University of Glasgow | School of Computing Science

Who To Follow In Context

Darren Burns

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 23, 2016

Abstract

abstract here....

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Contents

Chapter 1

Introduction

1.1 Goal

The aim of this project was to create an application which provides a relevant set of Twitter accounts to follow for information about events. In particular, we are interested in recommending accounts which are relevant at the current moment in time. Therefore, a large portion of the project is concerned with providing real-time updates and corrections to recommendations based on the evidence provided via a stream of Twitter statuses.

1.2 Motivation

Guiding users towards the information they require is becoming increasingly important aspect in the development of online services such as social media and e-commerce websites. Without this guidance, users are much less likely to find what they are looking for, and therefore less likely to remain engaged with the service.

The impact of this guidance should not be understated: Celma & Lamere (ISMIR 2007) noted that 2/3 of movies rented via Netflix were through recommendation, and 35% of sales made via Amazon are through recommendation. For large companies, this may equate to billions of pounds of additional revenue.

From a social media perspective, recommending interesting people to follow is exceptionally important in retaining users. A Deutsche Bank survey recently showed that the primary reasons people quit Twitter are directly related to the service not meeting their information needs. The top two reasons cited were:

1. *"I was getting the information from somewhere else."*
2. *"There was no useful information on Twitter."*

Hence, by directing users towards Twitter accounts which meet their information needs, it is hypothesised that they are less likely to quit the service. For social media platforms in particular, retaining users is of vital importance since it is directly associated with their primary source of income: advertisements.

Although several products exist today which attempt to solve this problem, they tend to disregard the quickly evolving nature of live events by providing a static set of results. Given the rapidly evolving and sometimes fleeting nature of events, immediate consideration of new tweets may give users insight into how new "event experts" emerge as they become relevant, and how that relevance changes over time.

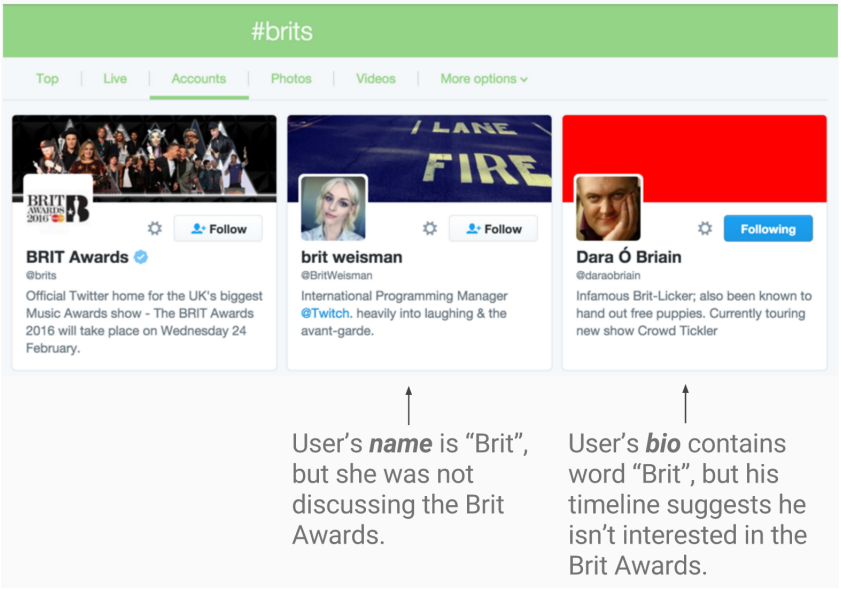

Figure 1.1: Twitter's own search service. Searching for a hashtag appears to either disregard or give very little weight to users who use the hashtag in their tweets.

1.3 Related Products

1.3.1 Twitter Search

Twitter's existing search service (shown in Figure 1.1) allows searching for accounts based on a search query. This works similarly to traditional search engines in that the user enters a query and the service returns a list of users that may satisfy the information needs based on that query. While this service works well for suggesting users based on their *long term* interests, it is not so useful for an ongoing event which may only occur for a few hours. The "Who To Follow In Context" project displays and updates results in real-time, and evidence from new tweets is applied immediately. Therefore if a query result page is left open, the ranking of accounts will be adjusted as people start and stop discussing the query topic, providing users with an insight into how new participants in the discussion emerge.

1.3.2 Who To Follow at Twitter

"WhoToFollow" was the service used at Twitter which recommended accounts to follow based on the people already followed by an account. The primary difference between this project and Twitter's initial Who To Follow system is that this project relies on information retrieval methods which rely entirely on the content of the tweets that users write, whilst Twitter relied on graph analysis, which examines interactions between users to try and identify users similar to those the author already follows. Since the initial deployment of this service in 2010, Twitter announced they are reimplementing the service using Pig and Hadoop to improve scalability, and also to apply machine learning approaches using a wider variety of signals.

1.3.3 Klout

Klout provides a well-known system for ranking the influence of users across numerous social media platforms and online communities. It takes into account over 3600 features from these sources, and processes around 500

million user interactions each day. Although not directly related to this project (Klout ranks users based on a static context: *"How influential is this user?"*), Klout's well documented architecture and popularity in the domain make it worthy of mention.

1.4 Relevant Research

1.4.1 Searching For Quality Microblog Posts

This paper examines an array of metrics for examining whether a post on a microblog is high quality or not. It introduces an array of features which can be extracted from tweets, and examines the impact of the reputation of external URLs linked to from the tweet. Many of the techniques examined in this paper are applied in the filtering stages of this project, in order to discard tweets which do not meet a threshold on the extracted features.

Chapter 2

Requirements

Requirements were gathered during the initial meetings between the author and the project supervisor. These requirements were frequently updated as new considerations were made and new restrictions realised. To ensure the customer (the project supervisor) that progress was being made in satisfying these requirements, weekly status reports were produced during the implementation stage.

2.1 Functional Requirements

Functional requirements were gathered in order to describe the specific behaviours the system may offer. In order to prioritise these requirements, the *MoSCoW framework* was used. That is, requirements were categorised in terms of their importance into the sets “must have”, “should have”, “could have”, “won’t have”.

2.1.1 Must Have

Requirements listed in this section were vital to the success of the project. Without satisfying all of these requirements, the project would be regarded as unsuccessful.

The system *must have* the following capabilities:

1. Present a web-based user interface providing querying and result-viewing capabilities.
2. Read tweets live from the Twitter Streaming API.
3. Extract the following features from incoming tweets:
   (a) The length of the tweet.
   (b) The number of correctly spelled words in the tweet.
   (c) The number of capitalised words in the tweet.
   (d) The hashtags used in the tweet.
   (e) The number of “likes” the tweet had at the time it was processed.
   (f) The number of retweets the tweet had at the time it was processed.
   (g) The number of URLs the tweet contains.
   (h) The number of users mentioned in the tweet.

4. Filter out Twitter users who post low quality tweets.
5. Index tweets and relevant metadata so that they can be quickly retrieved given a user query.
6. Display the results of the query as they update in real-time.
7. Display the timeline of Twitter accounts suggested by the application, without leaving the application.
8. Retrieve user suggestions based on an event identifying "hashtag" query.

2.1.2 Should Have

"Should have" requirements were considered important for the project, however they are not given the same level of importance as "must have" requirements. The exclusion of some of these requirements would not result in an unsuccessful project.

The system *should have* the following capabilities:

1. Read historical tweets from a file to improve evaluation capabilities.
2. When reading tweets from a file, any user timelines viewed should be a snapshot of their timeline corresponding to the time the tweets in the file were sampled from.
3. Alter its behaviour according to options defined in a configuration file.
4. Allow users to provide feedback in order to evaluate and improve the relevance of future results.
5. Provide users with visual feedback to inform them of the current status of the system.
6. Store tweet features and relevance judgements for evaluation purposes.
7. Examine a user's timeline to for additional tweets that can be used as evidence for their relevance.

2.1.3 Could Have

"Could have" requirements are those that were not considered important, but could improve some aspect of the project such as result relevance or user experience.

