Ryan Heminway
David Tandetnik

Eau2 System Architecture and Design

Introduction

The eau2 system aims to provide key-value storage over a distributed set of nodes acting as the backbone for applications which process large amounts of data. With the requirement of a cluster of compute nodes, users can run the eau2 system to store data across the cluster but interact with it as if it was local. Eau2 only accepts input data formatted based on schema-on-read (SoR) requirements, and that data is read-only once it is stored in the system. For queries, the eau2 system provides the data in the form of a dataframe which allows for easy data aggregation and access.

Architecture

Conceptually, there are three layers of the eau2 system. At the bottom is the key-value storage system. Data is stored across multiple nodes, so this layer also includes a network protocol which allows the key-value stores on each node to interact with each other. Each key consists of a string and a home node while each value is a serialized blob of data. Communication between the stores allows the user to view the key-value storage as a single system holding all the data, without any knowledge of the underlying distribution. The layer above provides abstractions like distributed columns and distributed dataframes. These utility classes are useful for data aggregation and are easier to work with than the raw data. At the top sits the application layer. This layer provides all the functionality necessary to run the eau2 system and interact with the lower levels. The user builds on top of this layer to leverage the underlying key-value store as the backbone of their program.

Implementation

The bottom layer will be implemented using one main class, for now referenced as Store. This class will have two main components, a Map and a Network. The map is the normal key-value storage structure, mapping Keys to Strings. Keys are a new type which represent a tuple of a String and an integer to denote the home node identifier of the data. Strings are used to hold serialized blobs of data. The Network object on the other hand contains all the necessary logic for registering and communicating with the other nodes in the cluster. Overall, the Store layer will have a simple public API supporting methods such as ‘get’, ‘getAndWait’, and ‘put’ which all work with DistributedDataFrames. There is also a set of analogous private API methods that work on chunks of DistributedColumns, described below.

In the middle layer we have the useful abstraction classes. Specifically, the DistributedDataFrame class and the DistributedColumn class. The DistributedDataFrame is a queryable table of data which supports aggregate operations such as ‘map’ and ‘filter’. All data

stored in a DistributedDataFrame is stored in columnar format with all data in each column being of a single type. The types supported by a DataFrame are the four SoR types: INT, BOOL, FLOAT, and STRING. Unlike normal DataFrames, DistributedDataFrames access their data as needed from other nodes across the network under-the-hood.

The DistributedColumn class has the API of a normal column, however it is implemented such that the values it stores are distributed in chunks across the cluster of nodes using the Store's private API. This abstraction allows this class to represent much more data than the average column stored in local memory.

For the top layer, we provide a single Application class. This class contains a 'run_' method which must be invoked on all nodes in the cluster to start up the system. This class has the Store as a field, providing the user which extends the Application class with access to all data stored on the system. Additionally, this class will contain any utility methods required for building a program on top of it. One example is the Application API method 'this_node' which returns the index of the current node.

Use Cases

```
// One node saves some data to the store, the other takes the max of the data and saves that to
// the store too.
class Demo : public Application {
Public:
   Key df("data", 0);
   Key max("max", 1);

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

   void run_() {
      switch(this_node()) {
      case 0: producer(); break;
      case 1: maxer();
      }
    }

   void producer() {
      float vals = new float[3];
      vals[0] = 1;
      vals[1] = 10;
      vals[2] = 100;

     DataFrame::fromArray(&df, store, 3, vals);
}
```

```
  void maxer() {
    DataFrame* frame = kv.getAndWait(df);
    int max = frame->get_int(0, 0);
    for (size_t i = 1; i < frame->nrows(); i++) {
      if (kv->get(0, i) > max) max = store->get(0, i);
    }
    DataFrame::fromScalar(&max, store, max);
  }
};
```

Open Questions

1) What is the intended setup for running our system across multiple nodes? Will this ever be a reality? We are struggling to set up a pseudo-network using threads. Are there resources which describe a good methodology for doing this?
2) What are some recommended ways for debugging issues with threaded applications? When we introduce threads for our pseudo-network in WordCount, we get new seg-faults in previously unseen places. The location of the seg-fault is not always consistent, and it feels like we are chasing a ghost. We have tried to place locks on data accessed by multiple threads, but likely we are not doing it entirely correctly.
3) We know we have some busy loops and this was something called out in our code-review. We want to replace them, but have pushed it off because we got a bit confused by the wait-and-notify design pattern. Most resources online detail this pattern using C++ tricks we are not familiar with. Our question is: are busy-loops bad practice because they waste resources, or do they actually cause unexpected problems?

Status

At this point, our system is capable of storing data in DistributedDataFrames that distribute their data across the network. More specifically, every DistributedColumn automatically distributes its data, in a round-robin fashion, across the nodes. Based on the round-robin distribution style, a constructed Row in a Dataframe will always be constructed with data all from the same node. This will be especially convenient when that node happens to be the current node. We leverage this in our M4 WordCount. We have also updated Application and Sorer classes to support creation of a DistributedDataFrame directly from a SoR file. With these changes, and some general cleanup and bug fixes, we are able to read in a Words SoR file into a single DDF. Instead of the space separated file mentioned in M4, we decided to implement the M4 code in our own way using a SoR file containing Strings. We felt this was a more beneficial test of our system since the end-goal is to process SoR data, not space separated data. Our WordCount demo can work with the test-file we created, but we are running into some problems:

1) If we make the file very large (multiple GB), the linux kernel kills the program for using too much memory. This would be resolved in a system with multiple nodes,

but we are currently testing with multiple threads which does not relieve the problem.

2) We try to create multiple “nodes” in the form of different threads running the WordCount application, but the threading seems to result in very inconsistent behavior. Rarely, it will work correctly. Most of the time, we get one of many different errors, usually a segfault somewhere in the thread library functions. We are still unsure how to correctly set this up.

Ultimately, we are at a point where we can carry out the WordCount demo as long as the provided file is small enough to store on one node, and we run with at most one WordCount thread.

Additionally, we still have some memory issues with our serialization approach and we will be going through that code with valgrind in the coming weeks.We have improved the networking and threading code in our Store, but clearly this is still a place for improvement now that we see the results from this week’s demo.