EAU2

Chase Broder

Marcin Kierzenka

CS 4500 - Software Development

Spring 2020

Usage:

Makefile

- "make build" - compiles all tests
- "make test" - runs tests supplied to us (eg. trivial, demo tests)
- "make ourTests" - runs sorer test and a bunch of others testing different parts of our program
- "make debug" - runs particular test from suite. Useful if one test in particular is failing and want to focus on that one
- "make memory" - runs selected test through valgrind
- "make clean" - deletes compiled files
- there are also targets for specific applications (eg. buildDemo and testDemo)

Program command line usage

./eau2 -app S -num_nodes N -i N -ip S -port N -serverIp S -serverPort N -pseudo -blockSize N

- app (string): which program you want to run; either trivial, demo, wordcount, or linus
- num_nodes (int > 0): how many nodes should be ran; default is 1
- i (int >= 0): this node's index (node number); default is 0
- ip (string): this node's IPv4 address (dotted quad)
- port (int > 0): node's port for receiving messages; default is 8080
- serverIp (string): IPv4 address (dotted quad) of server node
- serverPort (int > 0): server's port for receiving messages; default is 8080
- blockSize (int > 0): how many elements each chunk of stored data should have; default is 1024
- pseudo (flag): indicated whether pseudonetwork (threads) should be used

Introduction

The eau2 system is a framework to allow a wide variety of applications to operate on large datasets distributed across nodes in a network. Each node has a collection of key-value pairs stored locally as well as the ability to retrieve values from other nodes over the network, as needed. The system can be scaled for different use-cases (supports an arbitrary number of nodes). Most cases will have a different responsibility for each node, for example one to produce data and another to consume it.

Architecture

The eau2 system will run in parallel across a collection of nodes. Each node performs a portion of the "work" for the chosen application. There will be a lead node ("server") that keeps track of registered nodes as well as initiates and terminates the application. Besides that, there is no inherent difference between the server and other nodes.

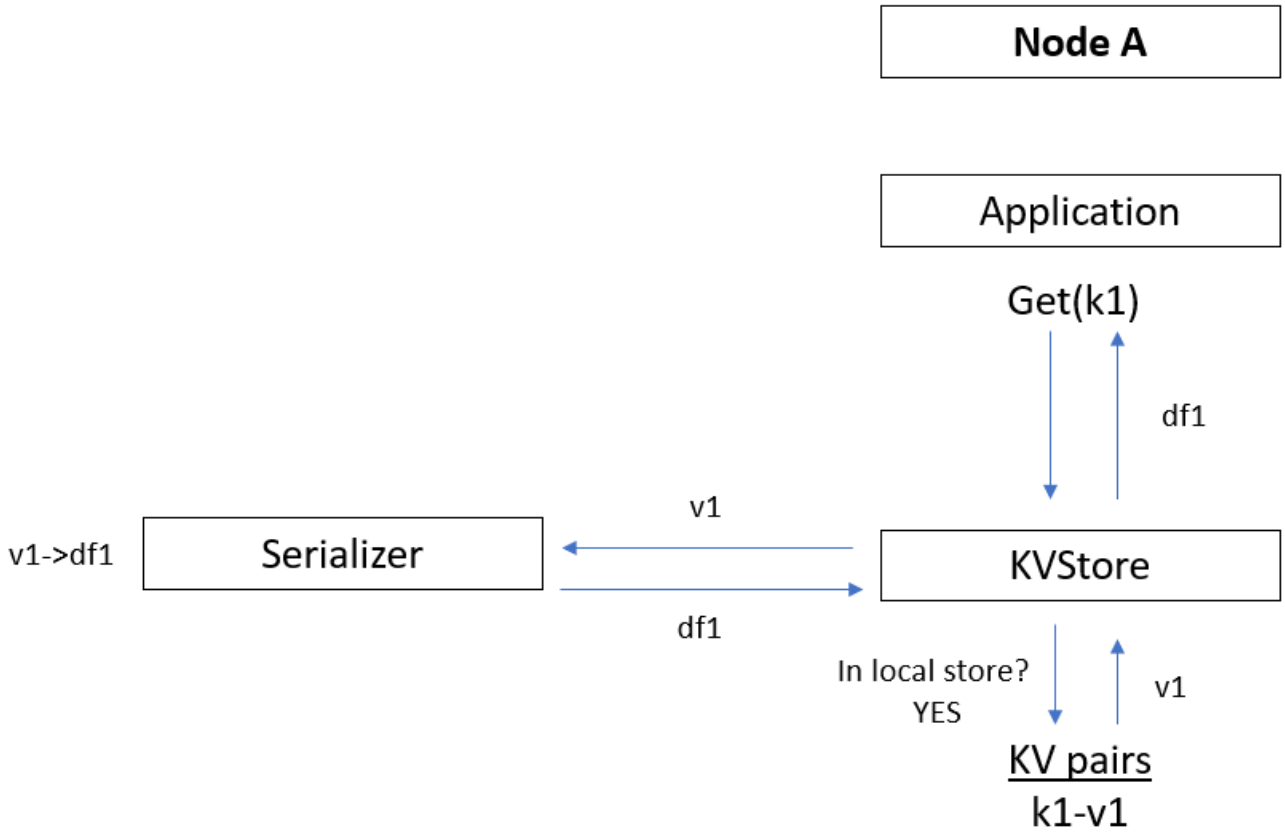
Each node is made up of three layers of abstraction.

