CS 694a, Fall 2011: Project, Part 1: CaseBook

Introduction and Logistics

In this project, you will build a version of Facebook using cloud computing technologies. Specifically, you will be given a "cluster" of virtual machines on which to run the front-end for your Facebook implementation, which must use the Cassandra key-value store (hence the name CaseBook for the project). The project will be due at 5 PM on October 7, 2011. You may choose to do the project individually, but we recommend forming a team of 2 people (larger teams are not allowed).

The Project

You are expected to implement the following functionality:

- support for multiple CaseBook *users*: each user has a single unique ID (consisting only of alphanumeric characters), and a name which may or may not be unique
- each user has one or more *friends*, comprising their social network
- each user has a *wall* on which their posts, or those of their friends, must appear. Each post can be at most 2K bytes in size.
- each user has a photo album to which he or she can upload any number of photos. Each photo can be at most 2MB in size.

This is clearly a simplified version of Facebook, since you do *not* have to implement any of the following functionality:

- Facebook email or chat
- support for multiple Facebook apps
- scaling to 500 million users!

Getting started on the project

To get started on the project, we recommend downloading and understanding the two pieces of code described below.

Cassandra

As discussed above, Cassandra is a "key-value" store. In a keyvalue store, you can "put" a *value* associated with a specific *key*, and *get* the value associated with a specified *key*. For scalability reasons, keyvalue stores are implemented in a distributed manner across large cluster.

Cassandra is a project developed by the Apache software foundation, and you can download it here. We suggest you start soon, download Cassandra and play with it in single-machine mode. In a couple of weeks, we will make a cluster of machines available to you for more extensive experimentation.

You have several options for programming Cassandra. Cassandra is built on top of the Thrift API, but several other clients exist for it. If you would like to build on top of Twissandra (see below), you can use the Pycassa client API designed for Python. You may also want to check out Cassandra's command-line interface. Your choice of client API may impact performance and scalability, so you might want to make this choice carefully.

Twissandra

Twissandra is an example of a Twitter-like service that uses Cassandra as the backend storage. Twissandra provides many of the abstractions that you will use for CaseBook: users, the social network, and posts (or tweets, in Twissandra's case). Also, Twissandra's implementation contains the components needed to host a Twitter-like service—a web server, a back-end database, etc.

You may either build CaseBook on top of Twissandra, or choose to implement it completely from scratch. As described below, your goal in the project is to push on the *scale* of your implementation; it is not clear if the Twissandra implementation supports the scales that you will need. Even if you choose to implement Casebook separately, you should definitely examine the Twissandra code which will give you a good idea of how to structure Cassandra-based applications.

If you choose to build upon Twissandra, then you may find these instructions on how to install Twissandra useful.

Implementing the Project

As discussed above, you will implement four elements of Facebook: users, their social network, wall posts and photo albums. In a real implementation, users will populate the site with their information. However, for us to test your project, we require that you implement functionality that would let us us *load* your CaseBook implementation with input data (see below for details).

The goal of the project is to push on *scale* and *efficiency*. What this means is that you implementation should be able to *load* as large a social network as possible within about *1 minute*. Specifically, your implementation should support, for the largest value of X possible:

- X users
- 10 friends per user, on average (some users may have more than 10, some fewer than 10)
- 100 photos per user, on average
- 1000 wall posts per user, on average

The points you get for the project will be proportional to the value of X that your implementation is able to achieve. Additionally, as described below, page load times in your implementation must be reasonable, otherwise you may lose points for it.

To achieve these performance optimizations, you may need to pay careful attention to how you structure your implementation and use parallelism when possible. To understand the performance of your system, you may need to profile it to determine where the bottlenecks are. If you notice bottlenecks in Cassandra itself, you are free (but not required) to modify Cassandra to improve the performance of your system.

You are given considerable freedom in implementing the project. For example, you can decide how much of the functionality you want to put into a front end server and how much (if any) to put in backend servers. However, you can considerably simplify the GUI, since this class is not about user interface design. You are free to choose a simple profile page/GUI. As you will notice, the Twissandra GUI is much simpler than the actual Twitter page, and is a good example of the kind of simplifications you should be looking at.