The system *could have* the following capabilities:

1. Show users any new queries as soon as they are entered in order to encourage interaction.
2. The reputation of the URLs the tweet contains.
3. The sentiment of the tweet (e.g. number of positive or negative emoticons used).
4. Show how user relevance changes over time.
5. Counts for several punctuation marks used within the tweet. Research has suggested that this may be an indicator of microblog post quality.

2.1.4 Won't Have

"Won't have" requirements are those which the student and project supervisor agreed should not be implemented.

The system *won't have* the following capabilities:

1. Each query should result in a live stream of relevant tweets being displayed alongside the suggested results for that query. This feature was not implemented due to the restriction imposed by Twitter that each application can only have a single stream open.
2. Provide the ability to "follow" users when providing relevance feedback

2.2 Non-Functional Requirements

In addition to the functional requirements defined in the previous section, as set of non-functional requirements were agreed on. These requirements describe criteria on how the system should be judged, rather than examining the behaviour it offers. The application was implemented and designed taking these requirements into account as much as possible.

1. Support multiple users concurrently without there being a noticeable impact on performance.
2. Support multiple queries concurrently without there being a noticeable impact on performance.
3. Be robust enough to handle a variety of issues such as network problems without failing.
4. Be constructed in a scalable manner so that it can easily be extended to run on a cluster of machines if required in the future.
5. Be constructed in a maintainable manner.
6. Display a response (whether that is results or an indication that no results were available) to queries within one second of them being issued.
7. Operate within the restrictions imposed by the Twitter REST API v1.1 rate limits.

Chapter 3

System Architecture

The architecture of the system was designed to adhere to the gathered requirements as far as possible.

3.1 Server Architecture

On the server side the architecture consists of multiple different components each with their own distinct responsibilities. The design of the server can be seen in Figure 3.1, and the remainder of this section refers directly to this diagram.

3.1.1 Programming Language

The server is written using the *Scala* programming language, which is a hybrid of the object-oriented and functional paradigms. Scala was chosen for its lightweight syntax and the fact that it compiles to Java Virtual Machine bytecode, allowing for clean interoperability with existing Java libraries and frameworks. In fact, a number of extremely popular Java libraries are written in Scala, including Apache Spark, Akka, and the Play Framework. Additionally, Scala provides powerful means of asynchronous programming through `Future`s and `Promise`s, and its `Option[T]` type and powerful pattern-matching features make `NullPointerException`s impossible since it explicitly requires the developer to check for the possibility that the `Option[T]` type contains a `None` value. Scala also benefits from its powerful REPL (Read-Execute-Print Loop) which comes bundled with the Scala compiler. This allows developers to immediately evaluate expressions and see the results. Existing code can even be loaded into the REPL so that the effects of performing actions on user defined objects can be examined.

Other languages examined in the initial investigations were Python, NodeJS, and Java. Python was ruled out due to its poor support for concurrency, and its dynamic typing would likely have caused difficulty as the complexity of the project source code increased. NodeJS was considered for its renowned support for real-time web applications. However, the ecosystem surrounding the language has not expanded into the field of processing data streams. As with Python, NodeJS is dynamically typed, and this also factored into the decision to rule it out. Finally, Java was considered due to its unmatched ecosystem of libraries and frameworks and its type system. Java however often requires extra boilerplate code in comparison to Scala. Additionally, Java libraries make excessive use of `null` values, which greatly increase the possibility of run-time failure.

3.1.2 Concurrency Model

The real-time nature of the application means that if performance is poor then it will immediately impact user experience. Should feature extraction for a user timeline take longer than expected, then a user will have to wait until the system gets around to processing that user. As such, high levels of concurrency were deemed vital in order to meet user requests in a timely manner. To accommodate such a requirement, the system was designed using the *Actor model*.

The Actor Model

Actor systems provide an alternative means of concurrency that avoid the pitfalls of typical synchronisation methods such as the use of shared mutable state and locks. The actor model of concurrency was first popularised by the Erlang programming language, and has increased in popularity in recent years with the release of applications such as WhatsApp which use the model extensively. An actor system consists of a set of actors. Actors are entities capable of performing some computation on a thread in response to messages. Actors also have the ability to create new child actors, and to send messages to other actors in the system, who will then react in their own way depending on the information contained within that message. By communicating between actors solely through asynchronous *immutable* message passing, we guarantee that the computations performed by a single actor cannot result in issues such as thread interference.

Akka

Akka is a framework for Scala and Java which implements the actor model. Despite vastly simplifying the concurrency aspect of the implementation, Akka required very little overhead. The actor model framework provided by Akka supports passing approximately 50 million messages per second on an average machine, and around 2.5 million actors can exist per gigabyte of heap space. Each component of the architecture was implemented as an actor, and communication between components in the system was facilitated through message passing.

Despite being primarily used for its concurrency model, Akka provided a number of features "out-of-the-box" which satisfied several project requirements. Actors in Akka have a "supervision strategy" which defines what actions should be taken when an exception occurs within them. This allows the system to meet the high degrees of fault tolerance and resilience required. The default supervision strategy (simply restarting the actor when an exception occurs) solved many issues that were encountered. For example, it was common for the Twitter streaming API to become unavailable (due to network issues) and an exception was therefore thrown. The actor handling the stream was automatically restarted and the stream handle resent to all of the actors that required access to it. No explicit error handling code was required, yet the application was able to self-heal and become fully functional as soon as the streaming API became available again.

3.1.3 Reactive Systems

The architecture of the system is implemented to conform to many of the tenets described in the "Reactive Manifesto". That is, the system architecture is designed to be responsive, resilient, elastic, and message-driven.

Responsiveness

Responsiveness is a key aim of the application. Messages passed from actor to actor in potential bottlenecks are load-balanced in order to minimise the work queued on a single actor or thread and maximise throughput. Within each defined actor is a response time guarantee which, if not met, causes the actor to be restarted in attempt to regain responsiveness.

Resilience

The application provides the guarantee of resilience through "self-healing". Should an actor executing some computation encounter an exception, its parent is notified and takes the appropriate action in order to cleanly recover. Additionally, if an exception occurs within any actor, it is treated in isolation, and other actors are not affected unless the parent actor's supervision strategy explicitly calls for it. Therefore, problems are compartmentalised and only closely related actors are affected.

Elasticity

Some actors within the system are implemented as state machines with the ability to change their behaviour at run-time, in response to certain events. This can be useful, for example, if the application hits the Twitter API rate-limit. The behaviour of the corresponding actor can be modified to stop making API requests for a specified period of time, and then modified back to its original API-reliant behaviour after this time period has expired.

Message-Driven

The application is entirely message driven. The root messages are batches of tweets read from a stream of tweets, or user interactions with the system. Each of these messages results in a chain of messages being sent asynchronously between actors as they handle the situation.

3.1.4 Dependency Injection

The system relies heavily on the *dependency injection* design pattern. This pattern greatly reduces coupling between components, and enforces a separation of concerns by separating dependency creation from behavioural code.

Google Guice

Google Guice is a dependency injection framework for Java which manages the dependency graph that exists between objects in an object-oriented environment. Using Guice, one can request an instance of an object when required by "injecting" the instance into a "setter" method or a constructor.

Guice was primarily used for the injection of actor references into other actors. If we ever wish to send a message to an existing actor in order for it to perform some computation, we simply ask Guice for the instance of the required actor by injecting the actor into the constructor of the current class using the `@Inject` and `@Named` annotations provided by the framework.

3.1.5 Feature Extraction and Filtering

Incoming tweets are initially passed into the feature extraction component. *Spark Streaming* (and Twitter4j for authentication purposes) provided a convenient way to access the Twitter streaming API, and distribute the feature extraction workload across multiple processing cores. Spark is designed to work in a distributed environment, and therefore also leaves open the possibility of distributing the stream across multiple machines in a cluster should the need arise.

This component extracts statistics from the content of the tweet and its metadata. These features are passed to the filtering component which will discard tweets which do not meet a configurable quality threshold. The purpose of filtering these tweets is to prevent unnecessary computational load and also to avoid hitting the rate limiting restrictions of the Twitter API.

After low quality tweets have been filtered, the features of the remaining tweets are stored for future use, and the remaining tweets are progressed to the indexing stage of the pipeline.