The first and lowest layer is the distributed key-value store. There will be one key-value store per node, containing the data it is in charge of. The node can put and retrieve data from its key-value store based on the needs of the application. Each key in the store will map to a chunk of serialized data or a dataframe. If a node asks its store for data not stored locally, the store will be able to communicate with the other key-value stores to retrieve the proper values. This will be done via a networking abstraction, which will allow key-value stores to send and receive messages to each other.

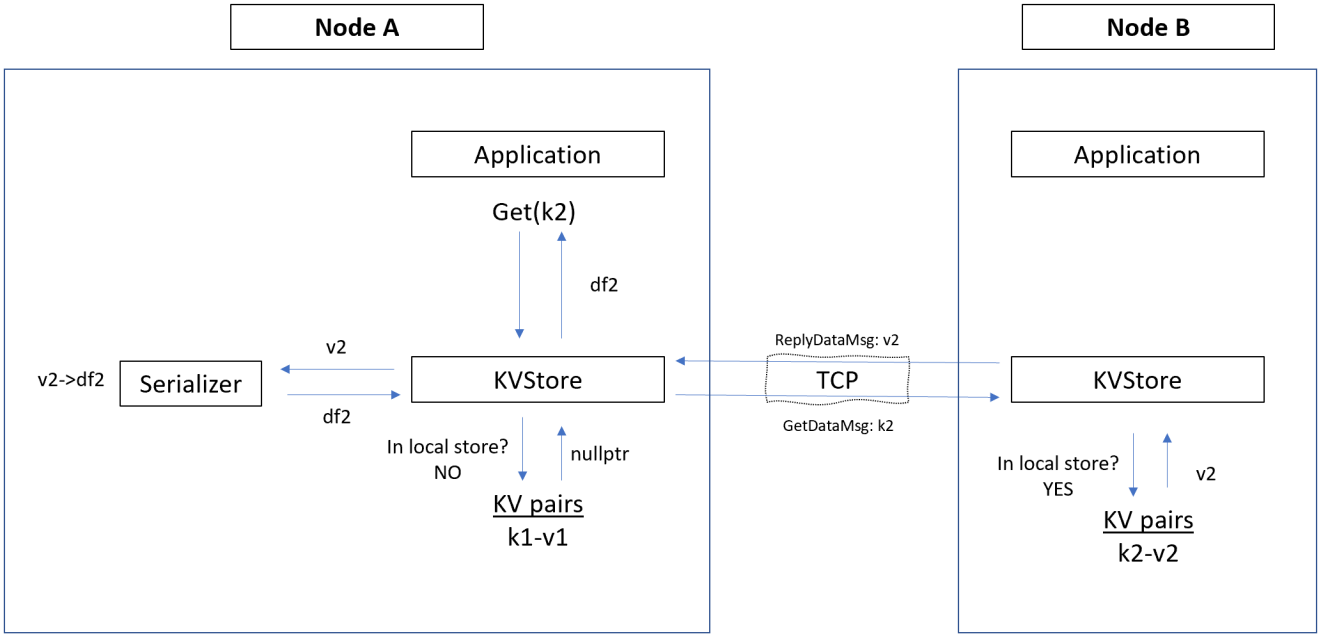
The next layer will include abstractions for holding and representing the data from the key-value stores. These abstractions include distributed arrays and dataframes. A dataframe allows the user to access a set of values distributed across multiple nodes. It supports data storage and retrieval without exposing the details of how and where the individual elements are stored. A dataframe holds a collection of columns. Columns do not own their own data, rather they are distributed arrays. Therefore, columns contain references to their data that exist across the nodes' stores.

