v)` - Reads the node index from `k`. If the index is equal to the current node's index, it puts serialized data blob `v` into its map at key `k`. Else, it sends a message to the correct node telling it to do so and waits for a Ack confirming it was done. * `const char* get(Key& k)` - Reads the node index from `k`. If the index is equal to the current node's index, it gets serialized data from its map at key `k` and returns it. Else, it sends a message to the correct node telling it to do so and waits for a Reply message containing the data. * `const char* wait_and_get(Key& k)` - Reads the node index from `k`. If the index is equal to the current node's index, it waits until `k` exists in its map, gets serialized data from its map at `k`, and returns it. Else, it sends a message to the correct node telling it to do so and waits for a Reply message containing the data. * `void startup_()` - Starts up the KVStore on the network. Creates a socket that other nodes will connect through. If not the server, it will also set up a socket to the server and send it its IP address and node index in a Register message. Then, it starts monitoring its sockets in a new thread. * `shutdown()` - Shuts down the network node. Closes all sockets and deletes all fields. * `send_to_node_(const char* msg, size_t dst)` - Sends the given serialized Message to the node at the given index. * `void monitor_sockets_()` - Monitors the sockets in an infinite loop, accepts new connnections, receives messages and processes them depending on what kind they are. * If a Register is received, it keeps track of the sender's node index and socket file descriptor. If the node is the server, it also replies with a Directory containing all client IPs and node indices. * If a Directory is received, the client sends Registers to all other clients in the directory, establishing connections with them. * If a Put, Get, or WaitAndGet message is received, the node starts a new thread, calls the corresponding function, and then replies with either an Ack or a Reply.

Key

Object representing a key that corresponds to data in a key/value store.

fields: * `String* key_` - The actual string that will be mapped to data in the KVStore. * `size_t idx_` - The index of the node that holds the KVStore containing the data.

methods: * `String* get_keystring()` - Getter for the key field. * `size_t get_home_node()` - Getter for the idx field.

Vector

An array of objects split into fixed-size chunks. When it fills up, it grows, allocating more memory for new chunks.

fields: * `Object*** objects_` - Array of pointers to chunks containing the data.

methods: * `void append(Object* val)` - Appends the given object to the end of the vector. * `Object* get(size_t idx)` - Returns the object in the vector at the given index.

IntVector

Similar to the Vector class, but holds int primitives instead of objects.

DataType

A wrapper for a DataFrame field.

fields: * `union Type t_` - A union that can hold an int, bool, float, or string. * `char type_` - The type of field that this DataType holds. One of 'I', 'B', 'F', or 'S'.

methods: * `void set_type`^`(type val)` - Sets `t_` to the given value if it has not already been set. Also sets `type_` to the corresponding char. * `type get_type()` - Returns `t_`'s value. If `t_` has not been set, the value is missing so a default value is returned.

Chunk

A wrapper for a fixed size array of DataFrame fields, a unit of the DistributedVector.

fields: * `DataType** fields_` - The array of fields. * `size_t size_` - The number of fields in the array. * `size_t idx_` - The index of this Chunk within the DistributedVector.

methods: * `void append(DataType* dt)` - Appends the given field to the end of the Chunk. * `DataType* get(size_t index)` - Returns the field at the given index.

DistributedVector

A vector of DataFrame fields where each chunk is serialized and put into the KVStore.

fields: * `Chunk* current_` - When fields are being added to the DVector, they are added to this buffer Chunk until it is full, at which point it is serialized, put into the KVStore, and then reset. When fields are being queried from the DVector, this field acts a cache; once it is deserialized, it is kept in memory until a field from a different chunk is requested. * `Vector* keys_` - List of keys that point to every serialized chunk. * `Key* k_` - The key to the Column that owns this DVector. * `bool is_locked_` - A boolean that is set to true when all fields have been added to the DVector.

