Introduction

Done by @grahamwren and @jagen31.

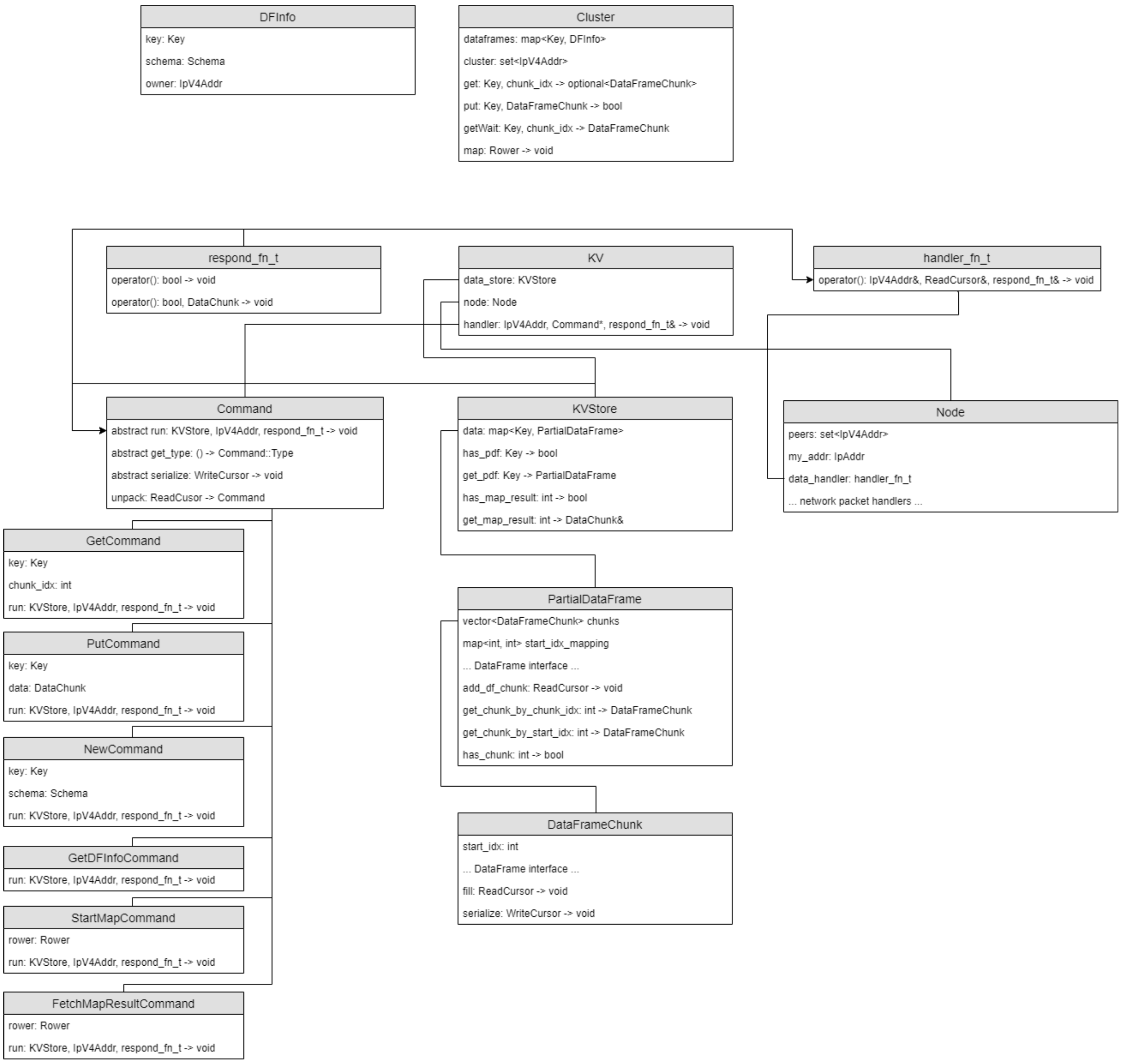
EAU2 is a system for distributing data over a network and running computations in parallel over that data. EAU2 uses a distributed key value store to manage multiple dataframes which may be larger than memory and communication over TCP/IP to orchestrate parallel computations over those dataframes.