The last and highest level is the application layer. In the application, the user will be able to specify what they want each node to do. Each node will run its assigned operations in its copy of the application. The application can get data from and put data into the distributed key-value store system. Distributed arrays can be used to track, organize, and work with specific data across the eau2 system.

In the following paragraphs, we will explain diagrams that outline the basic interactions between components of our system. There are three attached PNG diagrams that can also be found within this "report" folder.

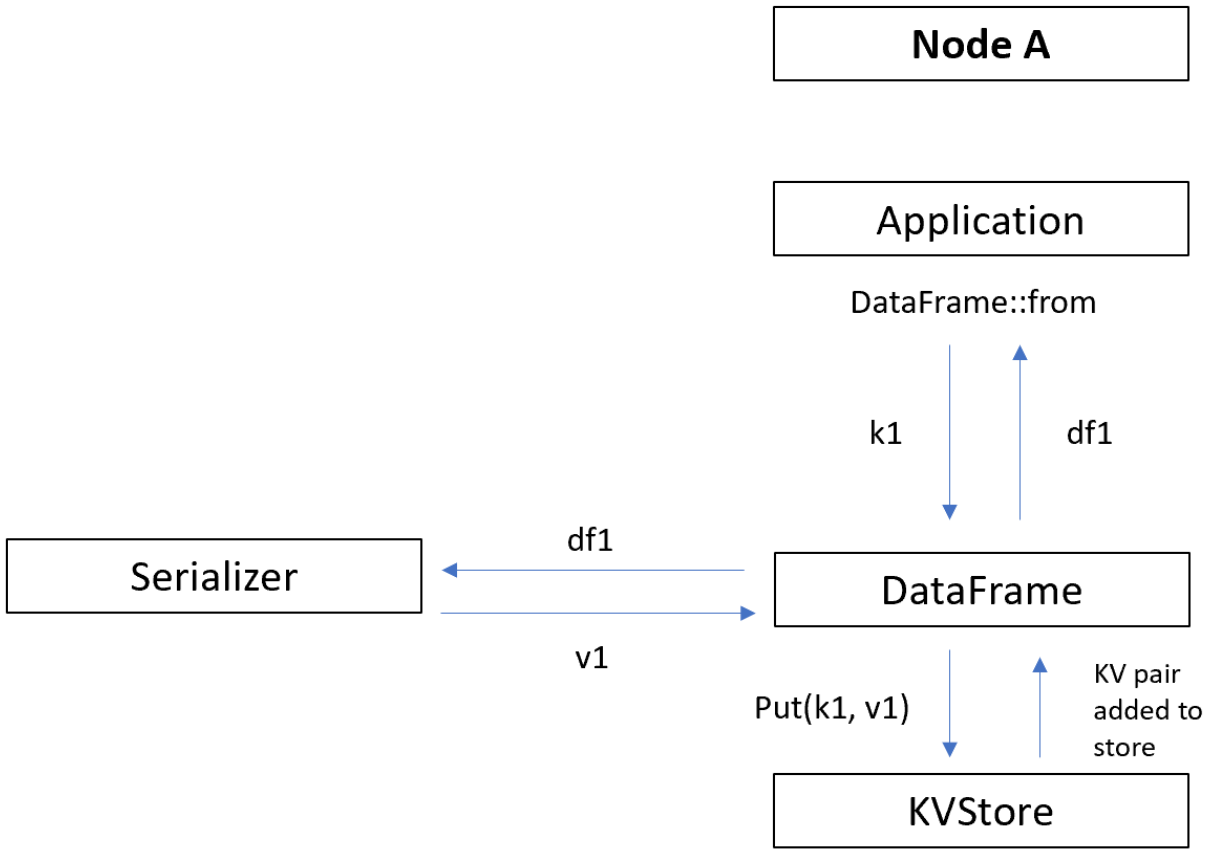


The first diagram, localgetflow, shows a node running an application that retrieves data that exists locally in its KV store. The process is fairly simple; the application calls on the store to get data under a certain key. The store recognizes it has that data. It then takes the serialized data, converts it into a dataframe, and then returns the result to the application.



The second diagram, networkgetflow, represents the process of retrieving data that does not exist in the node's local store. Like in localget, the application calls on its store to get data under a certain key. However, this time, the store does not have that key. Therefore, the store needs to request the data from a store

running on a different node. Through the networking layer, the local store asks the store mapped to that key for the data. If it is available, the remote store sends the data back to the local store. The store then converts the data received and returns the dataframe result.



The last diagram, addframeflow, represents how an application can insert a dataframe into its node's store. First, the application needs to call a static "from" method, which will create a dataframe given a key and certain pieces of information. Contrary to the previous cases, the method converts a dataframe into a blob of data. It then takes the key and the value generated and puts it into the store. Lastly, the "from" method will return the produced dataframe for the application to use.

Implementation

For the critical classes that we will implement for this system, we included a run-down of their fields and public methods. For components that are re-used from previous assignments, we provided descriptions of how they'll be used.

- KVStore: This class will represent a local key-value store.
    - Fields:
        - kvMap_ (MapStrObj): the String key will map to a serialized piece of data (Value)
        - node_ (INetwork): this network abstraction will allow the store to listen for messages from other nodes and send messages to other stores to request data or put data in another node's store