Both the feature extraction and filtering components may receive tweets from other locations within the system. If the system ever deems a user to be relevant (for example, if they appear in a result stream for an open channel, or a user of the system views their profile), then the tweets present on that users timeline are fed back into the pipeline in order to extract further evidence in the hopes of improving the relevance of results.

3.1.6 Persistence

Two different technologies provide the persistence layer for the application. Each of these provide different benefits and so the decision of which to use and where varied depending on context.

Redis

Redis is an in-memory data structure storage engine which is used for storage of extracted features. By keeping data in memory it enables quick reads and writes. Its built in data structures simplified the implementation of certain features. For example, a Redis SortedSet data structure is used for the calculation of the median time since the user last made use of the query term as a hashtag. Since the application must process vastly more reads than writes for individual users, allowing Redis to perform the sorting step at write time make the system more efficient than if we were required to sort an array each time the latest results were retrieved for a user.

MongoDB

MongoDB is a NoSQL database used for the caching of Twitter user metadata in order to reduce the number of API requests required. For example, we store the URL of the user's profile picture and timeline colour in MongoDB and check for its existence here before making an additonal API request for the information. We also store user relevance feedback in MongoDB for evaluation purposes.

NoSQL was used due to the lack of need of relational features. Each category of item stored in this database was entirely independent from the others, and thus guaranteed that relational database tools such as joins would not be necessary.

3.1.7 Indexing and Retrieval

After low quality tweets have been discarded, those remaining are sent to the indexer where they will be stored for efficient retrieval.

When a query is received, the Query Service will construct a result stream between the information retrieval engine and the user who entered the query. The information retrieval engine will then be frequently polled for new results, and the new results will be sent to the user as soon as they are available. Any user which enters the same query thereafter will see the same stream of results.

Terrier Information Retrieval Platform

Both indexing and retrieval capabilities are provided by the *Terrier Information Retrieval Platform*. Terrier's `MemoryIndex` allowed an index to be maintained online, and thus was the technology which enabled results to be displayed and updated in real-time.

3.1.8 Metrics Reporting

Throughout different stages of the pipeline, different metrics can be collected. These metrics are both logged and sent to the user in real-time to provide insight into the operation of the system. This is important as it ensures the user that the system is working as expected. If a large period of time passes and no new users have appeared in the stream of results they are viewing, they might think that something has went wrong. By keeping them constantly updated on the number of users the system has seen, they are ensured that it is working as expected.

Numerous different technologies were examined for providing this functionality, including the *Comet model* in HTTP 1.X, the WebSocket protocol, and using the persistent connection and multiplexing capabilities of HTTP 2.0.

Comet sockets were introduced to provide streaming capabilities before the introduction of the WebSocket API, and as such rely on many browser hacks to work properly. However, with the correct hacks this provides excellent compatibility with older browsers which do not support WebSockets. Facebook Messenger is one such service that (as of 2013) relied on Comet for providing real-time notification and chat.

HTTP 2.0 is the latest version of the HTTP specification which provides an abundance of new features and exceptional performance improvements over the currently dominant HTTP 1.X protocols. These features (such as connection multiplexing) mean that the "hacks" required with the first version of the protocol which enable streaming are no longer required. However, support for HTTP 2.0 in existing libraries and frameworks are limited as of the time of writing. As such, relying on HTTP 2.0 would have added too much development overhead to make it practical.

After consideration of the above technologies, the WebSocket protocol was chosen to enable real-time communication between the server and connected clients. The WebSocket protocol is well supported in modern browsers, and is specifically designed for immediate communication between servers and clients. Additionally, WebSockets are well supported in modern web frameworks meaning there is a large community of developers and supporting documentation available.

A downside of using a WebSocket reliant architecture is that the protocol is much slimmer than HTTP, and therefore it was required to implement the notion of a "keep-alive" to ensure that channels are cleaned up in the case that no client registers an interest in it.

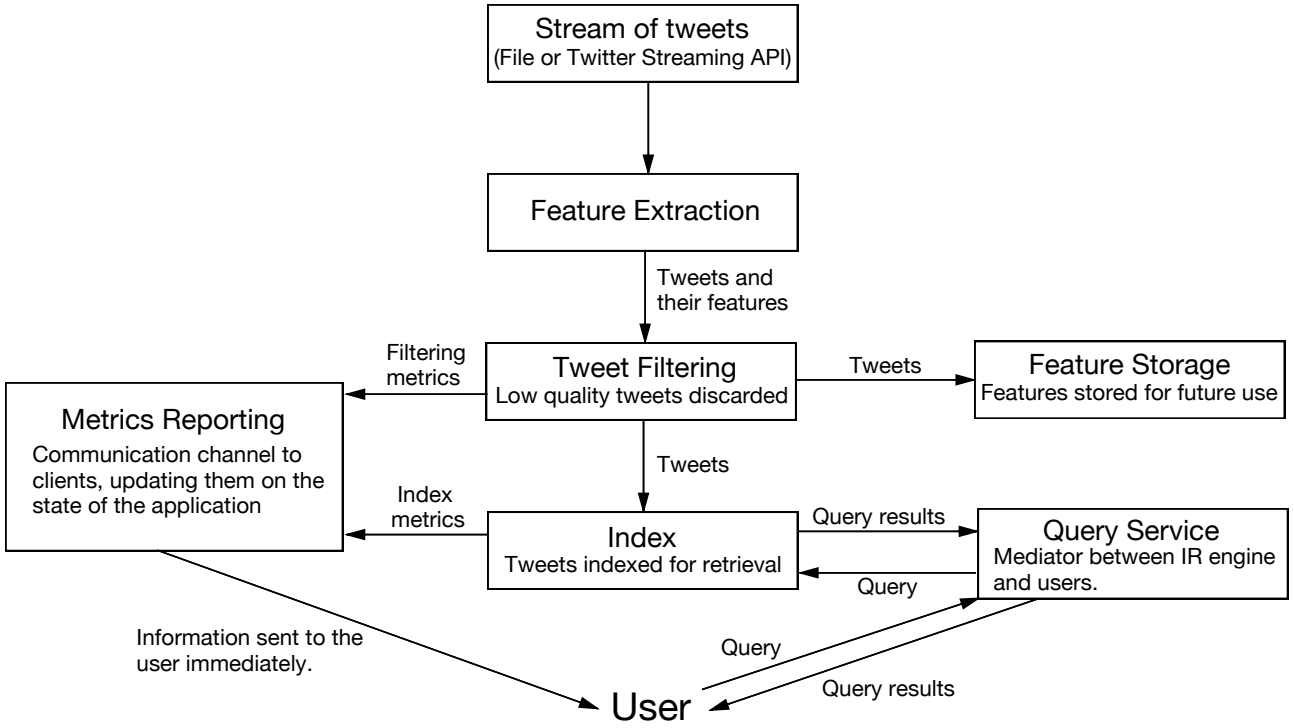

Figure 3.1: The high-level architecture of the system. The "Implementation" chapter looks at the internal workings of each component in detail.

Play Framework

The *Play Framework* was used for its WebSocket implementation and to create REST API endpoints, such as the endpoint used in fetching the timeline and metadata for a user when they are clicked on in a result set. Since routing and rendering was done on the front-end, only a small subset of the features of Play were required. This framework provided the benefit of having excellent integration with Akka (Play is also written using Akka) and Guice, allowing WebSocket connections to be accepted using actors. An additional benefit of using this framework is that it is one of the most popular web frameworks for Java and Scala, and as such it has an active developer community with clear and extensive documentation.

3.2 Front-End Architecture

3.2.1 Programming Language

The programming language used to create the front-end of the application was *TypeScript*. TypeScript is a typed superset of JavaScript which provides compile-time type checking which eliminates a large number of run-time errors. Existing JavaScript libraries can be used from within TypeScript by including a "type definitions" file in the project. The purpose of such files is to provide a mapping from the non-typed JavaScript constructs to their corresponding TypeScript types.

The front-end was initally written in JavaScript and then migrated to TypeScript as the number of type related errors began to increase due to the increasing complexity of the front-end. Although the initial migration to TypeScript was extremely time consuming, the number of run-time errors encountered significantly decreased. Since

TypeScript is a type checked language, we also gain the benefits of code-completion and refactoring capabilities within certain integrated development environments.