Input

`http://<home>/`. This homepage for your CaseBook site can have minimal content (e.g., a welcome message). In addition, when a user visits `http://<home>/LOAD/`, your implementation should load the configuration file from `http://enl.usc.edu/~cs694/casebook/config.txt`. Your implementation must execute the commands in the configuration file, and print "Configuration File Uploaded" only after all the configuration commands have been successfully executed. (While testing your implementation, you can use your own URL to download the configuration file. We will also place a small configuration file at the URL above, which you can test before submission).

Moreover, the home page should have a separate reset page `http://<home>/RESET/` which, when visited, , should completely erase the contents of CaseBook. This will enable us to repeatedly upload different configuration files to test your implementation. Please pay attention to the performance of the reset: it should take no longer to reset CaseBook then it takes to load the corresponding configuration file. The configuration file is a series of *commands*, one per line:

- `user <usertag> <name> <password>` is a command that defines a CaseBook user. `<usertag>` is an alphanumeric character string (no spaces) that uniquely identifies a user. `<name>` is a string (in quotes) that represents the actual name of the user and can be of arbitrary length. `<password>` is an alphanumeric string. For example, a valid input would be:

```
user rameshg "Ramesh Govindan" "foo"
```

- `friend <usertag1> <usertag2>` is a command that specifies that `<usertag1>` and `<usertag2>` are CaseBook friends. ~~The semantics of CaseBook friends is identical to that of Facebook.~~ Whenever a CaseBook user comments on their wall or updates their photo album, the comment or photo is visible on the wall of all of their friends.
- `wallpost <usertag> <posttext>` specifies that `<usertag>` posts on his or her wall a message containing `<posttext>`. The post is specified in quotes, and you can assume that each post is no longer than 2K bytes. An example of a valid `wallpost`command is:

```
wallpost rameshg "I saw a nice movie yesterday"
```

- `photo <usertag> <URL>` specifies that `<usertag>` uploads a photo available at `<URL>`. Your scripts should use photos available at `http://enl.usc.edu/~cs694/casebook/photos/` and you can assume that that URL will have 100 photos numbered photo1.jpg, photo2.jpg, etc. (Note that for testing the scaling of your implementation, you can always re-use photo images). Thus, an example of a valid `photo` command is:

```
• photo rameshg http://enl.usc.edu/~cs694/photos/photo10.jpg
```

Please keep in mind a couple of restrictions. First, your implementation should **NOT** use any client side code (e.g., Javascript). Second, you should store photos in CaseBook, **NOT** links to photos.

Output

As specified below, you will submit the largest configuration file that you have been able to upload to your CaseBook implementation in under *1 minute* (i.e., the upload should finish in under 1 minute). As discussed above, your upload program should return only when it has finished executing all the commands in the configuration file. If it returns earlier, and we are unable to examine any of the user pages (see below), you will be penalized.

Once the upload is done, we should be able to view users' walls and photo albums as follows:

- We should be able to go to `http://<home>/<usertag>/` to view the corresponding user's CaseBook page (you should present a password prompt to authenticate the user, if the user has not recently been authenticated). This should display the top 200 items on the user's wall (wall posts, photo album additions) in reverse chronological order, with an option to show 100 more items each time. The page load times for this page must be optimized; if it is noticeably slow, you may lose points.
- We should be able to go to `http://<home>/<usertag>/photos` to view the corresponding user's photos. This should display the top 20 items in the user's photo album in reverse chronological order, with an option to show 100 more items each time. The page load times for this page must be optimized; if it is noticeably slow, you may lose points.

Submitting the Project

On Blackboard, you will upload the following two files:

- A `report.pdf` that is no more than 1 page and contains two things:
  - Your `http://<home>/` URL
  - A brief description of your implementation structure and what optimizations you did in order to improve scalability
- A `config.txt` file containing the largest configuration file that safely works on your implementation and takes less than a minute to load.

More detailed submission instructions will be given later.

Acknowledgement

Abhishek Sharma came up with the idea for, and helped draft, the project description.

CS694a, Fall 2011: Project, Part 2: CaseBook Search

Introduction and Logistics

In this part, you are asked to add scoped search functionality to CaseBook, building upon your CaseBook implementation in part 1.