        - storeId (size_t): each local store will be represented by a unique numerical identifier (the "index" command line argument); at a higher level, this identifier will help keep track of where the data is stored
        - msgCache_ (Map): this map will hold the messages that the KVStore cannot handle right away. For example, if another node requests data from this store that is guaranteed to eventually be in this store, the store will hold that message in the queue until the data is present. The key will be an actual Key object and the value will be an array of messages referencing that key. For example, if two nodes try to get the same data from the same store, then the key for that data will correspond to two get messages. Right now only two types of messages will exist in this cache: WaitAndGetMsg and ReplyDataMsg
        - msgCacheLock_ (Lock): used when accessing the cache, to avoid another thread from reading from/writing to the cache at the same time

    - Methods:

        - put(Key, Value): adds given data to the key-value store specified in the key, not blocking; sends Put message if Key's node id doesn't correspond to this store's id
        - get(Key): request for the data; returns deserialized data from its own store or from the network if data for the key exists, otherwise returns nullptr
        - waitAndGet(Key): retrieves data with the given Key from the Key's node or from the network if not local; blocking, so won't return until data exists in store specified by key
        - getValue(Key): return the value that the given key maps to, in the raw storage format (Value); goes to the network if data not stored locally
        - addMsgWaitingOn(WaitAndGetMsg): add the message to cache; this kvstore currently doesn't have the data to respond to another node's request, so need to wait until the data is put into the local store
        - addReply(ReplyDataMsg): adds reply data message to the cache, to be forwarded to the application (which is waiting for it)

- Key: Identifies a piece of data at a level higher than the local KV store (ie. eau2 system-wide). Since data can exist in any of the stores, we need multiple attributes specify location

    - Fields:
        - kStr_ (String): maps to a piece of data within a key-value store
        - homeNode_ (size_t): the identifier for the key-value store in which the data is stored

NOTE: Keys are immutable. There will be methods to retrieve these values but not modify them

- Value: Represents the serialized data that a Key maps to. It will include a character pointer that holds the serialized data and a size_t that describes the maximum amount of bytes that can be stored in this pointer.

- INetwork: An interface outlining the required operations to be a valid network in our system. This was so that our entire program could use an INetwork, allowing for easy swapping between network implementations. The interfaces includes seven methods.