Architecture

EAU2 has three layers. The **Application** layer is where programs are written that make use of EAU2. Programs can construct multiple dataframes and make multiple queries. The **KV** layer is where data is distributed logically, and provides the interface through which the system is queried. The **Network** layer is where requests are made between nodes.

Each node in the network holds parts of dataframes and knows where to look to find the parts it does not own (see Implementation for more).

Implementation


DFInfo
key: Key
schema: Schema
owner: IpV4Addr
Cluster
dataframes: map<Key, DFInfo>
cluster: set<IpV4Addr>
get: Key, chunk_idx -> optional<DataFrameChunk>
put: Key, DataFrameChunk -> bool
getWait: Key, chunk_idx -> DataFrameChunk
map: Rower -> void
respond_fn_t
operator(): bool -> void
operator(): bool, DataChunk -> void
KV
data_store: KVStore
node: Node
handler: IpV4Addr, Command*, respond_fn_t& -> void
handler_fn_t
operator(): IpV4Addr&, ReadCursor&, respond_fn_t& -> void
Command
abstract run: KVStore, IpV4Addr, respond_fn_t -> void
abstract get_type: () -> Command::Type
abstract serialize: WriteCursor -> void
unpack: ReadCusor -> Command
KVStore
data: map<Key, PartialDataFrame>
has_pdf: Key -> bool
get_pdf: Key -> PartialDataFrame
has_map_result: int -> bool
get_map_result: int -> DataChunk&
Node
peers: set<IpV4Addr>
my_addr: IpAddr
data_handler: handler_fn_t
... network packet handlers ...
GetCommand
key: Key
chunk_idx: int
run: KVStore, IpV4Addr, respond_fn_t -> void
PutCommand
key: Key
data: DataChunk
run: KVStore, IpV4Addr, respond_fn_t -> void
NewCommand
key: Key
schema: Schema
run: KVStore, IpV4Addr, respond_fn_t -> void
GetDFInfoCommand
run: KVStore, IpV4Addr, respond_fn_t -> void
StartMapCommand
rower: Rower
run: KVStore, IpV4Addr, respond_fn_t -> void
FetchMapResultCommand
rower: Rower
run: KVStore, IpV4Addr, respond_fn_t -> void
PartialDataFrame
vector<DataFrameChunk> chunks
map<int, int> start_idx_mapping
... DataFrame interface ...
add_df_chunk: ReadCursor -> void
get_chunk_by_chunk_idx: int -> DataFrameChunk
get_chunk_by_start_idx: int -> DataFrameChunk
has_chunk: int -> bool
DataFrameChunk
start_idx: int
... DataFrame interface ...
fill: ReadCursor -> void
serialize: WriteCursor -> void

The Cluster class has the ability to make use of the KV store to manipulate dataframes. The Cluster accesses the KV store through the KV class. The KV class provides an interface to run Commands in the KV store. Commands are serializable objects that do something with the store. Standard commands are GET, PUT, DELETE, NEW, GET*DF*INFO, START*MAP, and FETCH*MAP_RESULT. The KV store is distributed over the network, and a local KV Store class (different from the KV class) is present on each node. The KV class communicates with the local KVs by serializing and deserializing commands.

The "Application" (the Cluster + user code) is not running as a node participating in the network, but rather, communicating with the network. This means an application has to send a message to the machine it's running on in order to run commands locally, rather than having a separate, faster implementation for local operations.

The local KV store, represented by the KVStore object, contains chunks of data frames grouped in the PartialDataFrame class. The chunks are organized by the Cluster class. When a new piece of data is added, a random owner is chosen from the cluster. The data is then distributed round-robin over the nodes, in fixed size chunks. Calculating the node which a specific chunk is at is a matter of finding the owner, and taking the chunk index modulo number of nodes. The local KVStore also contains another mapping of an integer id to map results. These are the local results of running START*MAP commands over the cluster, and are accessed using the FETCH*MAP_RESULT command.