3.2.2 Single-Page Application

The client side of the application is constructed as a single-page application (SPA) meaning that the entire application is mounted at a single URL (excluding API endpoints which return JSON rather than text/html content). Navigation between pages is then managed entirely by client-side code. Additionally, the entirety of the views are implemented on the client side, and data is fetched either via a REST API (as opposed to server-side rendering of views) or through an open WebSocket.

This approach provides the benefit of reducing the number of requests made to the server, and greatly improves the responsiveness of the front-end.

Although we gain an overall more responsive experience using this architecture, it does have some associated flaws. The initial page load is often slower, since the client contains more code than it otherwise would if rendering and routing was managed on the server. Additionally, the improvement in page load times can place a burden on the server, since it has to ensure that the data will be available when a user requests it. If the data is not available when required by the user, it may prove irritating as they wait on certain portions of the interface to update.

3.2.3 Component-Based Design

React is a JavaScript library developed by Facebook for constructing user interfaces in a component-based fashion. There is an abundance of modern libraries and frameworks for developing user interfaces. React was chosen primarily for its enforcement of the idea of "separation of concerns" and its renowned performance.

The other popular client library considered for the project was AngularJS by Google. However, Angular is framework rather than a library and as such forces the developer to write their application within its restrictions. Additionally, Angular is well known to have considerably worse performance in comparison to React, due to it taking considerably more time to perform DOM (Document Object Model) manipulation operations.

React Router

The decision to make routing a concern of the front-end was driven by the existence of *React Router* and the fact that the application view has a limited number of overall states. Using this library allowed views to be declared hierarchically, so that only required subtrees of the view are re-rendered on a URL change.

Chapter 4

Implementation & Design

4.1 Server Implementation

4.1.1 Tools & Practices

Git

Git is a distributed version control system which was used extensively during the development of the project. A working implementation of the project was always maintained on the `master` branch. Features were developed on their own branches so as to not interfere with `master`.

GitHub

The project repository is also linked to a remote repository hosted on *GitHub*.

Agile Software Development

The *Agile* software development process was followed throughout the lifetime of the project. Although there was an initial requirements gathering stage, these requirements frequently changed, and the product itself changed as a result.

Kanban & Trello

Project task management was done using the "Kanban" system, where projects are represented as a set of named lists and cards. *Trello* is an online platform which implements this system, and it was used extensively throughout all stages of the project.

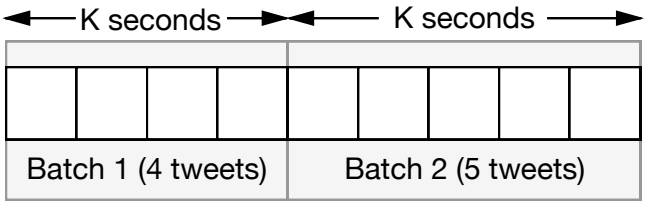

Figure 4.1: Tweets are batched temporally and sent throughout feature extraction together.

SBT (Scala Build Tool)

SBT is a build management system commonly used in Scala projects which is similar to popular Java build tools such as Maven and Gradle. SBT was used in the project to maintain and resolve dependencies, run tests, and to build and run the project.

4.1.2 Feature Extraction

Spark Streaming provides us with a `ReceiverInputDStream[Status]` (discretised stream) representing the incoming stream of tweets. The tweet stream was mapped to a "windowed" tweet stream. This allows us to view the stream through a "sliding window", and perform operations on these individual windows of tweets. The tweets in each window are batched and sent through the rest of the pipeline together. The width of the sliding window is defined temporally, as is shown in Figure 4.1.

The `FeatureExtraction` actor maps our windowed stream of tweets into a windowed stream of tweet feature vectors, which is immediately filtered to discard those tweets which do not meet the threshold quality criteria. These features are then sent to Redis for storage, and the original status is sent to the indexing actor for storage in a Terrier `MemoryIndex`.

In early iterations of the project, features were stored in MongoDB. However, this proved to be a bottleneck on performance, since the implementation was unable to keep up with the rate at which new features were extracted. There are several reasons which explain why this situation occurred:

- MongoDB persists data on disk, resulting in slow read/write times compared to in-memory solutions.
- An asynchronous MongoDB driver was not used and so blocking occurred frequently.
- MongoDB does not have built-in data structures that could be used to avoid additional computations.
- Parallel insertions were not used. The MongoDB Scala driver had no serialisation/deserialisation support and this made using the parallel insertion API overly cumbersome.

To overcome this bottleneck Redis was employed as the storage engine for features. This resulted in immediate performance improvements, but the system was still unable to store features at the rate they were being generated. Three additional steps were taken in order to ensure that features were persisted at an acceptable rate:

- Connection pooling was used to ensure that the application did not have to reconnect to the Redis server with each request.

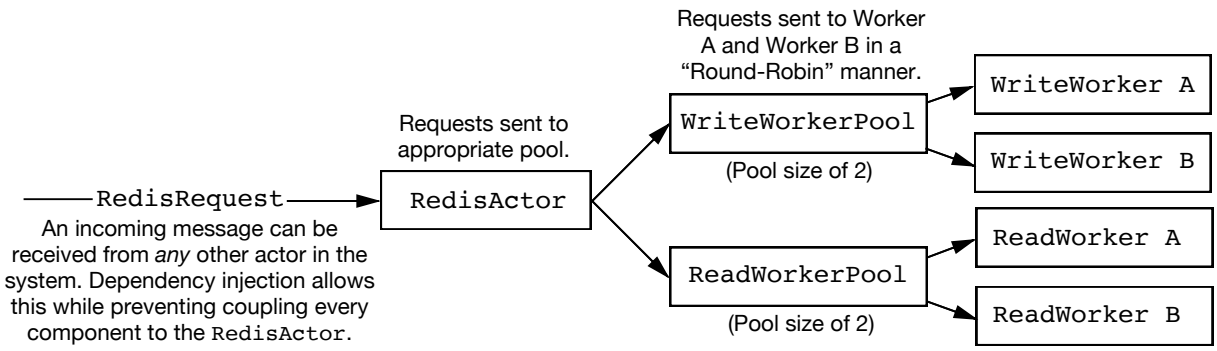

Figure 4.2: An example of how requests were load-balanced in order to prevent overworked actors and maximise concurrency. This is a lower level view into the inner workings of the "Feature Storage" component shown in Figure 3.1

- Requests to read and write features from Redis were partitioned by defining a `RedisRequest` type with two subtypes: `RedisReadRequest` and `RedisWriteRequest`. This allowed for the separation of reads and writes so that they can be processed by separate worker pools.
- Read and write requests were sent to a "routing" actor (named `RedisActor`) which maintains two pools of actors: one pool containing workers which deal with reading and unmarshalling data from Redis, and one containing workers which handle storing new features to Redis. These requests are distributed to workers using a Round-Robin algorithm, to ensure that the work is evenly distributed. Figure 4.2 shows how tasks were distributed between workers to maximise concurrency.

Akka made implementation of this load balancing simple since it provides extensive built-in support for such tasks, including protocols much more advanced than the Round-Robin method used in this component. The code for implementing a worker pool was a single line, as can be seen in the code for creating the `WriteWorkerPool` shown below.

Listing 4.1: Creating an actor pool to reduce the work required by any single actor.

```
val writingWorkerPool = context.actorOf(RoundRobinPool(4)
    .props(Props[RedisWriterWorker]), "writingWorkerPool")
```

After implementing the feature storage component using these techniques, it was found that it no longer restricted overall system performance.

4.1.3 Batched Feature Extraction

When a user becomes relevant and is displayed on an open result stream, their latest tweets are fetched (unless their timeline has already been processed recently). As a result, the number of tweets that we must process increases. If N users become relevant at once, we will typically have around $20N$ additional tweets to process on top of the tweets being processed from the stream. It is important in this case that there is no serious impact on performance.