- PseudoNetwork: A fake networking layer to aid with debugging our eau2 system (implements INetwork). The design was similar to that specified by Professor Vitek in his videos. A PseudoNetwork has a pointer to an external array of message queues; each queue keeps track of the messages received

and pending for the node (id = index in array). Send pushes the given message to the target node's queue, while receive pops from the node's queue. This means each node (ie. thread) needs access to the same message queues. Therefore, we initialize a common MessageQueueArray that is passed into every PseudoNetwork object. This is the network class used when the "-pseudo" flag is specified. The INetwork methods server_init(), client_init(), handleRegisterMsg(), and handleDirectoryMsg() have empty implementations because their behavior is unnecessary for this fake network.

- Network: A TCP networking layer for the eau2 system (implements INetwork). It's implementations of the INetwork methods contain all the code for opening/closing sockets, sending/receiving, etc. It implements the registration and teardown processes outlined below. Each node has a socket that it uses to receive messages. For sending it will create a new connection directly to the target node, send the data, then close the connection.
    - Registration:
        - Server starts up on an IP Address and Port known to all nodes, will accept all incoming connections
        - The rest of the nodes each start up and send a RegisterMsg containing their NodeInfo. Then blocking, waiting for a response from the server
        - Once all of the nodes (expected number specified on the command line) have registered, the Server node will send each one a Directory containing the information for how to contact every node.
    - Teardown:
        - Once a non-server node completes their application tasks, it will send the server a DoneMsg and wait to receive a response
        - The server node tracks which nodes have finished. Once all nodes in the system are finished, the server sends everyone a TeardownMsg and shuts down.
        - Each client shuts down when it receives a TeardownMsg
- NodeInfo: A class to wrap a struct sockaddr_in allowing easy storage/retrieval in our Map, Array, Messages, etc.
- Messages: Messages are what nodes use to communicate each other. Data can be placed inside messages. They are serialized by the sending node, then sent, then deserialized and read by the receiving node. There is an enum (MsgKind) that outlines the types of messages used in the eau2 system. Each message type corresponds to a message subclass with a distinct purpose. The following outlines the current message types sent between nodes in our program:
    - GetData: this signifies a sending node's request for data from the destination node. If the target node does not have the data when it receives this message, it should respond with nullptr
    - WaitAndGet: like a GetData message, except it expects the response to contain the value. If the target node does not have the data when it receives his message, it should cache it and respond once it adds the data to its store. The sending node's application will pause execution until the data comes back (ReplyData, non-nullptr Value)
    - ReplyData: this is a response to a data request (either GetData or WaitAndGet). The node who received the request message will use this message to respond with the serialized data (or nullptr in some cases)

    - Put: this message puts data, both a key and a value, in another node's store
    - Register: allows a client to register with a main, or server, node that keeps track of connections
    - Directory: broadcasts a list of connections to each client; necessary for clients to be able to send messages between each other
    - Done: a node sends this message to the server when it's done executing its task. It means it is ready to begin the teardown process, once the other nodes complete
    - Teardown: broadcasted from the server to all connected nodes. Means that the program is terminating, so all nodes should close their connections/sockets and delete their objects

- DataFrame: The DataFrame API will be similar to that of previous assignments. It will include operations to store and perform operations on data, such as map. A Schema will be used to describe the DataFrame's column structure. A DataFrame will be immutable, so it will not support operations such as deleting, setting, and modifying columns and rows.

  DataFrame's data storage will change. In early assignments, DataFrames held all of their data in columns and rows. However data storage is now distributed. Instead of storing the actual data, Columns will now be distributed. Like before, the DataFrame will have an array of columns. However the Column class will now contain a Distributed Array of keys. Each key will map to a "block" of data that is held in a KVStore in the eau2 system.

  The DataFrame will contain a pointer to the KVStore running on that node. It may contain some of the DataFrame's data, but some blocks can also exist in other KVStores. Therefore, the store is not only important for local retrieval, but for also getting data that exists on other nodes. Not only does the DataFrame need this object, but any other objects within the frame that need to store and request data will also get a pointer to it.