Keys in the KV Store are strings, though one string can represent data spread over several nodes. A string key combined with an index can be used to get specific chunks.

Commands are serializable objects with a run method. The run method is passed a respond*fn*t, which is a callback that may only be called once, and allows the command to make a network response before it finishes performing its computation. This is used in the START_MAP command to allow for parallel mapping, by returning an id to request the results of the computation later.

When `put` is called with a new key, the KV adds the new mapping to its ownership map and broadcasts a corresponding message to the other nodes in the network. The data passed into `put` is converted into rows one chunk at a time. The chunks are distributed over nodes in the network in a round-robin fashion. That is, nodes have a fixed order and chunks are distributed by traversing them as a ring, starting at the node where the put request was made.

The `DataFrame` object itself is a virtual dataframe, that is, it delegates to the KV in order to perform its operations.

A computation is run on a dataframe by issuing `START_MAP` commands with a `Rower`. Rowers can be serialized, and in a map, one START*MAP is sent over the network containing one serialized Rower, to each node. When a node receives a Rower, it responds right away with an integer id that serves as a handle to fetch the results later. It then runs the computation over its PartialDataFrame, which maps over all the chunks it owns, in parallel. Then, the Rower is reserialized and stored in the KVStore as a map result under the aforementioned id. These results can be fetched with a FETCH*MAP_RESULT command.

The KVStore's map method takes care of issuing the `START_MAP` commands to each node. It then waits on each of them by issuing a `FETCH_MAP_RESULT` command. This means the map happens in parallel, but the results "finish" in order. The KVStore joins them using `join` from the deserialized rowers. The joins are also done in order.

A `Node` is the low-level interface to the networking layer. The node will handle network commands automatically, and takes a callback for handling application commands. The callback, set through `set_data_handler` by the KV, will be used by the KV to handle Application level requests made of this Node by other Nodes in the cluster. The callback in KV deserializes the data as a Command and calls the Command's run method.

Use cases

Sum Numbers in DataFrame

```
input_file.sor:

<1><2><3>
<4><5><6>
```

```C++
Key key("example");
Cluster cluster(IpV4Addr("<ip in cluster>"));

cluster.load_file(key, "input_file.sor");

SumRower r;
cluster.map("example", r);

> r.result
21
```

WordCount

First chapter of Moby Dick or The Whale. Obtained from archive.org.

▶ loomings.sor

```C++
Key key("loomings");
Cluster cluster(IpV4Addr("<ip in cluster>"));
cluster.load_file(key, "loomings.sor");

WCRower rower(0);
cluster.map(key, rower);

for (auto it = rower.get_results().begin(); it != rower.get_results().end(); it
  cout << "word: " << it->first << ", count: " << it->second << endl;
}
```

Notes

Each node in our EAU2 cluster is a `kv_node`, to build this executable use the `build/kv_node.exe` make task and then run it with ip and server ip arguments:

```Bash
$ build/kv_node.exe --ip 172.0.0.2
```

And on another machine in the network:

```Bash
$ ./build/kv_node.exe --ip 172.0.0.3 --server-ip 172.0.0.2
```

Once a cluster is running, you can start an app which interacts with this cluster. Specifically for our linus implementation, each file can be loaded with the `load_file` app:

```bash
$ ./build/load_file.exe --ip <ip addr in cluster> --key commits --file datasets
$ ./build/load_file.exe --ip <ip addr in cluster> --key projects --file dataset
$ ./build/load_file.exe --ip <ip addr in cluster> --key users --file datasets/u
```

Then the computation is run with the `linus_compute` app:

```bash
$ ./build/linus_compute.exe --ip <ip addr in cluster>
```

To run the app with the simple test data just use the default make task. This task will launch the cluster in docker, load the there easy-data files into the cluster, and then perform the "7 Degrees of Linus" computation on them.

```bash
$ make run
```