To minimise the disruption caused by any sudden influx of new tweets, extraction of features from tweets from a user's timeline was performed using the `Future` asynchronous programming abstraction provided by Scala. A `Future` represents a value that may become available *at some point in the future*. Therefore, the value contained within a `Future` cannot be used until an associated computation is completed. A *callback function* can be used to access the value when it is available.

Since we want to avoid extracting features for the same tweet twice, we must check to see if the tweet has already been processed. To do this, we *ask* the `RedisActor` if this is the case by sending it a message asynchronously. This actor has to go and do additional work to figure out how to reply, and therefore will not be able to respond immediately. Since the response is delayed, a `Future` is an ideal abstraction. The annotated code below shows how for each status in the batch we check if it has been processed in a non-blocking way. (Note: the `?` syntax is how we asynchronously send another actor a message and capture the eventual response in a `Future`.)

Listing 4.2: Check asynchronously whether a status has already been processed.

```
var tweetAnalysisHistory =
   scala.collection.immutable.Set.empty[Future[(Long, Boolean)]]
// No iteration of this loop blocks waiting for a response, ensuring that
// every ask is sent immediately rather than after the response of the
// previous ask
tweets.foreach(tweet => {
   // Add the Future representing the redisActor's eventual response to the set
   tweetAnalysisHistory +=
      (redisActor ? HasStatusBeenProcessed(tweet)).mapTo[(Long, Boolean)]
})
```

`tweetAnalysisHistory` is a set of `Futures`, each of which will eventually contain a pair of the form (TweetID, Boolean), where the boolean value is true if the tweet already been processed. Only after *all* of these `Futures` are complete can we determine which tweets we have seen before and so will be discarded.

To determine when all of the `Futures` in the set are complete, we can *compose* them into a single combined `Future`. Specifically, the set of type `Set[Future[(Long, Boolean)]]` is mapped to the type `Future[Set[(Long, Boolean)]]` using the `Future.sequence` method. Thus, rather than writing a callback function for every status in the set (which itself is impossible since we cannot know its cardinality until run-time), we can write a single callback function to handle the case where all `Futures` are complete, and this gives us access to the `Set[(Long, Boolean)]` contained within. The code showing this process is shown below, and has been annotated with comments and types for clarity (adapted from `BatchFeatureExtraction.scala`).

Listing 4.3: Composing Futures unto a single future and registering the onComplete callback.

```
// Compose the Futures in tweetAnalysisHistory and register the
// callback to be fired when the composed Future completes.
Future.sequence(tweetAnalysisHistory) onComplete {
   case Success(seenBefore: Set[(Long, Boolean)]) =>
      // Future completed successfully, perform feature extraction
      // Original tweet list is filtered and feature extraction occurs here.
      ...
   case Failure(error) =>
      // Future unsuccessful, an actor may have missed response time guarantee
      // Error is logged
}
```

The feature extraction code itself (contained in the "Success" case of the pattern matching expression in the snippet above) is computationally expensive and as a result also relies heavily on asynchronous constructs, and uses many of the techniques described in this section.

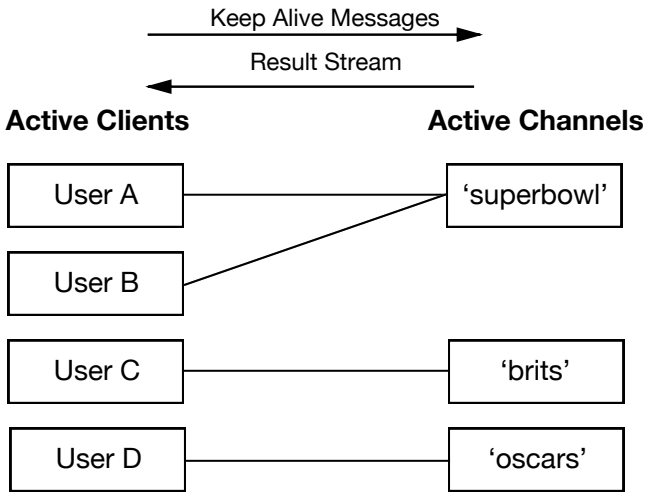

Figure 4.3: *An example of the one-to-many relationship that exists between users and channels*. Users A and B both connect to a single channel after entering the same query ("superbowl"), and will receive the results it outputs. This channel will perform the exact same actions regardless of the number of users connected.

4.1.4 Indexing

Tweets received at the indexing component are stored in a Terrier `MemoryIndex`. This is a special type of index that can be updated online, and therefore is especially suited to this application. Each document stored in the index consists of one or more tweets from a single Twitter user. If a tweet appears in the stream authored by a user that we have not seen before then we store the author's username, screen name, and profile biography in a Terrier metadata index so that it can be quickly retrieved in the case that the document is returned in response to a query.

4.1.5 Retrieval

To maximise performance, it was important to ensure that two users entering the same query does not result in the system computing the same stream of results twice. The traditional means of preventing this is through caching query results. Unfortunately, caching is not as practical in a system that is performing real-time indexing and providing results as a stream, since the results of a query change so frequently.

To get around this limitation, we map each new query to a "channel", that the results of that query are forwarded through. When a user inputs a query to the system, they are connected to a channel and will see the same stream of results for that query as any other user who has entered the query. If the channel corresponding to their query does not already exist, then it is created. Figure 4.3 gives a simplified example of what the clients/channels relationship may look when the system is running.

When a channel is created, it registers a scheduled request to poll Terrier for the latest result set in a configurable interval. The code below shows how the polling rate was read from a dependency injected configuration file in a way that avoids `null` values, and then used to schedule the channel's work.

Listing 4.4: Reading configuration from a file and scheduling the updates of results.

```
// Read the polling duration from the config file
val resultPollDuration =
    config.getInt("results.pollingrate").getOrElse(1000)

// Schedule the requests for the latest query results
```

```
val fetchTick = context.system.scheduler
     .schedule(Duration.Zero, FiniteDuration(resultPollDuration,
        TimeUnit.MILLISECONDS), self, FetchLatestQueryResults)
```

Through repeated polling, the result sets obtained from Terrier were transformed into a stream of results and sent to clients. These result sets are calculated using the BM25 weighting model, which is a popular information retrieval model known to generally give relevant results across a variety of collections. Further investigation would be required to determine the most effective retrieval technique.

Channels will remain open for a query for as long as at least one user is connected to the result stream. In order to determine whether any user is interested in the channel, connected users automatically send special "keep-alive" messages that the server interprets as a user expressing a desire to keep the channel open. Each open channel monitors the timestamp associated with the latest keep-alive receieved, and will free its associated resources if the latest keep-alive is ever older than some configurable threshold. In this way we ensure that a channel is never closed while a user is viewing a stream of results. Using this technique also allows us to close result streams when they are no longer required and therefore prevents resource leaks.

4.1.6 Metrics Reporting

This component is implemented as a single actor which can receive data from any other component in the system. By sending information to this actor, and specifying how it should translate this information (e.g. a Scala case class) to JSON, it allows any component to communicate with the client. In the example shown below, the `MetricsReporting` actor receives a message from some other actor informing it of the number of Twitter users that have been processed, and sends this message to all connected clients.