  The DataFrame will have methods that support converting data types into frames. An example is fromArray; it will take in an array of a certain type, and it will return the DataFrame version (add the values in one new column). It will break up the array into chunks; each chunk will be assigned a key to be stored in the distributed KV system. The key will then be added to the column's distributed array. Once the input array is completely broken up and stored, the DataFrame will be serialized and stored under its own key in the distributed system.

    - Fields:

        - columns_ (ColumnArray): holds column information
        - store_ (KVStore): dataframe passes store to classes that need it for data lookup; static initialization
        - schema_ (Schema): keeps track of the DataFrame's column structure
        - key_ (Key): key for this dataframe in the store; will be used to create the keys for the frame's columns' chunks

    - Methods:

        - void add_array(size_t, *type*): adds array into the DataFrame as a column; one for each type
        - void map(Rower): perform an operation on all data in the DataFrame
        - void local_map(Reader): perform an operation on all local data in the DataFrame (ie. data stored in the KVStore field)
        - Schema& get_schema(): return schema of DataFrame

        - void add_column(Column): add column to DataFrame
        - get: get and return certain value from DataFrame's columns; one for each type. These block on network lookups
        - void add_row (Row): add a row to the DataFrame (currently not fully implemented)
        - void visit(Writer): get new rows from a Writer that are sequentially added to this df
        - size_t nrows(): return number of rows in the df
        - size_t ncols(): returns number of columns in the df

- DataToDf: this class contains all of our static "from" methods that create and serialize DataFrames fromData. These methods include fromFile, fromArray, fromVisitors, and our from scalars (int, String, bool, double). These files were initially in dataframe.h. However, since these methods are stateless, we decided to move them out to reduce the size of the dataframe file.

- Column: A column stores data in the distributed KV Store, instead of locally. A Column will be a DistributedArray where each Key points to a Value containing a fixed sized number of elements that belong to this Column

    - Fields:

        - blocks_ (DistributedArray): holds blocks of column data
        - size_ (size_t): number of elements in the column (!= number of keys in array)
        - store_ (KVStore): will be used by column to retrieve its own data when needed and add keys to the store
        - type_ (DataType: enum): what type of column is this? String, boolean, int, or double
        - baseKey_ (Key): key of the dataframe that the column belongs to; will be used to build up block keys

    - Methods:

        - most methods from before, for getting data from the column. not 'set' methods (or anything else that modifies the column).
        - void add_all(size_t, *type*): one for each data type; adds all given elements of that datatype to the column; breaks data up into chunks, creates keys for those chunks, and adds them to the KVStore
            - The chunks are inserted into the KVStore in a distributed fashion. Therefore, not every chunk is inserted into the same store as its Dataframe. When adding a block, we determine which block number it will be in the column. We then find the modulus of dividing the block number by the number of nodes. For example let's say we're about to add block 9 to the column and there are four nodes running. We'd add the block to the KVStore on node 9 % 4 = 1.
        - void setStore(KVStore): pass a KVStore to the dataframe; will initially start in the dataframe, which will be passed to the column array and eventually the column
        - size_t size(): return size of column
        - get: get element from column; one method for each type

- Serializer: This class supports writing primitive types, character pointers (strings), and message types to its buffer, and can also read these when deserializing. It will be in charge of serializing and deserializing data, and buffering the result. The Serializer will be used in multiple places in this project. Serializers are used in the static DataFrame fromX methods to store the new df in the distributed key-value store.

They are also used in the KVStore:get and KVStore::waitAndGet methods to build a DataFrame from the serialized KVStore data. Also, in the sendMsg/receiveMsg Network methods to move data (as a series of bytes) across a TCP connection. Finally, Serialization is also needed whenever attempting to retrieve data from the DataFrame, as all the data is serialized in the distributed KVStore system.