methods: * `void store_chunk_(size_t idx)` - Serializes `current_` and puts it into the KVStore once it fills up or once the last field is added to the DVector. `idx` is appended to the column's key, and then that key is used to store the chunk and added to `keys_`. * `void append(DataType* val)` - Appends the given field to the end of the DVector as long as it isn't locked. Calls `store_chunk_()` once `current_` is full. * `DataType* get(size_t index)` - Returns the field at the given index. If the chunk containing the field isn't cached, it fetches it from the KVStore, deserializes it, and resets the cache to it. * `void lock()` - Called after the last field is added to the DVector. Call `store_chunk_()` and then sets `is_locked_` to true.

Column

A column in the DataFrame, holds fields of a specific type.

fields: `DistributedVector* fields_` - A vector containing every field in the column. `char type_` - The type of field that this Column holds. One of 'I', 'B', 'F', or 'S'.

methods: * `void push_back(type val)` - Appends the given field to the end of the Column. * `type get_type(size_t idx)` - Returns the field at the given index. * `void append_missing()` - Appends a missing value to the end of the Column. * `void lock()` - Called after the last field has been added to the Column.

DataFrame

Table containing columns of a specific type

fields: * `Schema schema_` - The DataFrame's schema. * `Vector columns_` - List of all of the DataFrame's columns.

methods: * Getters for specific fields (one for each type) * `void add_column(Column* col)` - Adds the given column to the DataFrame. Pads either the given column or every other column with missing fields, whichever's length is smaller. * `void add_row(Row& row, bool last_row)` - Adds the given row to the bottom of the DataFrame. If `last_row` is true, it calls every column's `lock()` method. * `void map(Rower& r)` - Visits every row of the DataFrame. * `void local_map(Rower& r)` - Visits every row of the DataFrame that is stored on the current node. * `DataFrame* filter(Rower& r)` - Builds and returns a new DataFrame containing rows which the given visitor accepted. * `static DataFrame* fromTypeArray(Key* k, KDStore* kd, size_t size, type* vals)` - Static method that generates a new DataFrame with one column containing `size` values from `vals`, serializes the dataframe and puts it in `kd` at `k`, and then returns the DataFrame. * `static DataFrame* fromTypeScalar(Key* k, KDStore* kd, type val)` - Static method that generates a new DataFrame with one column and row containing `val`, serializes the dataframe and puts it in `kd` at `k`, and then returns the DataFrame. * `static DataFrame* fromVisitor(Key* k, KDStore* kd, const char* scm, Writer& w)` - Builds a DataFrame with the given schema using the given Writer visitor, serializes the dataframe and puts it in `kd` at `k`, and then returns the DataFrame. * `static DataFrame* fromFile(const char* filename, Key* k, KDStore* kd, char* len)` - Builds a DataFrame from `len` bytes of the given input file using the Sorer, serializes the dataframe and puts it in `kd` at `k`, and then returns the DataFrame.

KDStore

Middle-man between the Application layer which uses DataFrames and the KVStore which stores serialized blobs of data.

fields: `KVStore kv_` - The current node's KVStore.

methods: * `DataFrame* get(Key& k)` - Gets the serialized DataFrame stored at the given key, deserializes it, and returns it. * `DataFrame* wait_and_get(Key& k)` - Waits until the given key exists in the KVStore, gets the serialized DataFrame stored at it, deserializes it, and returns it. * `void done()` - Called when the application has finished execution. Shuts down the network in the KVStore.

Application (abstract class)

Users will subclass this class in order to write their applications that use the eau2 system. The application will be run on each node in the system. Each node can perform different tasks based on its index.

fields: * `size_t idx_` - index that differentiates one node from another * `KDStore kd_`

methods: * `virtual void run_()` - runs the application * `size_t this_node()` - returns this node's index * `void done()` - Shuts down the entire system once all operations have completed.