Listing 4.5: Forwarding new messages through the WebSocket to connected clients.

```
override def receive = {
  case numUsers @ NumberOfUsersSeen(_) =>
     channelMeta.channel push Json.toJson(numUsers)
  ...
}
```

The code which maps user-defined Scala types to valid JSON relies on the Play Framework's "Writes" converters. The converter for the example above is shown below:

Listing 4.6: Code for conversion of the NumberOfUsersSeen Scala case class to JSON format.

```
implicit val numberOfUsersSeenWrites = new Writes[NumberOfUsersSeen] {
  def writes(numberOfUsersSeen: NumberOfUsersSeen) = Json.obj(
     "numUsersSeen" -> numberOfUsersSeen.numUsers
  )
}
```

4.1.7 Real-time Trending Hashtags

One of the more interesting features to implement was the real-time hashtag usage counter. This component counts the number of occurrences of each hashtag within a configurable window of time, and updates this number as more tweets are processed. The user interface of this aspect of the system is shown in Figure 4.5. The code which generates the data for this component is written using Apache Spark Streaming, and consists of several

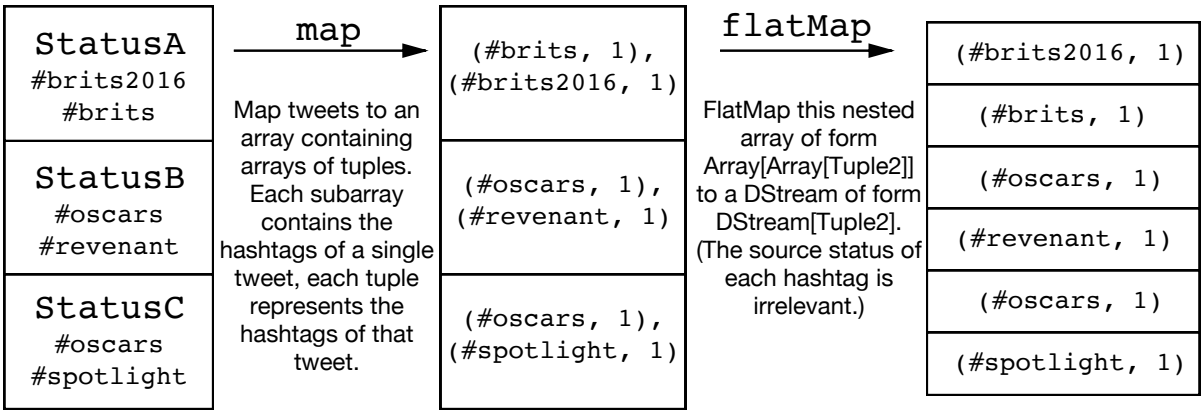

Figure 4.4: Example of how a tweet stream is mapped to a stream of pairs in the initial step of the real-time hashtag counting process.

steps which in combination produce an output of `(hashtag, count)` pairs which are then sent to connected clients for rendering.

The initial step of this process maps the incoming stream of tweets into a stream of `(String, Int)` tuples, where the string is the text of the hashtag and the integer is the value 1. Figure 4.4 shows this process diagrammatically, and the code used in this mapping process is shown below.

Listing 4.7: Initial mapping of tweets to hashtags pairs.

```
// Map the tweets in the stream to a stream of (hashtag, 1) tuples
val hashtags = stream flatMap(status => {
  status.getHashtagEntities map(hashtag => (hashtag.getText toLowerCase, 1))
})
```

The new stream of tuples is then *reduced* within a sliding window using the `reduceByKeyAndWindow` operation provided by Spark Streaming. The reduction takes place based on the key of the tuple (the first element; in this case the hashtag), and is analogous to a "`GROUP BY`" clause in SQL combined with a custom *reducer function* which tells Spark how to reduce/combine two values from the stream into one. Since the stream is an array of tuples of the form `(<hashtag>, 1)`, after grouping by the hashtags, we want to reduce these tuples by summing their values in order to count the total number of times the hashtag has occurred in the stream (within the current window). Therefore, our reducer function is a simple addition:

```
(p: Int, q: Int) => p + q
```

After applying this reduction, the data in the stream is now of the required form. After sorting the stream, a subset of the results are sent to the client to be displayed to connected users. The code which performs windowed reduction, sorts the results, and sends those results to all connected clients is shown below.

Listing 4.8: Counting the hashtags used within a temporal window.

```
// Aggregate hashtags within the window
val hashtagCountInWindow = hashtags
  .reduceByKeyAndWindow(
    (p:Int, q:Int) => p+q, Minutes(trendingHistoryMins),
       Seconds(reportFrequency)
  )
  .map{case (hashtag, count) => (count, hashtag)} // Reverse tuples
  .transform(_.sortByKey(ascending = false)) // Sort by counts
  .map{case (count, hashtag) => HashtagCount(hashtag, count)}
```

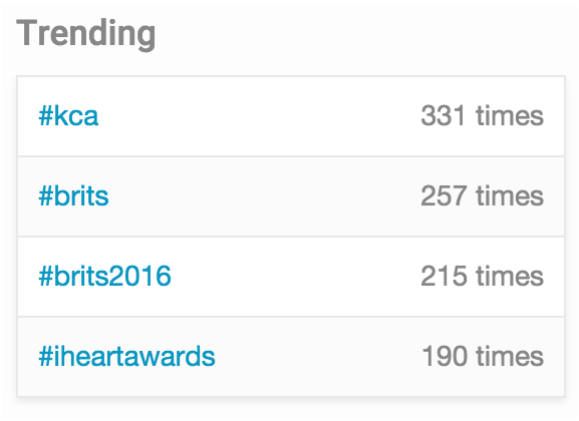

Figure 4.5: The trending hashtags component. This screenshot was taken during the 2016 BRIT awards and the Kids Choice Awards.

```
// Send latest trending data to connected clients
hashtagCountInWindow foreachRDD(rdd => {
  val hashtagCounts = rdd.collect take numberOfHashtagsToShow
  metricsReporting ! TrendingHashtags(hashtagCounts.toList)
})
```

4.2 Client Implementation

4.2.1 Tools & Practices

LESS

LESS is a compiled superset to CSS (Cascading Style Sheets) which introduces a number of additional features including variables, nested definitions, and a module/import system.

CSS Flexbox

Flexbox is method for laying out elements on a page. The majority of user interface elements used in "Who To Follow In Context" are custom built rather than from an external component library, and Flexbox was used to align child elements within these components.

CommonJS Modules

CommonJS introduces a module system into JavaScript/TypeScript. As a result, it avoids the need to add one script tag to our index HTML page for each TypeScript file created. During the compilation step, the dependency tree of the front-end application is scanned and bundled into a single file.

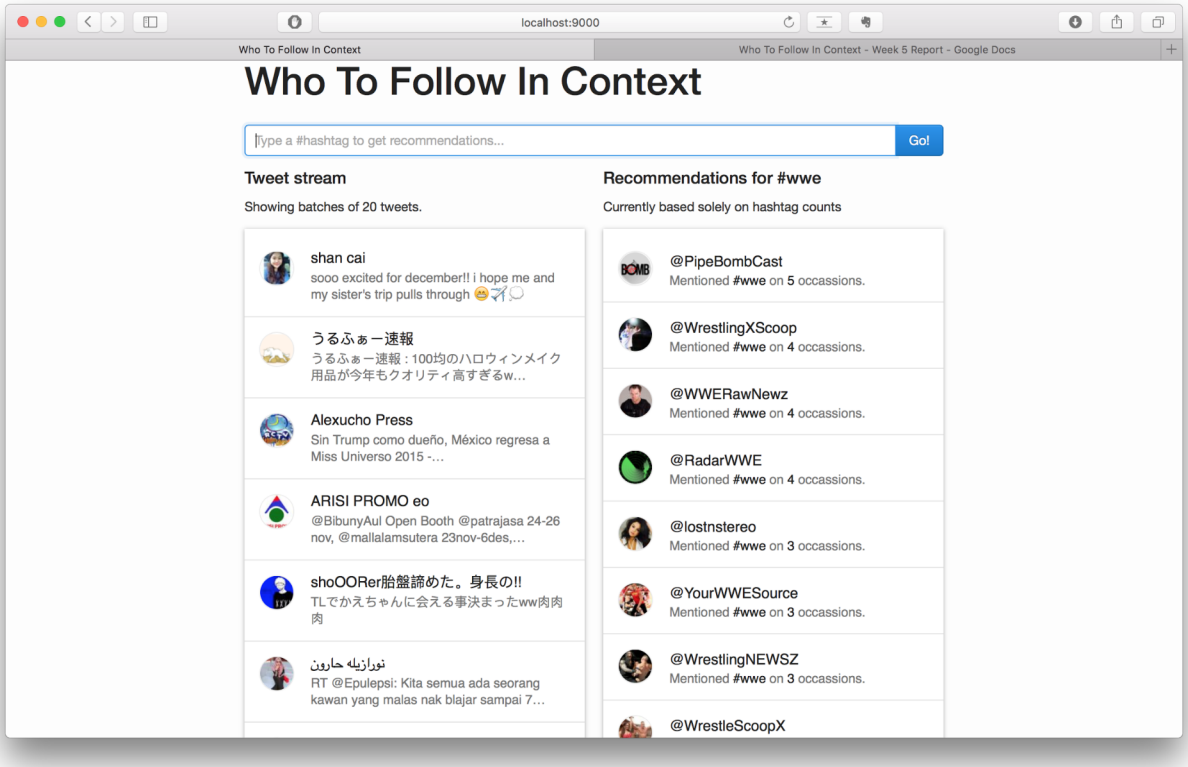

Figure 4.6: An early user interface prototype.