- DistributedArray: This class will hold reference information about data throughout the system. The purpose of this class is to bridge data that lives in different areas of the system. This class also holds a KVStore reference, so that it can look up data not stored locally.
    - Fields:
        - store_ (KVStore): to get data from the distributed system
        - keyList_ (KeyArr): this array will hold all Key values of interest
        - cache_ (Cache): this object will hold the data (Values) for some of the Keys within this distributed array. Before sending a data request to a key-value store, we will first check to see if the data exists in the cache.
    - Methods:
        - bool containsKey(Key): does the distributed array's key list contain the given key?
        - void addKey(Key): add the given key to the distributed array's list of keys
        - get(Key, size_t): get the value for the specified key from the distributed array. Check the cache first and if the cache does not contain the data, then make a call to the KV store; one for each type
        - Key getKeyAtIndex(size_t): return key at the given index in the array
        - void setStore(KVStore): this method will be used to pass a store into the distributed array; the store will come from the dataframe initially and move down the hierarchy (dataframe -> columnarray -> column -> distributed array)
        - Serialization and deserialization methods
- Cache: class that represents a cache within a distributed array. Holds the deserialized data of some of the distributed array's keys. The cache is limited in size and will be refreshed occasionally. We currently use a FIFO strategy to refresh the cache; if we want to add a new item to the cache but it is full, then we remove the element that has been in there the longest. Decided to store the deserialized data for improve retrieval efficiency.
    - Fields:
        - maxSize_ (size_t): defines a maximum size of the cache; the cache's map cannot hold more than
          this amount of elements at any given time
        - data_ (Map): this map will hold the data within the cache. The keys will be Key objects, and they will map to the deserialized data.
        - keyOrder_ (Queue): this queue will keep the order in which keys are added to the cache. This will help us update our cache by adding and removing elements on a FIFO basis
    - Methods:
        - bool containsKey(Key): does the map contain data for the given key?
        - Block getBlock(Key): get the mapped deserialized data for the given key

        - void put(Key, Block): add the Block, with the given key, to the map. If map full, replace an element
        - void clear() - remove all entries from the map

- Application: this is the highest level of the program. The Application is what the user interacts with, in which he or she defines operations to perform over a certain set of data. An application will run on each node. To specify operations, Application can be subclassed
    - Fields:
        - kv_ (KVStore): this will be the local kv store for the node that the application is running on. When initialized, it will take in the node id from the Application's and set it as its id
        - idx_ (size_t): node index that this application is running on
        - net_ (INetwork): network node for this application; passed into constructor but steals ownership
    - Methods:
        - size_t this_node(): returns id of this node
        - KVStore getStore(): retrieve KVStore from application
        - INetwork getNetwork(): retrieve network from application
        - run_(): given the node id, perform a certain set of operations. The node id will be placed in a switch statement, and each case will contain a different helper for each application running (ex.producer to initialize and create the data, summarizer to perform some operation on the data). Can retrieve and read data from the KV store
    - Current applications supported
        - Trivial: simple single-noded program. Creates a dataframe with a large amount of elements in a single column, then makes sure that the get methods on the dataframe retrieve the proper data.
        - Demo: currently runs with three nodes. Some nodes are putting data into the KVStore, some are retrieving data from the KVStore, and others are doing both. Program tests that the KVStore can be read from and written to concurrently by using messaging in a network layer.
        - Wordcount: tests the functionality of fromVisitor, which allows the user to pass in a writer and generate a dataframe by visiting that writer. This program tests a Summer writer, which takes in an SIMap of words that appear in a file to the number of occurrences in that file.
        - Linus: calculates the degrees of Linus. Uses the sorer to read in large .sor files describing the users, projects, and commits to those projects

- Utilities: We have a variety of common classes used throughout the code base
    - Object: base class of whole program. Provides method signatures for equals, hash, and clone. Class provided to us
    - String: represents implementation of a string. Provided to us
    - Array: represents implementation of an array. Used for storing a collection of objects
    - Map: represents implementation of a map. Used for when we want an item to map to a specific value

        - Various implementations depending on needs - SIMap (String-Num), MapStrObj (String-Object)
    - Queue: represents implementation of a queue. Used when we want to store and retrieve data on a first-in, first-out basis
    - Thread: represents a thread of execution. Allows program to split into multiple, "simultaneous" tasks
    - Args: class for handling command line inputs. Sets the field based on what is provided on the command line. Use of "extern" allows entire program to access these elements. We run an entry .cpp file that passes the arguments into this Args object and then starts the program based on the configurations.
    - DataType: our enum for the four data types
    - DataTypeUtils: contains static methods relating to data types; currently support converting chars to data types and vice versa