The project will be due at midnight on December 2, 2011.

The Project

You are expected to implement the following (deceptively simple) functionality:

- Search the wall posts of a user, his/her friends' wall posts, or all wall posts for keywords.
- Search photos of a user, his/her friends or all photos for photos with at least *m* faces in them, where *m* is a search parameter.

Implementing the Project

To implement this functionality, you may need to build search indices that speed up search response time. You may store these indices in Cassandra or any other storage structure of your choice. As before, your project will be graded on the *performance* of your implementation.

Your implementation must compute these indices on-line (as wall posts or photos are inserted using the configuration file) unlike some real systems where, because indexing is resource-intensive, it is often done in the background. Of course, this means that you may be able to load smaller configuration files than in Part 1: this is perfectly acceptable.

Finally, your implementation may need to use a face detection algorithm; you may use one of several available on the web.

Your implementation will be graded on 2 aspects of performance: the largest configuration file that can be correctly loaded in under *2 minutes*; and the correctness and response time of a variety of searches conducted on your implementation.

As with Part 1, to optimize performance, you may need to pay careful attention to how you structure your implementation and use parallelism when possible. To understand the performance of your system, you may need to profile it to determine where the bottlenecks are.

Input

As in Part 1, when a user visits `http://<home>/LOAD/`, your implementation should load the configuration file from `http://enl.usc.edu/~cs694/casebook/config.txt`. *The configuration file syntax for Part 2 is identical to that of Part 1*. Your implementation must execute the commands in the configuration file, build indices online and print "Configuration File Uploaded" only after all the configuration commands have been successfully executed and all necessary indices have been constructed.

After loading the configuration file, we will run several search queries on your implementation by encoding searches as URLs. These encodings will, in general, have the following format:

```
http://<home>/SEARCH?user=<u>&type=<t>&scope=<s>&terms=<tm>
```

The search parameters have the following semantics:

- The user parameter specifies that search is to be performed from the perspective of the user whose tag is <u>. This perspective is important for the scoped search (see below).
- The type of the search is either post or photo. The former specifies a search on wall posts, and the latter a search on photos for faces.
- The scope of the search can be one of user, snet, or glob. A user scope indicates that only <u>'s posts/photos should be searched, a snet scope indicates that all posts/phots by <u> and his/her friends should be searched, and a glob scope indicates that all posts/photos in the system should be searched.
- Finally, <tm> depends upon the search type. For a photo search, <tm> is an integer specifying the minimum number of faces required in a photo. For example, if <tm> is 2, then your search must return all photos with *at least 2 faces in them*. For a postsearch, <tm> is a sequence of search keywords, with spaces replaced by the + character. For example, foo+bar+baz specifies a search for posts containing *at least one* of the words foo, bar, or baz.

Output

All search results must be displayed on a single page in inverse chronological order. It is not necessary to spend a lot of time on formatting the search response, since this class is not about user interface design.

As discussed above, your implementation will be graded on two criteria. First, it will be graded on the performance of loading and indexing, specifically, the largest configuration file that can be loaded or indexed within *2 minutes*. More precisely, we should be able to issue the following command from a command-line in order to time the configuration loading time:

```
% time wget http://<home>/LOAD/
```

If necessary, we should also be able to reset your implementation using:

```
% wget http://<home>/RESET/
```

Second, your project will be graded on the completeness and responsiveness of the search. Specifically, all post queries must return all posts within the specified scope and matching the specified terms. However, it is acceptable for a photo search to be slightly inaccurate: face detection algorithms have small false positive and false negative rates. We will time the response to each search; faster searches will receive more points. More precisely, we should be able to time a search as shown in the following example:

```
% time wget "http://<home>/SEARCH?user=user1&type=post&scope=glob&terms=Churchill+Winston"
```

Submitting the Project

On Blackboard, you will upload the following two files:

- A report.pdf that is no more than 1 page and contains two things:
    - Your http://<home>/ URL
    - A brief description of your implementation structure and what optimizations you did for indexing and search response times
    - The largest X (number of users, see Part 1) configuration file that your Part 2 implementation can load.

More detailed submission instructions will be given later.

Acknowledgement

Abhishek Sharma came up with the idea for, and helped draft, the project description.