Gulp

Gulp is a front-end build system which was used to speed up parts of development. Gulp tasks were defined to watch TypeScript files for changes and automatically transpile them into new JavaScript files using the "tsc" TypeScript compiler. A Gulp task was also used in order to compile LESS files to CSS so that they could be interpreted by browsers.

Node Package Manager

Node Package Manager (NPM) was used to manage front-end dependencies. By specifying these dependencies in a `package.json` file, developers can simply run `npm install` to fetch all of the requirements for the application. NPM was used in conjunction with the *TypeScript Definition Manager* which fetches TypeScript type declaration files.

4.2.2 User Interface

The user interface was one of the most heavily iterated upon aspects of the application. Using a component-based user interface framework allowed components to be iterated on in isolation, and so changing the UI could be done gradually to allow for progress on other features.

An early prototype of the user interface (show in Figure 4.6 displayed the stream of incoming tweets on the left half of the window, and the recommendations for the current query on the right hand side. The ability to

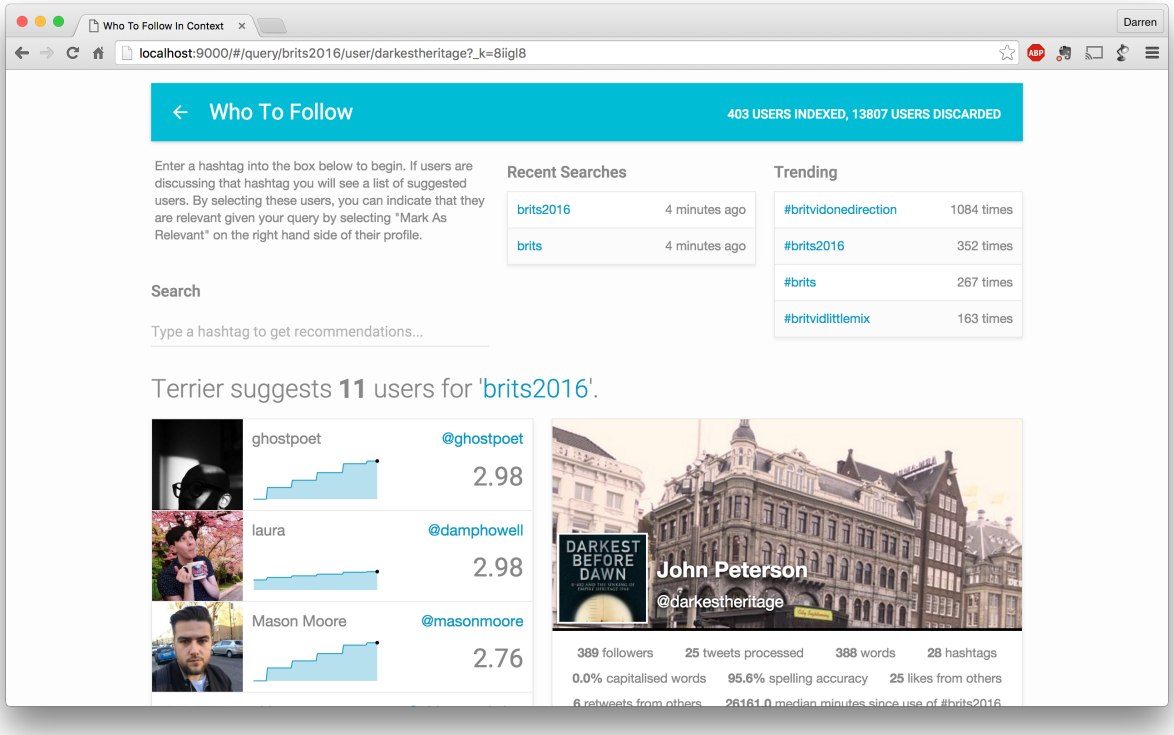

Figure 4.7: The final iteration of the user interface.

view user timelines was not included at this point. After multiple iterations and further refinement of the project requirements, the final version of the user interface was created (see Figure 4.7).

4.2.3 Searching

When a user enters a hashtag Q into the search box and presses enter, an event handler is fired which changes the URL to `/#/query/Q`. Since we are using React Router, the hierarchy of components mounted on the screen relates directly to the current URL. The code which defines this hierarchy of components and their associated URLs is shown below.

Listing 4.9: Definition of the hierarchy of React component and how they map to the URL.

```
ReactDOM.render((
   <Router>
      <Route component={App}>
         <Route path="/" component={Home}>
            <Route path="/query/:query" component={QueryResults}>
               <Route path="/query/:query/user/:screenName"
                  component={TwitterUserPreviewPane} />
            </Route>
         </Route>
      </Route>
   </Router>
), document.getElementById("wtfc-app-mount"));
```

As a result of the code above, any requests to a URL beginning with `/#/query` results in the `QueryResults` component and all of its parents to be mounted. Immediately as `QueryResults` mounts, React calls its

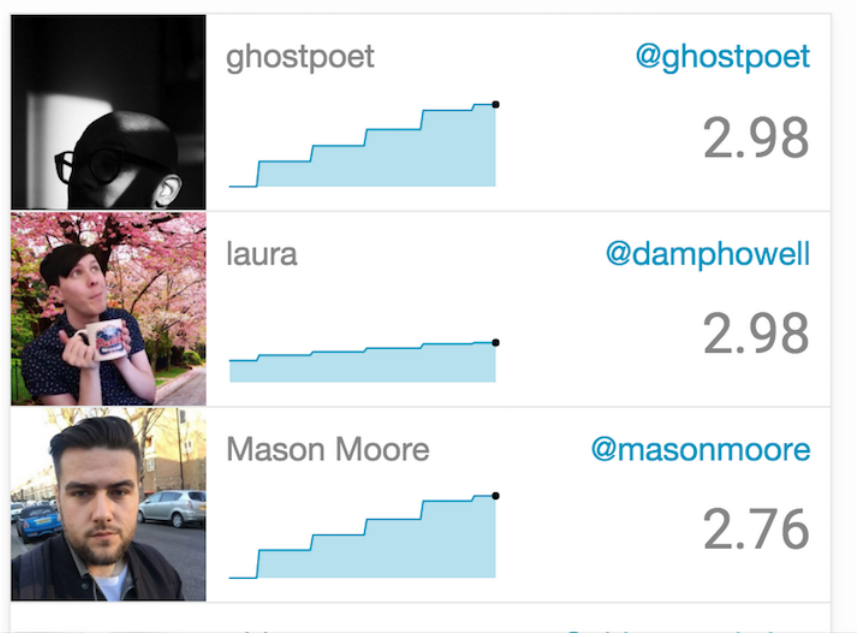

Figure 4.8: An example of the display of suggested users.

`componentDidMount` method, shown below. Inside this method, we send a request to the server to start the WebSocket handshake, and the server will create the channel for this query. In addition, we register callback functions which will execute when the client receives data from the server through the connected WebSocket, and the keep-alive messages that the client will send to the server for as long as `QueryResults` remains mounted.

Listing 4.10: The method called by React when a new component mounts.

```
componentDidMount: function() {
   // Creates WebSocket, keep-alive scheduled, registers callbacks
   this._setQueryChannel(this.props.params.query);
},
```

4.2.4 Query Results

Query results are presented as a list on the left hand side of the display. Figure 4.8 shows how the query results are displayed. As the score for each user is re-evaluated, the ordering of the results changes. The numbers shown on the right hand side of the suggestions represent the users relevance score. The graph shown on each suggestion shows the historical changes in score for the corresponding user.