Use cases

The following is an example of a use case of the eau2 system. The lines of pseudocode are an outline of what each node will run.

```
Application:

    dataA = Key("dataA", 0);
    dataB = Key("dataB", 1);
    firstDifPos = Key("firstDifPos", 2);
    dataAFirstDif = Key("dataAFirstDif", 2);
    dataBFirstDif = Key("dataBFirstDif", 2);
    same = Key("areSame", 2);
    sameSize = Key("sameSize", 2);

    ProducerA (node_num = 0):
        Sorer s = new Sorer();
        DataFrame* df = s.read("fileA");
        // store df in distributed key-value store under key dataA
    return;

    ProducerB (node_num = 1):
        Sorer s = new Sorer();
        DataFrame* df = s.read("fileB");
        // store df in distributed key-value store under key dataB
    return;

    Comparer (node_num = 2):
        DataFrame* dfA = waitAndGet(dataA);
        DataFrame* dfB = waitAndGet(dataB);
        int areSame = 0;
        if (size of dfA != size of dfB) {
            DataFrame::fromScalar(&sameSize, &kv, 0);
            DataFrame::fromScalar(&same, &kv, 0);
            return;
        }
```

```
            DataFrame::fromScalar(&sameSize, &kv, 1);
            // loop through elements
            if (dfA->get(i, j) not equal dfB->get(i, j)) {
                int* pos = new int[2];
                pos[0] = i;
                pos[1] = j;
                DataFrame::fromArray(&firstDifPos, &kv, 2, pos);
                // generalize for any type
                DataFrame::fromScalar(&dataAFirstDif, &kv, dfA->get(i, j));
                // generalize for any type
                DataFrame::fromScalar(&dataBFirstDif, &kv, dfB->get(i, j));
            }
            DataFrame::fromScalar(&same, &kv, 1);
        return;

        Reporter (node_num = 3):
            DataFrame* sameDF = waitAndGet(same);
            if (sameDF->get(0, 0)) {
                pln("Same Data! :)");
                return;
            }
            pln("Different Data.");
            DataFrame* sizeDF = waitAndGet(sameSize);
            if (sizeDF->get(0, 0)) {
                pln("Non-Equal Size");
                return;
            }
            DataFrame* posDF = waitAndGet(firstDifPos);
            DataFrame* aDif = waitAndGet(dataAFirstDif);
            DataFrame* bDif = waitAndGet(dataBFirstDif);
            pln("Different Data!");
            p("Example: ");
            p("dataA[").p(posDF->get(0, 0)).p(", ").p(posDF->get(0, 1)).p("] = ")
                .p(aDif->get(0));
            p(" , dataB[").p(posDF->get(0, 0)).p(", ").p(posDF->get(0, 1)).p("] = ")
                .p(bDif->get(0));
            pln("");
        return;
```

Status

We have decided to use another group's Sorer implementation, as ours was written in Python and not fully correct. The group was chosen based on the results of our testing in Assignment 5, Part 1 -> https://github.com/yth/CS4500A1P1 As part of incorporating their Sorer into our codebase, we discovered a bug that we fixed locally and notified the team so they could fix their codebase.

We succeeded in creating an adapter to use their Sorer with our DataFrame classes; a test demonstrating this can be found in tests/sorerTest.cpp. We use the datafile.sor file for this test, which can be found in the data directory. We recently updated our adapter to set the rows of the dataframe by using a writer (SorWriter); the writer finds the element in the file and sets it in the row passed in to visit.

We have both a real TCP network layer implementation and a pseudonetwork implementation (spawning a thread to represent each "node", passing messages between themselves on shared MessageQueues)

Our Trivial, Demo, and Wordcount applications work fully on both our pseudo and real networks. For Wordcount, we use the test.txt file, which can be found at the top of the repository.

Our Linus program needs to undergo more testing. The program works for a small test case we created, with only four users and four projects. However, we noticed for larger files (1,000,000 line sor files), we get different outputs depending on the number of nodes running. For each number of nodes, the output is consistent. We believe there is an issue with our local_map implementation, so we'll need to add more tests for that (DemoLocalMap highlights the issue). Running Linus with the full files currently takes up a large amount of RAM; if possible, we'd like to determine a way to reduce this RAM usage.