Use Cases

The code below builds a DataFrame containing 100 ints and strings, and then calculates the sum of every int in the DataFrame.

```
Schema s("IS");
DataFrame* df = new DataFrame(s);
Row* r = new Row(s);
String* str = new String("foo");
for (int i = 1; i <= 100; i++) {
    r->set(0, i);
    r->set(1, str);
    df->add_row(*r);
}
SumRower* sr = new SumRower();
df->map(*sr);
assert(sr->get_total() == 5050);
delete df;
delete r;
delete str;
delete sr;
```

This code builds a DataFrame with one column of floats, adds it to a KVStore, gets it back from the store, and ensures that the floats' values remain consistent.

```
class Trivial : public Application {
public:
    Trivial(size_t idx) : Application(idx) { }
    void run_() {
        size_t SZ = 1000*1000;
        float* vals = new float[SZ];
        float sum = 0;
        for (size_t i = 0; i < SZ; ++i) sum += vals[i] = i;
        Key key("triv",0);
        DataFrame* df = DataFrame::fromFloatArray(&key, &kv_, SZ, vals);
        assert(df->get_float(0,1) == 1);
        DataFrame* df2 = kv_.get(key);
        for (size_t i = 0; i < SZ; ++i) sum -= df2->get_float(0,i);
        assert(sum==0);
        delete df; delete df2;
    }
};
```

This runs an eau2 application with three nodes * Node 0 builds one DataFrame containing an array of floats and another one containing the floats' sum. It puts both into its KVStore. * Node 1 gets the DataFrame with the array of floats from Node 0's KVStore and calculates its own sum. It puts this sum into Node 0's KVStore as well. * Node 2 gets the two sums from Node 0's KVStore and compares them.

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

    Demo(size_t idx) : Application(idx), main("main", 0), verify("verif", 0),
        check("ck", 0) {
        run_();
    }

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

    void producer() {
        size_t SZ = 1000;
        float* vals = new float[SZ];
        float sum = 0;
        for (size_t i = 0; i < SZ; ++i) sum += vals[i] = i;
        delete DataFrame::fromFloatArray(&main, &kv_, SZ, vals);
        delete DataFrame::fromFloatScalar(&check, &kv_, sum);
        delete[] vals;
    }

    void counter() {
        DataFrame* v = kv_.wait_and_get(main);
        float sum = 0;
        for (size_t i = 0; i < 1000; ++i) sum += v->get_float(0,i);
        p("The sum is ").pln(sum);
        delete DataFrame::fromFloatScalar(&verify, &kv_, sum);
        delete v;
    }

    void summarizer() {
        DataFrame* result = kv_.wait_and_get(verify);
        DataFrame* expected = kv_.wait_and_get(check);
        pln(expected->get_float(0,0)==result->get_float(0,0) ?
            "SUCCESS":"FAILURE");
        done();
        delete result; delete expected;
    }
};
```

Status

What has been done:

- Implemented Vector classes that hold objects, ints, floats and booleans respectively. These data structures are resizable while maintaining constant look-up time and avoiding copying the payload.
- Implemented the DataFrame class. Users are able to read data from a file using an adapter and generate DataFrames from it. They can also map and filter the DataFrame.
- Implemented a working network layer that allows for client registration with a server and direct client communication.
- Implemented a protocol for serialization and deserialization.
- Implemented serialization and deserialization of Object, String, all vectors, columns, Message subclasses, and DataFrame.
- Implemented the KVStore and Application classes. They now run properly on a single node.
- The network layer was integrated into the KVStore class. Now an application can run with multiple nodes and nodes correctly share data between their KVStores. However, entire dataframes are still being serialized and put into KVStores which isn't scalable.
- Implemented the DistributedVector class so that DataFrames' contents are all serialized and distributed across each nodes' KVStores.
- Implemented the KDStore class so the Application layer can still put and get DataFrames.
- Got the word count and degress of Linus applications working.
- Revamped serialization, removed unused code, added more tests, refactored here and there.