4.2.5 Encouraging Interaction

At the top right hand corner of the screen we display real-time lists of the latest queries which have been entered by users of the system. We also display the most popular hashtags found within the past 10 minutes of streaming. The aim of these components is to encourage users to interact with the system by providing them with up-to-the-moment information on what is popular on Twitter and what users are currently searching for. By selecting an item in the list, it is entered as a query. It was hoped that this feature would make people more likely to interact with the system, since it may provide them with suggestions for hashtags they could search for.

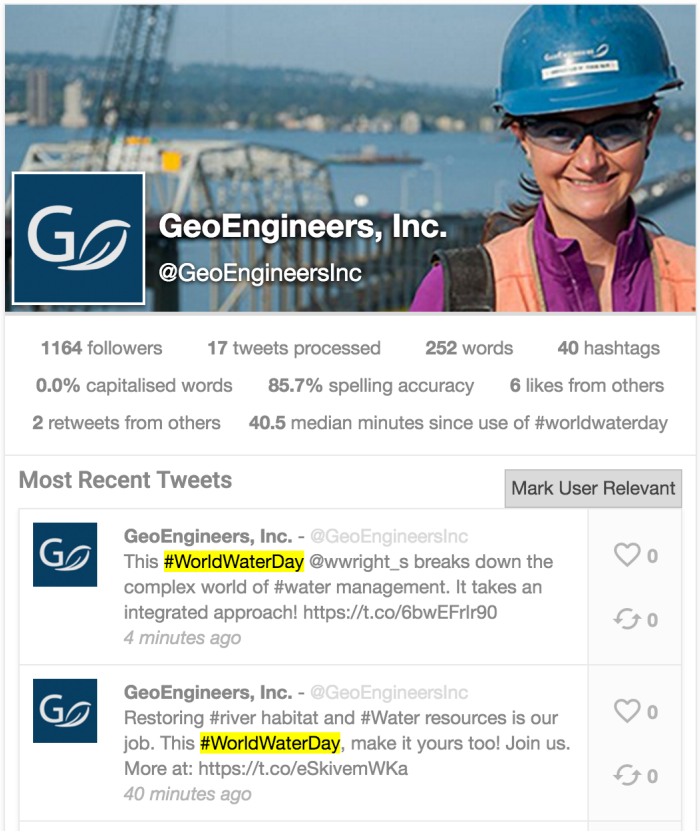

Figure 4.9: The preview of a user's Twitter profile that is shown when a suggestion is selected.

4.2.6 Previewing Twitter Users

On selecting a suggested account, users are shown a preview of that account without having to leave the application. Figure 4.9 shows how we display the preview to users. By previewing the suggestion within the application, users can provide relevance feedback for the purposes of evaluation and improving the relevance of future query results. Any uses of the query term are highlighted to bring attention to potentially relevant tweets.

Chapter 5

Evaluation

5.1 Experiments

5.1.1 Experimenting With Filtering Controls

The implications of altering the threshold at which tweets are discarded during the filtering stage were examined. It was hypothesised that the "Spelling Accuracy" and "Follower Count" filters would have the largest impact on the number of users discarded. By altering this configuration, the effect of filtering on the number of users the application has to process was investigated. Using the default filtering controls shown in Table 5.2, it was found that in a sample of 15000 tweets, approximately 97% of users were discarded at the filtering stage. This shows that the combination of seemingly relaxed filters can result in a large number of users being dropped and therefore massively reducing the load on the server.

Further experiments were performed to determine the impact that the individual default filters have on the overall number of users filtered. This experiment was performed by disabling all user filtering, and examining the effects of individual filters by applying them in isolation. The determined contribution of the individual default filtering controls is shown in Table 5.1. The total percentage of users filtered is greater than 100% because a single tweet will usually fail the checks imposed by several filters at once.

5.1.2 Investigating the Effects of Stream Replay Rate

The rate at which tweets are read when using a source file has a direct effect on system performance and usability. An investigation was conducted into the performance and usability impact of altering the *stream replay rate*. The stream replay rate is the rate at which new tweets are read into the system from a source file. This rate is configurable using two parameters: "BatchDuration" and "BatchSize". The "BatchDuration" parameter defines the time between examining batches of tweets from the source. The "BatchSize" parameter defines the number

Filter Name	% of Users Filtered
ALL FILTERS	97
Spelling Accuracy	92
Follower Count	44
Word Count	30

Table 5.1: The individual impact of different filtering controls. ($N = 15000$).

Filter Name	Required Value	Description
Spelling Accuracy	$> 50\%$	Tweets with 50% or less of words spelt correctly are discarded.
Word Count	> 3	Tweets with 3 or less words are discarded.
Follower Count	≥ 300	Tweets with authors who have less than 300 followers are discarded.
Hashtag Count	< 5	Tweets with 5 or more hashtags are discarded.
Capitalised Words	N/A	Tweet entirely capitalised are discarded.

Table 5.2: The default filtering controls.

of tweets in each of these batches. These parameters can be configured using the file entitled "application.conf", and are shown in Listing 5.1. Users were given the opportunity to perform tasks with the system with two different stream replay rates, and asked to provide feedback on which version they preferred.

Listing 5.1: Configuration of the stream replay rate in application.conf

```
# path to gzipped json file containing tweets which will be read
# leave blank for live stream
stream.sourcefile.path = "/path/to/gzipped/tweets.json.gz"
# the number of tweets in each batch
stream.sourcefile.batchSize = 100
# time in milliseconds between each batch
stream.sourcefile.batchDuration = 500
```

5.2 User Evaluation

In addition to weekly feedback from the project supervisor, a user evaluation study was conducted in the final weeks of the project.

5.2.1 Questionnaires

In the evaluation phase of the project, two separate questionnaires were used. The first aimed to determine the experiences that participants have when they use Twitter, and the second was a post-evaluation questionnaire enabling them to provide additional feedback after the think-aloud session.

5.2.2 Usability Study

Computer usability questionnaire, think-aloud during evaluation to determine issues encountered...

5.2.3 Relevance Judgements

What users thought of results, did they make relevance judgements often?

5.3 Behaviour Testing

5.3.1 Server

Testing on the server...

5.3.2 Client

The front-end of the application was tested using the *Jest* testing and mocking library...

5.4 Integration Testing

In addition to tests which ensure that components behave as expected, tests were also written to ensure that they work correctly within their intended environment. In the context of actor systems, this is done by testing how actors respond to *stimulus* from a *probe*. Since actors perform computation in response to messages, we can probe their behaviour by sending them test messages and ensuring that they respond as expected.

As an example of how integration testing was useful: When a `SparkContext` has been started, it is illegal to attempt to register additional actions on the data. Since the system registers Spark actions in asynchronous calls, it is a possibility that the `SparkContext` is started before all actions are registered. To avoid this scenario, any actor registering Spark actions must inform its parent when it has done so. The parent will wait for all child actors to make this confirmation before starting the context. Integration testing is used to ensure that all actors which register Spark actions correctly inform their parent when this registration is complete, and thus avoids a runtime `IllegalStateException`.

5.5 Performance Testing

5.5.1 Actor Message Response Time Testing

5.5.2 Server Throughput Testing

5.5.3 Profiling Client Side Performance

765 rules (95%) of CSS not used by the current page.

Use Chrome devtools and painting and profiling info here... Read the Google Developers tutorial again.

Chapter 6

Conclusion

6.1 Reflecting on Requirements

6.2 Future Work

6.3 Reflection

Appendices

Appendix A

Running the Programs

An example of running from the command line is as follows:

```
> java MaxClique BBMC1 brock200_1.clq 14400
```

This will apply $BBMC$ with $style = 1$ to the first brock200 DIMACS instance allowing 14400 seconds of cpu time.

Appendix B

Generating Random Graphs

We generate Erdós-Rënyi random graphs $G(n, p)$ where n is the number of vertices and each edge is included in the graph with probability p independent from every other edge. It produces a random graph in DIMACS format with vertices numbered 1 to n inclusive. It can be run from the command line as follows to produce a clq file

```
> java RandomGraph 100 0.9 > 100-90-00.clq
```

Bibliography

[1] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *Proceedings IJCAI'91*, pages 331–337, 1991.

[2] Brian Hayes. Can't get no satisfaction. *American Scientist*, 85:108–112, 1997.