Architecture of EAU10 (binary EAU2)

Introduction

EAU2 is a distributed key value system to process queries in a big data setting. This can be loosely compared to a very basic hadoop architecture without most of the conveniences of the hadoop api. EAU2 is based around an API to abstract and manage a distributed key value store to manage large scale data.

Architecture

The client will produce a package to run on a fixed set of N+1 distributed nodes. One of the nodes will be reserved as the “server” node that each “client” connects to, leaving N “compute nodes”. A server node does not do any computing, but servers as a coordinator for the compute nodes. The data is batched across the N compute nodes, and is read-only with the queries being processed in memory. There is a KV store running on each compute node.

Implementation

The overall architecture will have compute nodes and a server node.

The server node exists to register and update clients on said registration It also manages graceful shutdown, telling all clients to “END” when it’s shutdown begins. A server node does not do any computing.

Each compute node will have three layers:

- The KV store abstracted around our networking API.
- Any distributed data structures abstracted around the KV API
- The Application(s) the customers will be running abstracted over the data structures and the kv store.

A class named Application pre-defines the code for setting up the distributed key value store storage (maintaining a local key-value store [see next paragraph]) and query system similar to how Hadoop has parent implementations for Mappers and Reducers. The application class is expected to be extended by the customer in order to interface with the API. The application will also provide context for each instance of itself for algorithms that require information from the global state such as dependency to finish. Each application is initialized with a unique index to identify it's key value store.

The key-value store will spin up a network Client from our networking API and maintain a local map with string keys. It will accept Keys which are (String, Size_t) where size_t indicates the index of which application’s KV store contains the data. If the index is not that of the current application, it will send a message across the aforementioned Client it wraps around to the other application asking it for the value it. This data on the other application is streamed to the application asking for the data via the network layer. The KV store will wrap around the C++ standard Map class for the local storage.

For the foreseeable future only strings, numerics (ints and floats), and booleans are supported data types.

Use cases

This could be in the form of code like the one above. It is okay to leave this section mostly empty if there is nothing to say. Maybe just an example of creating a dataframe would be enough.

Here is an example application. An example use case would be in a datacenter with 3 nodes. There is a producer, counter, and summarizer node running in a distributed setting. The dataframe data is distributed on node 0 in this example, and the other nodes get the data on other nodes via network streams.

```
 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");
  }
};
```

Open questions

Status

As of today we have the base non distributed base classes such as dataframes. We need to create algorithms to manage the distributed setting of this application and the key value store. We need to create the base application api for programmers to interface with.

The aforementioned work that needs to be done has not been done as of submitting this report and should be the first priority. Further, this code should be extensively tested both for correctness and performance. As we work, we will need to maintain communication with the client and make changes to the API / design / architecture / implementation as new requests come in

and the scope creeps. Management also needs to be kept in the loop to ensure we maintain the standards of CS4500 inc.

This will probably take us about 5 weeks.

As of 3/22:

- we have a one node KVStore with no distribution working with the sample "Trivial" class.
- We have no memory leaks
- We have a base Application class that manages the kv store.

As of 3/30:

- we have a distributed kv store
- we have a serializable dataframe
- we demo the client's requested functionality

As of 4/6:

- The distributed kv store correctly seriales and deserializes data frames
- The kv store has a working wait and get for existing keys
- Chunks can be put into the kv store from a node not storing the chunk, ie the key for a put can have a different index than the node (the put is distributed).
- Currently empty dataframes cannot be redistributed over the network. This is probably be fine for the eau2 use case, but is a limitation that needs to be refactored.
- Wait and Get needs to be refactored, so we can request for non-existing keys.
- We have *two* viable demonstrations of a WordCount program (m4 and WordCount) in the last stages of development. This is meant to show how to use our eau2 system in order to meet the customer requests. The program is fully distributed. The former demo is meant to closely parrallel the customer application draft; the ladder is meant to show an alternative straightforward way to implement similar functionality with our design / architecture in mind.

As of 4/13

- we have a distributed dataframe that distributes in equal chunks (by row) across the nodes.
- Our linus demo matches our api / the customers wishes
- Linus compiles
- We integrated Sorer (with dataframes and distributed dataframes) from TeamRP
- Linus has currently been tested with 1 degree of seperation on 1 node due to time and hardware concerns. We expect it to fully scale up in terms of functionality as the networking code and dataframes / other datastructures have been thouroughly tested and verified.
- NOTE: to run, first compile the server and whichever client-based appplication you want to run (like linus). Then, run the server with "-ip 127.0.0.1". Next, run the client you chose (in a seperate shell/terminal). When the client has completed, press enter on the server session to initiate graceful shutdown.

As of 4/22

- Word Count is fully demonstrable
- Sorer is tested and up to date for complex data files
- Documentation has been increased
- Wait_and_get is more robust, now capable of waiting on local keys